CS406-Homework 1
Matrix-Matrix, Matrix-Vector Multiplication

Görkem Yar 27970

March 2024

1 Introduction

The aim of this homework assignment is to optimize the performance of the matrix-matrix and matrix-vector multiplication operations in C++. In the coming sections, the effect of the compiler optimizations and SIMD will be discussed. Also, the effect of the different datatypes, specifically float and double, is another metric that this report will cover.

For bought matrix-matrix multiplication and matrix-vector multiplication, I have developed 5 different algorithms. From these 5 algorithms, I will introduce 2 algorithms for matrix-vector multiplication and 3 algorithms for matrix-matrix multiplication in the Algorithms & Results section. To check other algorithms, you can look at my code archive.

In this report, I cannot put all the results such as the instruction count, L1 cache misses, TLB misses, and LLC misses. For this reason, the algorithms will be compared based on the GFLOPS and the execution time. For the other information, I am adding a results folder where each algorithm runs with different data types and different optimization levels. In the Discussion section, I will compare the performances and elaborate on the reasons.

2 Methodology

This report details the development and optimization of matrix-matrix and matrix-vector multiplication kernels, focusing on optimization in C++ and SIMD-AVX instructions to enhance computational efficiency. The goal is to explore and implement optimization techniques that could impact the performance of these fundamental linear algebra operations.

3 Algorithms & Results

3.1 Matrix-Vector Multiplication

3.1.1 Basic Matrix-Vector Multiplication

This is the simplest way to make a matrix-vector multiplication.

```
template<typename T>
void matrix_vector_basic(T** mat, T* b,T* res, int N){
    for (int i = 0; i < N; i++){
        T tmp = 0;
        T* a = mat[i];
        for (int j = 0; j < N; j++){
            tmp += a[j] * b[j];
        }
        res[i] = tmp;
}}
```

3.1.2 SIMD Matrix-Vector Multiplication

This algorithm is implemented using SIMD. SIMD instructions enable doing multiple operations at once. I used simd256 which can do operations for 256 bits which is equivalent to 8 floats or 4 doubles. With simd256 we will able to make 8 float or 4 double operations at once. As a result, the overall performance of the algorithm increased.

```
void mat_vector_simd_256_float(float** mat, float* b, float* res, int N){
    for (int i = 0; i < N; i++){
        float* a = mat[i];
        float* b_c = b;
        __m256 summ = _mm256_setzero_ps();
        const float* aEnd = a + N;
        for(; a < aEnd; a += 8, b_c += 8){
            const __m256 a_data = _mm256_loadu_ps(a);
            const __m256 b_data = _mm256_loadu_ps(b_c);
            const __m256 mul_data = _mm256_dp_ps(a_data, b_data, 0xFF );
            summ = _mm256_add_ps(summ, mul_data);
        }
        const __m128 low = _mm256_castps256_ps128(summ);
        const __m128 high = _mm256_extractf128_ps(summ, 1);
        const __m128 result = _mm_add_ss(low, high);
        float sum = _mm_cvtss_f32(result);
        res[i] = sum;
}}
```

Double implementation is much similar.

The role of the AVX instructions above (the explanation is taken from the Intel Intrinsic guide):

- **_mm256_setzero_ps:** Initializes a 256-bit register with all bits set to zero.
- **_mm256_loadu_ps:** Load 256-bits (composed of 8 packed single-precision (32-bit) floating-point elements) from memory into the destination.
- **_mm256_dp_ps:** Conditionally multiply the packed single-precision (32-bit) floating-point elements in a and b using the high 4 bits in imm8, sum the four products, and conditionally store the sum in the destination
- **_mm256_add_ps:** Add packed single-precision (32-bit) floating-point elements in a and b, and store the results in dst.
- **_mm256_castps256_ps128 and _mm256_extractf128_ps:** Gets low and high 128 bits from a 256-bit register to a 128 bit register, respectively.
- **_mm_add_ss:** Add the lower single-precision (32-bit) floating-point element in a and b, store the result in the lower element of destination and copy the upper 3 packed elements from a to the upper elements of destination.
- **_mm_cvtss_f32:** Copy the lower single-precision (32-bit) floating-point element of a to dst

3.1.3 Results for Matrix-Vector Multiplication

Performance comparison between Basic and SIMD256 implementations across different matrix sizes, with different data types and Optimization Levels

