Part 2: Performance Analysis and Optimization

1) Benchmarking

Baseline Performance of code in phase1benchmarking.cpp:

Dimension = 4 (500 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	706	707
Function 2	654	697
Function 3	3,070	3,070
Function 4	2,760	1,330

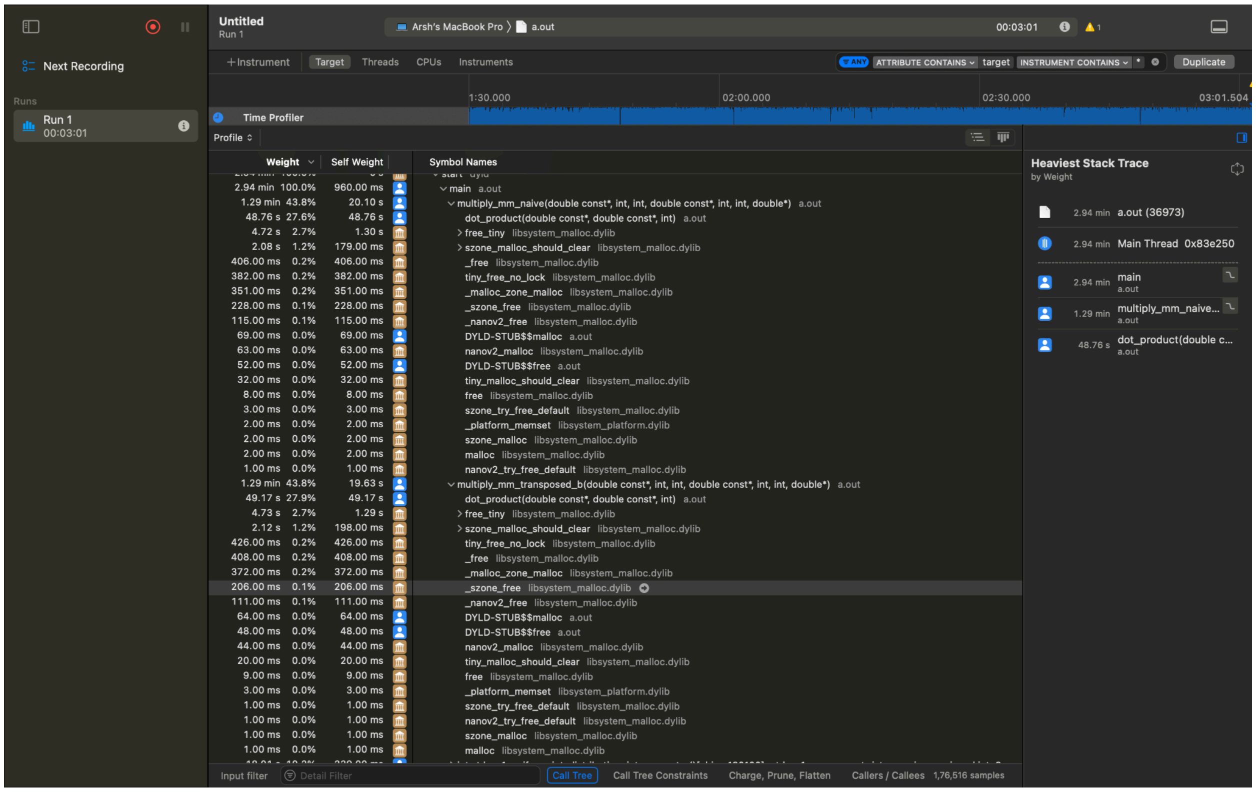
Dimension = 10 (500 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	2,430	3,120
Function 2	1,980	1,530
Function 3	19,100	89,70
Function 4	19,300	10,300

Dimension = 100 (100 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	92,200	34,800
Function 2	88,900	21,400
Function 3	9,040,000	4,360,000
Function 4	8,620,000	1,450,000

