CITS3402 Project 1 Report

Henry Hollingworth 21471423 23rd/09/2019

Contents

Operating System

This submission was developed on Windows 10 (version 10.0.18362) using the Microsoft Visual Studio 2019. The C code was compiled with the MSVC toolkit included with Visual Studio. The *Test* sub-project has a dependency on the check (libcheck, 2019) unit testing framework for C. The project was compiled as C code

with the /openmp and /bigobj (for the test project) flags enabled. An installation of Python 3.7 with the parmap (zeehio, 2019) was used to run a test fixture generating script and performance test script.

The project can be compiled using Visual Studio 2019 without check installed by *unloading* the test project and then compiling just the main executable.

Sparse Representations

Three sparse matrix representation formats were implemented in the project. These formats are Coordinate Format (COO), Compressed Sparse Row Format (CSR) and Compressed Sparse Column Format (CSC). The CSR and CSC implementations were derived from a blog post on sparse matrix formats by GormAnalysis (GormAnalysis, 2019). The three formats were exposed via a common interface of typedef-ed function pointers as defined in *matrix.h*. In this way it is trivially easy to change the sparse representation used for matrices in a calculation.

The CSR and CSC formats are optimized for fast row and column retrieval respectively because they store the non-zero values for a row or column contiguously in memory. The Big-O complexity for the retrieval of a column or row is proportional to the number of non-zero values in the column or row for CSR and CSC respectively.

COO format has significantly worse column and row retrieval complexity because it cannot be determined in *O(1)* time *where* the first value of a particular row or column would appear in the (row, col, value) triple list. Therefore, column or row retrieval requires a worst-case *O(width*height)* scanning of the triples list.

The various implemented matrix operations use the different sparse formats in order to improve the theoretical performance as detailed in the following table:

Table 1: Operation Sparse Matrix Representations

Operation	Left Representation	Right Representation	Reasoning
sm	COO	N/A	The scalar multiplication of the whole matrix can be subdivided into tasks which multiply each *element* of the matrix. These tasks can be completed independently as they are a function of (value, row, column) only. Therefore, the storage mechanism which allows for the fastest visitation of such triples is optimum. COO format matches this description and hence was chosen.

Operation	Left Representation	Right Representation	Reasoning
			Previously *sm* was implemented using a CSC representation. Upon performance testing this was changed, see *below* for more details.
tr	COO	N/A	A trace involves the retrieval of a single value from either all rows or all columns. The two parallelization options: a) Have many workers take a column/row each (storing the matrix in CSC/CSR form) and find a single value each, and these values are reduced. b) Scan through all matrix values sequentially (stored contiguously in memory) with many workers. Implementation wise this is similar to parallelizing a list summation – except that we only sum values which are on a diagonal. Option b) was chosen due to the intuition that the cost of constructing the rows/columns was to high to simply retrieve a single value.
ad	CSC	CSC	Matrix addition requires corresponding values from the two matrices to be added together. To do this efficiently we want a way to quickly traverse the same region(s) of both matrices at the same time and sum the element pairs in the region. This lends itself to using either the CSR or CSC formats, which can efficiently return column or row region(s) upon which an elementwise addition can be parallelized. CSC was chosen over CSR arbitrarily.

Operation	Left Representation	Right Representation	Reasoning
ts	COO	N/A	Similar to ***sm*** the transposition of every matrix value can be done independently since the only information required is the value and its row and column position. Since fast traversal of this information is important COO chosen. **Previously *ts* supported either CSR or CSC formats and would be told how to retrieve either a row or column from a matrix. Again, after performance testing this operator was re-written to use the COO format. See *below* for more details.**
mm	CSR	CSC	Multiplication of matrices involves the sum of elementwise multiplication of a row from the left matrix, and a column from the right matrix. It is therefore logical to store the left matrix in CSR format and the right in CSC format to optimize those access patterns.

Parallelization Intuition

Generic Parallelization Strategy

The main technique used in the project to parallelize operations is to construct a series of tasks or reductions (Jaka, 2016) which operate sequentially over a contiguous section(s) of memory to minimize cache invalidation. For matrix operations the result value of each task corresponds to a cell in the result matrix. A shared memory buffer is used to allow the workers completing the tasks to write their result values without collisions or race-conditions.