Table 1: Float with Optimization 0

Matrix Size	Implementation	GFLOPS	Time (seconds)
128	Basic	0.007739253	0.004234
128	SIMD256	0.008074913	0.004058
512	Basic	0.041909512	0.012510
512	SIMD256	0.045933765	0.011414
1024	Basic	0.076008553	0.027591
1024	SIMD256	0.073871992	0.028389
2048	Basic	0.077802687	0.107819
2048	SIMD256	0.100716877	0.083289
4096	Basic	0.104999349	0.319568
4096	SIMD256	0.123597262	0.271482

Table 2: Float with Optimization 3

Matrix Size	Implementation	GFLOPS	Time (seconds)
128	Basic	0.007779677	0.004212
128	SIMD256	0.007450659	0.004398
512	Basic	0.051899425	0.010102
512	SIMD256	0.039712770	0.013202
1024	Basic	0.077291563	0.027133
1024	SIMD256	0.086759556	0.024172
2048	Basic	0.104765929	0.080070
2048	SIMD256	0.104040879	0.080628
4096	Basic	0.140579218	0.238687
4096	SIMD256	0.170162086	0.197191

Table 3: Double with Optimization 0

Matrix Size	Implementation	GFLOPS	Time (seconds)
128	Basic	0.007659654	0.004278
128	SIMD256	0.008053084	0.004069
512	Basic	0.050281768	0.010427
512	SIMD256	0.069121687	0.007585
1024	Basic	0.053377586	0.039289
1024	SIMD256	0.055028916	0.038110
2048	Basic	0.083323645	0.100675
2048	SIMD256	0.124545060	0.067354
4096	Basic	0.095730586	0.350509
4096	SIMD256	0.109771594	0.305675

Table 4: Double with Optimization 3

Matrix Size	Implementation	GFLOPS	Time (seconds)
128	Basic	0.008406362	0.003898
128	SIMD256	0.008149216	0.004021
512	Basic	0.038971827	0.013453
512	SIMD256	0.058605857	0.008946
1024	Basic	0.070480658	0.029755
1024	SIMD256	0.105713882	0.019838
2048	Basic	0.115946426	0.072349
2048	SIMD256	0.178113425	0.047097
4096	Basic	0.173809430	0.193053
4096	SIMD256	0.127853683	0.262444

3.2 Matrix-Matrix Multiplication

3.2.1 Basic Matrix-Matrix Multiplication

The simplest way to implement a matrix-matrix multiplication.

```
template <typename T>
void matrix_mult_simple(T** mat1, T** mat2, T** res, int N){
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            res[i][j] = 0;
            for (int k = 0; k < N; k++) {
                res[i][j] += mat1[i][k] * mat2[k][j];
}}}}
```

3.2.2 Faster Matrix-Matrix Multiplication

This is again a very simple implementation that goes over all the indices; however, by changing the order of the loop it enables consecutive memory blocks to be accessed. As a result, this implementation is faster.

```
template <typename T>
void matrix_mult_simple_faster(T** mat1, T** mat2, T** res, int N){
    for (int i = 0; i < N; i++) {
        for (int k = 0; k < N; k++) {
            for (int j = 0; j < N; j++) {
                res[i][j] += mat1[i][k] * mat2[k][j];
}}}}
```

3.2.3 SIMD Matrix-Matrix Multiplication

Matrix-Matrix multiplication exercises the SIMD, AVX instructions.

```
void simd_matrix_mult_with_transpose_float(float** A, float** B, float** C, int N){
    float temp[8];
    __m256 sum, a, b;
    int i, j, k;
    float* adx, *bdx;
    for (i = 0; i < N; i++) {
        adx = A[i];
        for (j = 0; j < N; j++) {
            bdx = B[j];
            sum = _mm256_setzero_ps();
            for (k = 0; k < N; k += 8) {
                a = _mm256_loadu_ps(adx + k);
                b = _mm256_loadu_ps(bdx + k);
                sum = _mm256_add_ps(sum, _mm256_mul_ps(a, b));
            }
            sum = _mm256_hadd_ps(sum, sum);
            sum = _mm256_hadd_ps(sum, sum);
            _mm256_storeu_ps(temp, sum);
            C[i][j] = temp[0] + temp[4];
}}}
```

Implementation for the double is similar. After getting the transpose of the matrix. We can access the same index locations. This helps us to take advantage of the spatial locality (data in close memory locations). In each iteration, with the load function, 8 consecutive floats from each matrix are loaded. With the multiply operation, these 8 floats from each matrices multiplied. After summing up the entire row, we sum the elements and store them in the result matrix. **To use this algorithm one needs to get the transpose of the second matrix first.**