5) Using a Profiler to Identify HotSpots for Optimization



```
   Weight           Self Weight   Symbol Names
 2.94 min 100.0%    960.00 ms     main  a.out
 1.29 min  43.8%     20.10 s        multiply_mm_naive(double const*, int, int, double const*, int, int, double*)  a.out
  48.76 s  27.6%     48.76 s          dot_product(double const*, double const*, int)  a.out
   4.72 s   2.7%      1.30 s        > free_tiny  libsystem_malloc.dylib
   2.08 s   1.2%    179.00 ms       > szone_malloc_should_clear  libsystem_malloc.dylib
406.00 ms   0.2%    406.00 ms         _free  libsystem_malloc.dylib
382.00 ms   0.2%    382.00 ms         tiny_free_no_lock  libsystem_malloc.dylib
351.00 ms   0.2%    351.00 ms         _malloc_zone_malloc  libsystem_malloc.dylib
228.00 ms   0.1%    228.00 ms         _szone_free  libsystem_malloc.dylib
115.00 ms   0.1%    115.00 ms         _nanov2_free  libsystem_malloc.dylib
 69.00 ms   0.0%     69.00 ms         DYLD-STUB$$malloc  a.out
 63.00 ms   0.0%     63.00 ms         nanov2_malloc  libsystem_malloc.dylib
 52.00 ms   0.0%     52.00 ms         DYLD-STUB$$free  a.out
 32.00 ms   0.0%     32.00 ms         tiny_malloc_should_clear  libsystem_malloc.dylib
  8.00 ms   0.0%      8.00 ms         free  libsystem_malloc.dylib
  3.00 ms   0.0%      3.00 ms         szone_try_free_default  libsystem_malloc.dylib
  2.00 ms   0.0%      2.00 ms         _platform_memset  libsystem_platform.dylib
  2.00 ms   0.0%      2.00 ms         szone_malloc  libsystem_malloc.dylib
  2.00 ms   0.0%      2.00 ms         malloc  libsystem_malloc.dylib
  1.00 ms   0.0%      1.00 ms         nanov2_try_free_default  libsystem_malloc.dylib
 1.29 min  43.8%     19.63 s        multiply_mm_transposed_b(double const*, int, int, double const*, int, int, double*)  a.out
  49.17 s  27.9%     49.17 s          dot_product(double const*, double const*, int)  a.out
   4.73 s   2.7%      1.29 s        > free_tiny  libsystem_malloc.dylib
   2.12 s   1.2%    198.00 ms       > szone_malloc_should_clear  libsystem_malloc.dylib
426.00 ms   0.2%    426.00 ms         tiny_free_no_lock  libsystem_malloc.dylib
408.00 ms   0.2%    408.00 ms         _free  libsystem_malloc.dylib
372.00 ms   0.2%    372.00 ms         _malloc_zone_malloc  libsystem_malloc.dylib
206.00 ms   0.1%    206.00 ms         _szone_free  libsystem_malloc.dylib
111.00 ms   0.1%    111.00 ms         _nanov2_free  libsystem_malloc.dylib
 64.00 ms   0.0%     64.00 ms         DYLD-STUB$$malloc  a.out
 48.00 ms   0.0%     48.00 ms         DYLD-STUB$$free  a.out
 44.00 ms   0.0%     44.00 ms         nanov2_malloc  libsystem_malloc.dylib
 20.00 ms   0.0%     20.00 ms         tiny_malloc_should_clear  libsystem_malloc.dylib
  9.00 ms   0.0%      9.00 ms         free  libsystem_malloc.dylib
  3.00 ms   0.0%      3.00 ms         _platform_memset  libsystem_platform.dylib
  1.00 ms   0.0%      1.00 ms         szone_try_free_default  libsystem_malloc.dylib
  1.00 ms   0.0%      1.00 ms         nanov2_try_free_default  libsystem_malloc.dylib
  1.00 ms   0.0%      1.00 ms         szone_malloc  libsystem_malloc.dylib
  1.00 ms   0.0%      1.00 ms         malloc  libsystem_malloc.dylib
```

```
Heaviest Stack Trace
by Weight
2.94 min  a.out (36973)
2.94 min  Main Thread 0x83e250
2.94 min  main  a.out
1.29 min  multiply_mm_naive...  a.out
 48.76 s  dot_product(double c...  a.out
```

Here, we go a little out of order and choose to do step 5 early! This is because we wanted to identify our biggest hotspots immediately so that we could tackle those problems right off the bat.

We can see a major hotspot is dotproduct() which is using about 25% of our runtime. Additionally, we see that free() is taking about 3% of our runtime. Due to these findings, we decided to integrate the logic of dot product dynamically into each function rather than using the dotproduct() function call itself in each loop!