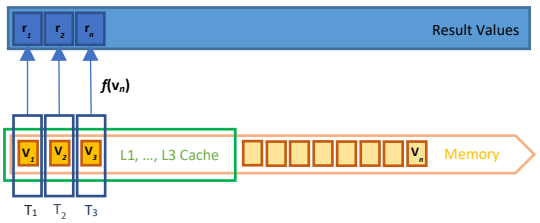


Figure 1: Generic Parallelization Strategy

Scalar Multiplication

First Approach

In the initial implementation, scalar multiplication was parallelized over the elementwise multiplication of a column by

a scalar. The outer non-parallelized loop went over each column, the inner parallelized loop would take elements from the column, multiply them by the scalar and update the result value buffer.

The intuition behind this approach is that if you were to instead parallelize over the *columns* (not the *elements of a column*), as the workers were multiplying the elements of their column, the column stored in the cache would keep changing. Imagine two workers w_1 and w_2 which were operating over the column matrices a and b respectively. As w_1 was going to process $a_{1,1}$ it would pull a into the CPU cache, but then as w_2 goes to process $b_{1,1}$, b would be brought into the cache potentially invalidating the copy of a that w_1 needs. The validity of this reasoning depends on the properties of the CPU caches in a processor. Important points to consider are; what are the sizes of the cache(s), what is the latency of the cache(s) and which if any caches are shared between cores.

Second Approach

However, after performance testing and further reflection I realised that the parallelizable tasks could be instead the transposition of a single value because to transpose a value correctly you only need the value itself and its column and row position. This approach (when used with a COO representation) also has better cache invalidation properties because **all** task parameters (the COO triples) are stored contiguously in memory without requiring any retrieval logic (unlike CSC which must do work to re-construct a column).

Trace

The parallelization strategy for the trace operation is less intuitively clear than the other operations. As described in Table 1 - if the columns/rows are traversed and a single element retrieved from each, significant time will be spent constructing the column/rows, for a singular lookup operation. The intuition is that it is probably better to scan through the contiguous list of non-zero elements in memory (low cache invalidation) and only reduce the values which sit on the diagonal.

Transposition

First Approach

Transposition was parallelized by traversing the rows/columns of a matrix single-threadedly and then having worker(s) operate over the same row/column to place values into a result matrix at the correct transposed position.

While conceptually this implementation is very close to the definition of transposition, namely *"the rows of a matrix become columns or vice versa"* by using the CSC/CSR sparse representation formats and operating at the row/column level, many cycles will be spent constructing these rows/columns. This is wasteful because the *order* of cell traversal is unimportant because the only information required to transpose a cell is its *column* and *row* – and these properties are included in the COO triples.

Second Approach

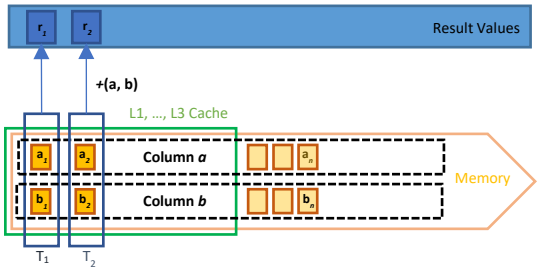
After performance testing and realising the operator could be faster by operating over COO represented matrix the operator was re-written. The new

implementation simply traversed the COO triples and used their column and row properties to calculate the correct transposed position.

Addition

Addition of matrices conceptually operates over pairs of either rows or column sourced from two matrices. Therefore, given the *Generic Parallelization Strategy* outlined previously the parallelization strategy is two store two columns (one from each matrix) at a time, and have workers sum elementwise the corresponding values of the two columns.

Figure 2: Implemented Addition Parallelization



A key reason why we simply did not store the two matrices in COO format and traverse the triples as pairs is because the first triple from some matrix *a* **does not necessarily describe the same cell location** as the first triple of a matrix *b.* Therefore, the simple contiguous scan mechanism suggested as an improvement in the *Transposition* section is not appropriate.

Matrix Multiplication

The parallelization of this this operation did not follow the *Generic Parallelization Strategy*, rather than having tasks correspond to an elementwise operation, each task was the multiplication of a row vector from the left matrix with a column vector.

The right matrices' columns are traversed in order, and for each column the multiplication of that column with a singular row the left matrix is distributed as a task. Before moving to the next right matrix column, the multiplication of all the left matrices rows with the current right matrices' column is completed.