3.2.4 Results for Matrix-Matrix Multiplication

Performance and Execution Time for Different Matrix Multiplication Implementations

Table 5: Float, Optimization Level 0

Test Name	Matrix Size	GFLOPS	Time (seconds)
matrix_mult_simple	128	0.17296160	0.024273
matrix_mult_simple_faster	128	0.24752384	0.016945
simd_matrix_mult_with_transpose	128	1.1908864	0.003522
matrix_mult_simple	512	0.2847232	0.942791
matrix_mult_simple_faster	512	0.38358016	0.699814
simd_matrix_mult_with_transpose	512	1.899783424	0.141286
matrix_mult_simple	1024	0.282165632	7.610688
matrix_mult_simple_faster	1024	0.395438848	5.433923
simd_matrix_mult_with_transpose	1024	2.094950400	1.025076
matrix_mult_simple	2048	0.173217280	99.237506
matrix_mult_simple_faster	2048	0.412671232	41.630939
simd_matrix_mult_with_transpose	2048	2.141475328	8.022948
matrix_mult_simple	4096	0.119224320	1152.468583
matrix_mult_simple_faster	4096	0.413917440	332.236590
simd_matrix_mult_with_transpose	4096	2.211405440	62.113672

Table 6: Float, Optimization Level 3

Implementation	Matrix Size	GFLOPS	Time (seconds)
matrix_mult_simple	128	0.429040916	0.009776
matrix_mult_simple_faster	128	0.767063643	0.005468
simd_matrix_mult_with_transpose	128	0.669268230	0.006267
matrix_mult_simple	512	0.943932259	0.284380
matrix_mult_simple_faster	512	7.679897462	0.034953
simd_matrix_mult_with_transpose	512	5.477939228	0.049003
matrix_mult_simple	1024	1.112686981	1.929998
matrix_mult_simple_faster	1024	10.274353143	0.209014
simd_matrix_mult_with_transpose	1024	7.431202109	0.288982
matrix_mult_simple	2048	0.597943163	28.731609
matrix_mult_simple_faster	2048	8.195278655	2.096313
simd_matrix_mult_with_transpose	2048	7.298023819	2.354044
matrix_mult_simple	4096	0.222081184	618.868068
matrix_mult_simple_faster	4096	5.326561660	25.802565
simd_matrix_mult_with_transpose	4096	5.472853241	25.112852

Table 7: Double, Optimization Level 0

Implementation	Matrix Size	GFLOPS	Time (seconds)
matrix_mult_simple	128	0.204490468	0.020511
matrix_mult_simple_faster	128	0.215158715	0.019494
simd_matrix_mult_with_transpose	128	0.375262056	0.011177
matrix_mult_simple	512	0.278762201	0.962955
matrix_mult_simple_faster	512	0.385460489	0.696402
simd_matrix_mult_with_transpose	512	0.977703923	0.274557
matrix_mult_simple	1024	0.281474627	7.629404
matrix_mult_simple_faster	1024	0.419033954	5.124844
simd_matrix_mult_with_transpose	1024	1.002672404	2.141760
matrix_mult_simple	2048	0.136513375	125.847516
matrix_mult_simple_faster	2048	0.419270184	40.975652
simd_matrix_mult_with_transpose	2048	0.987961851	17.389203
matrix_mult_simple	4096	0.107218182	1281.862373
matrix_mult_simple_faster	4096	0.410861144	334.514362
simd_matrix_mult_with_transpose	4096	0.969999005	141.689788

Table 8: Double, Optimization Level 3

Implementation	Matrix Size	GFLOPS	Time (seconds)
matrix_mult_simple	128	0.564129657	0.007435
matrix_mult_simple_faster	128	1.382889548	0.003033
simd_matrix_mult_with_transpose	128	1.216798375	0.003447
matrix_mult_simple	512	0.922258947	0.291063
matrix_mult_simple_faster	512	4.621026958	0.058090
simd_matrix_mult_with_transpose	512	2.918633251	0.091973
matrix_mult_simple	1024	1.102490928	1.947847
matrix_mult_simple_faster	1024	5.915721056	0.363013
simd_matrix_mult_with_transpose	1024	4.064663464	0.528330
matrix_mult_simple	2048	0.485514685	35.384860
matrix_mult_simple_faster	2048	2.988463396	5.748730
simd_matrix_mult_with_transpose	2048	3.059006909	5.616159
matrix_mult_simple	4096	0.218024395	630.383372
matrix_mult_simple_faster	4096	2.532985008	54.259679
simd_matrix_mult_with_transpose	4096	2.824848394	48.653568