This has optimizations in two ways. First, we are inlining the function, which reduces overhead. Second, in the initial implementation, in order to pass a row or column to the dot product function, we were copying it into a contiguous chunk of memory, and then passing it to dot product, and afterwards, freeing that memory. By implementing the dot product logic locally, we are relieving all of these unnecessary operations and copies! The benefits of this are explored in the following section.

4) Inlining

Inlining the Dot Product Logic and Removing Code to Copy Data to Contiguous Memory

Inlining the logic of the dot product and using the matrices as they are stored without copying each row and column into contiguous memory.

Dimension = 4 (500 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	215	83
Function 2	293	50
Function 3	486	91
Function 4	489	78

Dimension = 10 (500 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	584	181
Function 2	548	179
Function 3	5410	839
Function 4	5480	614

Dimension = 100 (100 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	62,800	65,100
Function 2	59,700	29,000
Function 3	5,880,000	626,000
Function 4	6,130,000	906,000

These results demonstrate the immense benefits from not copying the rows and columns into contiguous memory before taking the dot product and rather inlining the dot product logic. This allowed for the program to run much faster with far less overhead from function calls and unnecessary copying.

<u>Non-Inline 2D Indexing Function</u>

We also experimented with creating a function to handle the simulated 2D indexing rather than using the inline implementation which we had implemented in the baseline functions. This function handled the (i * colsA + j) type of operations! Below are the performance metrics with the non-inline indexing function.

Dimension = 4 (500 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	761	1120
Function 2	685	599
Function 3	2,900	558
Function 4	2,940	833

Dimension = 10 (500 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	2,390	1,721
Function 2	2,160	642
Function 3	23,500	9,000
Function 4	23,900	13,400

Dimension = 100 (100 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	124,000	44,900
Function 2	122,000	28,900
Function 3	12,400,000	1,100,000
Function 4	12,500,000	1,030,000

The above results demonstrate the performance of the non-inlined indexing function. Clearly, we can see that especially for larger matrices, the benefits of inlining the indexing function are significant! Hence, we made the decision to inline the indexing function to optimize the performance of the programs.

Using the non-inline indexing function lead to more overhead, ultimately reducing our performance and leading us to stick with the inline implementation.

<u>**Using MSVC 02 Compiler and Demonstrating the Benefits of Cache Locality Access**</u>

Using Windows MSVC compiler with the O2 optimization, we were able to further optimize the performance of the program.

Dimension = 4 (500 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	69	47
Function 2	65	48
Function 3	129	142
Function 4	143	51

Dimension = 10 (500 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	124	99
Function 2	115	38
Function 3	987	141
Function 4	1140	241

Dimension = 100 (100 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	8,850	1,230
Function 2	9,040	661
Function 3	1,110,000	132,000
Function 4	1,480,000	191,000

Dimension = 1000 (10 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	5,730,000	1,260,000
Function 2	6,400,000	812,000
Function 3	1,770,000,000	147,000,000
Function 4	1,360,000,000	108,000,000

Clearly from the figures above we notice the improvements that are made by using a speed-focussed compiler! The runtime was reduced significantly and we actually were able to see some of the main benefits that we have been looking for! For instance, with the very large dimensions (1000x1000) matrix multiplication, we can see the benefits of using contiguous memory for caching.

<u>2) Cache Locality Analysis</u>

In the matrix vector multiplications, we see the benefits of having contiguous rows! This is because when we take each dot product of the vector with each row of the matrix, we can smoothly iterate through one row as a contiguous entry in memory. When we perform the first multiplication of the dot product, the computer prepares the adjacent elements in memory to be used leading to cache hits when the elements are stored contiguously in rows (since the next element to be multiplied in the matrix is adjacent to the current element). Hence, in this row major implementation of the matrix vector multiplication (function 1), we see an improved runtime over the alternative column major implementation (function 2). This improvement is over a 10% boost in runtime!

Similarly, we see the same benefits when we use the column major implementation on the right in the matrix - matrix multiplication (note that this is function 4 since this is the same as storing B as its transpose). In both functions 3 and 4, we get the previously discussed benefits of having the left matrix stored in row major form. However, in function 4, we get further caching benefits from using contiguous memory for sequentially accessed elements since we store the right columns in contiguous memory. Thus, when we take the dot product between each right column and each left row, iterating over each as we do so, we are iterating over contiguously/adjacently stored elements both on the left and the right in function 4! Hence, even though these accomplish the same function, we are more likely to have caching benefits in implementation 4 over implementation 3. This improvement is reflected in our results, as we witnessed a roughly 25% improvement in runtime from implementation 3 to implementation 4.