The intuition here is that with the correct *scheduling* of the inner parallel for loop the cache invalidation can be acceptably reduced to such a level that due to each task containing substantial work, a performance benefit can be realised.

Correctness Testing

The provided reference input and output files were processed by a Python script to generate C unit test code. By traversing and parsing all the example output log files a set of operation, input file(s) and expected value(s) tuples could be constructed. From these tuples C code testing the implementation could be generated. Due to the size of the matrices, for operations involving non-scalar results only 500 of the non-zero values of the output matrix were asserted upon.

The generated 'spec' tests were combined with other handwritten tests in the *CITS3402P1-Tests* project. All tests passed and a log file of a test run is included in the root directory of the source code.

Having these tests allowed for more rapid bug-fixing and refactoring due to being to check for any introduced regressions. They also provided a useful debug entry

point when some component of functionality was not working correctly.

Performance & Scalability

Results

The performance of all operators with 1, 2 and 4 threads were tested using the 64, 128, 256 and 1024 square integer matrices. A Python script was written to invoke the program for 100 trials and the trial results were exported to a CSV file. This CSV file was then processed by another Python script to produce the histogram plots attached in the Appendix.

Only in the case of ***mm*** did an increasing thread count result in a performance improvement for all input matrix sizes. The ***ts***, ***ad*** and ***sm*** operations all received a performance penalty with multithreading for the 64, 128 and 256 sized square matrices. However, for the 1024 square matrix input the ***ts***, ***ad*** and ***sm*** showed performance improvement as the thread count increased.

Table 2: Threading Performance Summary
Summarizes if an increasing thread count improved performance (+) or if it resulted in a penalty (-).

Operation	Matrix Size			
	64	**128**	**256**	**1024**
mm	+	+	+	+
sm	-	-	-	+
ad	-	-	-	+
ts	-	-	-	+
tr	-	-	-	-

Overview

The runtime performance of the implementation of the matrix operators is comparable to the performance of the reference solution whose runtimes are found in the example output files.

Unfortunately, the memory performance of the implementation is quite poor despite using sparse matrix representations. The reason for this is the constructors for the various sparse matrix representations accepted a dense array of *union matrix_value* elements from which the sparse representation would be built. Therefore, when loading the input files into memory or calculating a matrix type result there is at least one allocation of a *union matrix_value values[width * height]* array. This could be improved in at least two ways:

a) Provide a lazy *iterable* of *union matrix_*value to the constructors instead of an array. This would require some complicated C code to provide a generic implementation of iterables – outside the scope of this assignment.
b) Remove the collection of *union matrix_value*s as a parameter to the constructors and instead pass a filename. The file reading logic would then be repeated in each function with the necessary sparse matrix construction logic interleaved.

Neither of these options were pursued due to both time constraints and negative effect such fixes could have on the quality of the code.

Conclusion

This was a difficult assignment that required a combination of C programming skills, understanding of how the CPU operates and OpenMP. Overall, I am quite happy with the quality of the code I produced however I wish I had more time to write an iterable library to improve the memory performance (see *on page 7*). I think the inclusion of unit tests and performance measurement graphing scripts are useful additions that make this submission stand out.

Upon fully testing the program and seeing that the ***mm*** operation benefited from parallelization at all input sizes I wonder if my assumptions about cache invalidation were incorrect given a modern CPU and the size of data involved. If so, the performance of my ***ad*** operator could be improved by changing from elementwise parallelization that my *Generic Parallelization Strategy* employed to instead parallelize over *rows* or *columns* as I did in the ***mm*** operator.

I was pleased to note a drastic single and multithreaded performance improvement in the ***sm*** and ***ts*** operators after re-writing the operators to work against a COO matrix representation. This performance difference can be see by comparing the First Approach and Second Approach versions of the respective operations' histograms attached in the *Appendix*.