4 Discussion

4.1 Effect of Optimization Levels

Optimization levels are methods that are employed by compilers to improve the efficiency of the generated machine code without changing the semantics of the code. Generally, these optimization techniques range from simple code transformations, like dead code elimination, to even complex methods such as loop unrolling. There are 5 optimization levels, which are -O0 (no optimization), -O1, -O2, -O3, and -Ofast. Each level enables more optimizations to increase the execution speed.

Observations from Results: After examining the results from the implementations for the matrix-vector and matrix-matrix multiplication with different compiler optimization levels, it becomes clear that optimization levels have a significant impact on performance. For instance, transitioning from -O0 to -O3 often results in marked improvements in GFLOPS and reductions in execution time. This is observable across both basic and SIMD-enabled implementations, underscoring the compiler's role in increasing efficiency. Looking at the examples of matrix-matrix multiplication with datatype float, the effect of compiler optimizations is more than visible. In almost all the test cases the execution time is at least reduced by 50% and the GFLOPS is increased 2 times.

Especially, **the matrix_mult_simple_faster** function improved a lot by the compiler optimizations. This may stem from the inner loop unrolling or prefetching and cache utilizations. The compiler and CPU can work together to prefetch data into the cache before it's needed. They can take advantage of temporal locality (reusing the same data within short periods) and spatial locality (reuse of data within close memory locations). This loop order may make it easier to take advantage of these access patterns for the compiler.

4.2 Effect of Datatypes

The choice of data type, particularly the single-precision (float) and double-precision (double), plays an important role in performance and precision. Floats are represented by 32-bit while Doubles are 64-bit. As it clear, the floats consume less memory memory and potentially they are faster in computations. This efficiency in memory and computation can be advantageous when processing large datasets (big matrices).

The smaller size of floats enhances cache efficiency, allowing for higher throughput as well as reduced memory bandwidth consumption. This advantage became clearer for the larger matrices as the time used for the float executions are much smaller than the double executions.

Throughout the test cases, float implementations achieved higher GFLOPS compared to double implementations, illustrating the direct impact of data type size on computational throughput. For example, in SIMD matrix-vector multiplication with a matrix size of 1024x1024, the float implementation outperformed the double implementation by approximately 85% in terms of GFLOPS.

4.3 Effect of Matrix Size

The effect of matrix size on performance is an important factor that directly affects the execution speed and resource utilization. As the size of the matrices increases, the computational complexity and memory requirements also became large. The theoretical complexity of matrix-vector multiplication is $O(n^2)$, while the matrix-matrix multiplication has a complexity of $O(n^3)$. Since the factor between consequent test cases is 2. Theoretical growth in time should be 4 and 8 for matrix-vector and matrix-matrix multiplications, respectively. In both matrix-vector and matrix-matrix results, this theoretical expectancy is met. The time growth is approximately 4 and 8 in the respective test cases.

4.4 SIMD(AVX) vs Basic Algorithms

SIMD makes computations faster by exploiting data-level parallelism in many computational tasks, especially if there is an array or vector that undergoes the same operation for all its indices.

In classical implementations, the CPU can handle one data element per instruction, meaning that loops iterates over each element sequentially. On the other hand, SIMD allows a single instruction to simultaneously perform the same operation on multiple data elements. For instance, if a SIMD register can hold eight 32-bit floating-point numbers, a single SIMD instruction can perform eight parallel additions. This reduces the number of instructions and iterations needed to process large datasets. For this reason, the algorithms that uses SIMD instructions overall performed much better than the classical implementations. If we examine table 5 compiler "Float, Optimization Level 0", the SIMD dominates other implementations by far.

5 Conclusion

Throughout this report, different matrix-vector and matrix-matrix multiplication algorithms are introduced. Their performance regarding the time and GFLOPS is measured and discussed with respect to compiler optimizations, datatypes (single-precision and double-precision), different-sized matrices, and classical implementations vs SIMD implementations. The effect of compiler optimization on GFLOPS and time showed that O3 had a significantly better performance compared to O0. Also, it is clarified that with single-precision the performance of the algorithms is much better. Additionally, the SIMD instructions and algorithms explained and demonstrated that they dominate the classical implementations.