To further show these benefits, we used dimension 2000 for a test!

Dimension = 2000 (1 trial)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	5,500,000	N/A
Function 2	8,760,000	N/A
Function 3	13,900,000,000	N/A
Function 4	11,800,000,000	N/A

With especially large matrices, our chances of having loaded the non-adjacent elements are far lower since they are so much farther apart in memory! Hence, we can very clearly see the benefits of implementation 1 and 4 over implementations 2 and 3, respectively. Again, it is important to remember that these functions accomplish the same multiplications, and only differ in how the matrices are stored. We clearly see the benefits of using row major on the left and column major on the right for matrix multiplication.

We can design even more extreme test cases to demonstrate this. For instance, consider matrix A with dimensions 10000x2 and matrix B with dimensions 2x10000. Using column major for B, the sequentially accessed elements are adjacent. Otherwise, they are about 10,000 elements apart. Here is the runtime for these cases:

Dimension = (10,000x2) x (2x10,000) (1 trial)

	Mean (nanoseconds)	SD (nanoseconds)
Function 3	1,610,000,000	N/A
Function 4	426,000,000	N/A

The runtime benefits are obvious. Cache Locality Access makes a difference, and this is manifested through the informed choice of using column major format for matrix B, cutting our runtime by 75% in this extreme case.

3) Memory Alignment

Using MSVC 02 Compiler and Intentional Memory Alignment

Dimension = 4 (500 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	74	47
Function 2	67	49
Function 3	113	49
Function 4	130	51

Dimension = 10 (500 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	88	33
Function 2	87	36
Function 3	569	49
Function 4	762	613

Dimension = 100 (100 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	11,100	4,610
Function 2	12,900	6,590
Function 3	1,650,000	352,000
Function 4	1,370,000	282,000

Dimension = 1000 (10 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	1,360,000	140,000
Function 2	1,920,000	337,000
Function 3	1,480,000,000	31,800,000
Function 4	1,340,000,000	25,600,000

Using Agner’s optimization manual, in order to word align a vector of doubles(which is usually 64 bits) if we use the ‘new’ operator it automatically handles the memory alignment. We see some modest benefits to the runtime, especially for the 1000x1000 and the 10x10 sizes. Hence, the memory alignment did provide a noticeable improvement, however it did so inconsistently, handling the largest case very well, and also handling the second smallest case very well. From the results that we are seeing, it is difficult to generalize conclusions as to its practical applications. However, it seems to be an overall improvement to the program.

6) Optimization Strategies

Over this document, we have detailed the process that we took as a team to optimize the implementation of these matrix algebra algorithms. The final, optimized implementation of our code is in the optimized folder of the github repository!

The performance of the final optimized implementation is shown below along with a brief description of the changes that were implemented from the baseline.

- Inlining the Dot Product Logic and Removing Code to Copy Data to Contiguous Memory
- Using MSVC 02 Compiler
- Using the ‘new’ operator to implement intentional memory alignment in the matrices

Final Performance:

Dimension = 4 (500 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	74	47
Function 2	67	49
Function 3	113	49
Function 4	130	51

Dimension = 10 (500 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	88	33
Function 2	87	36
Function 3	569	49
Function 4	762	613

Dimension = 100 (100 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	11,100	4,610
Function 2	12,900	6,590
Function 3	1,650,000	352,000
Function 4	1,370,000	282,000

Dimension = 1000 (10 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	1,360,000	140,000
Function 2	1,920,000	337,000
Function 3	1,480,000,000	31,800,000
Function 4	1,340,000,000	25,600,000

Baseline Performance:

Dimension = 4 (500 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	706	707
Function 2	654	697
Function 3	3,070	3,070
Function 4	2,760	1,330

Dimension = 10 (500 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	2,430	3,120
Function 2	1,980	1,530
Function 3	19,100	89,70
Function 4	19,300	10,300

Dimension = 100 (100 trials)

	Mean (nanoseconds)	SD (nanoseconds)
Function 1	92,200	34,800
Function 2	88,900	21,400
Function 3	9,040,000	4,360,000
Function 4	8,620,000	1,450,000