References

galgalesh. (2019, 01 08). *Is there a more basic tutorial for the c unit testing framework check*. Retrieved from StackOverflow: https://stackoverflow.com/questions/14176180/is-there-a-more-basic-tutorial-for-the-c-unit-testing-framework-check

GormAnalysis. (2019, 04 12). *Sparse Matrix Storage Formats*. Retrieved from GormAnalysis: https://www.gormanalysis.com/blog/sparse-matrix-storage-formats/

Jaka. (2016, 06 6). *OMP For Reduction*. Retrieved from jakascorner: http://jakascorner.com/blog/2016/06/omp-for-reduction.html

libcheck. (2019, 09 23). *Home*. Retrieved from Check: https://libcheck.github.io/check/

Microsoft. (2019, 07 30). *OpenMP Library Reference*. Retrieved from microsoft: https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-library-reference?view=vs-2019

zeehio. (2019, 09 23). *parmap*. Retrieved from pypi: https://pypi.org/project/parmap/

Appendix

Figure 3: Addition Operation Threading Performance

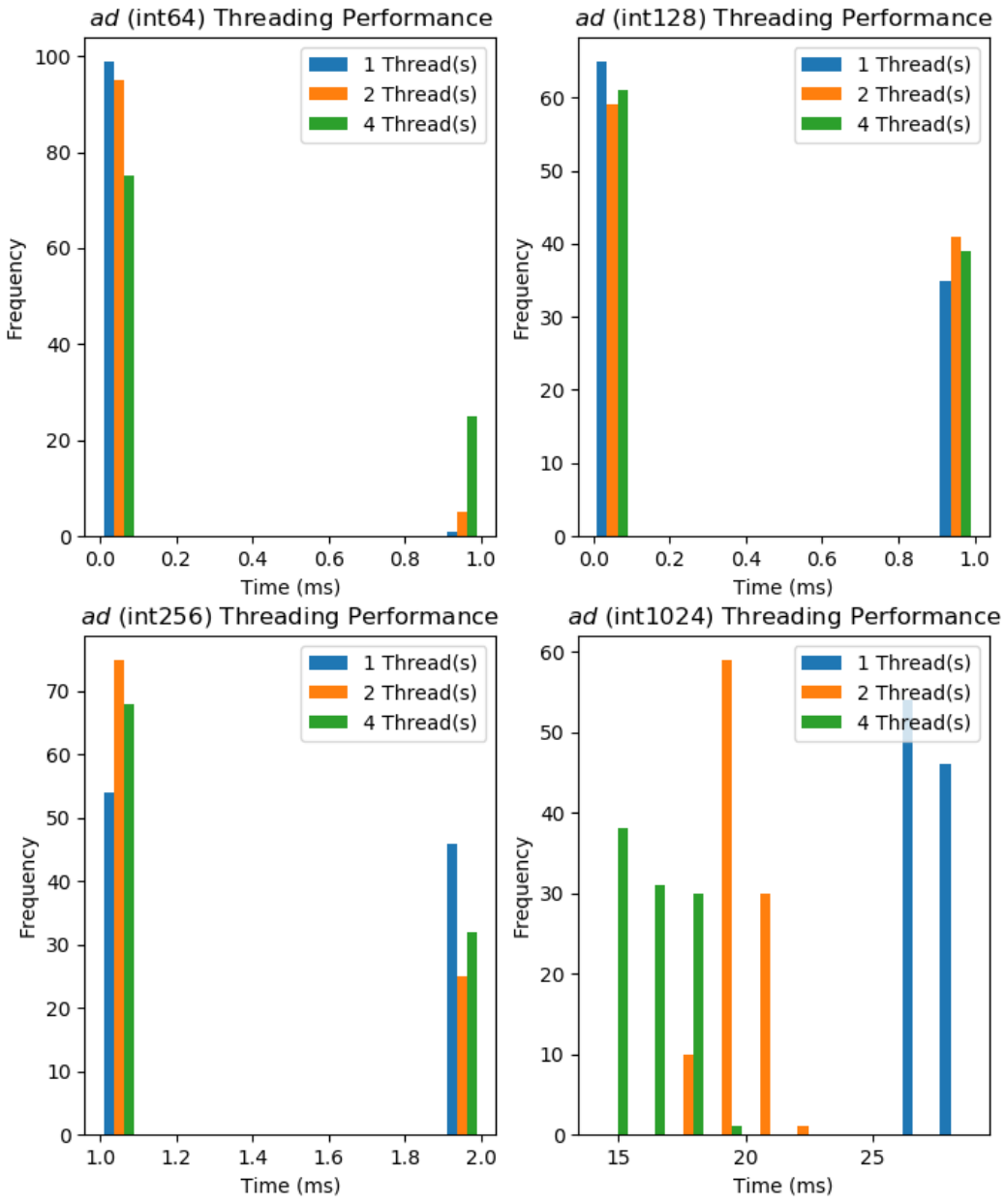


Figure 4: Matrix Multiplication Operation Threading Performance

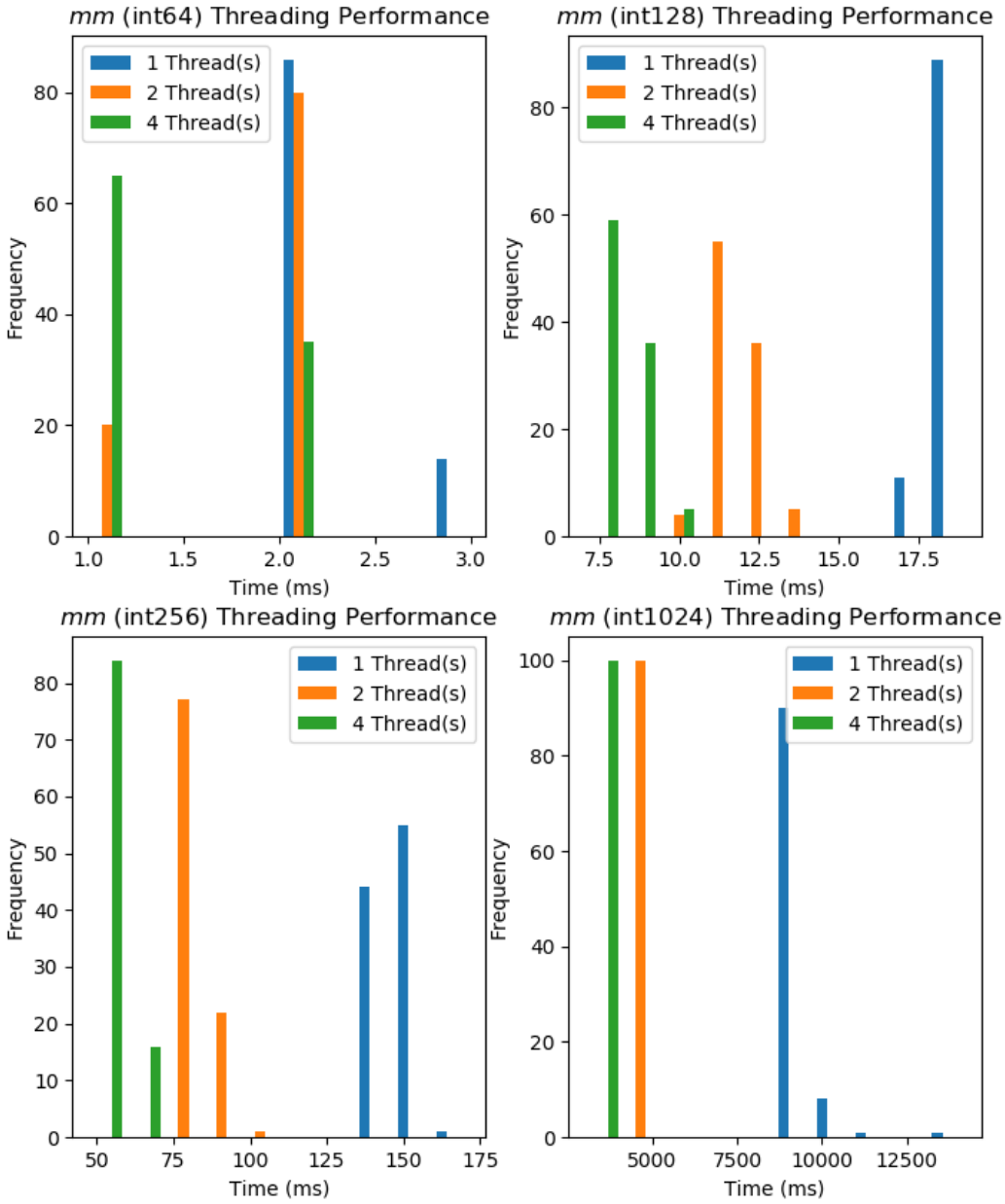


Figure 5: Scalar Multiplication Operation (First Approach) Threading Performance

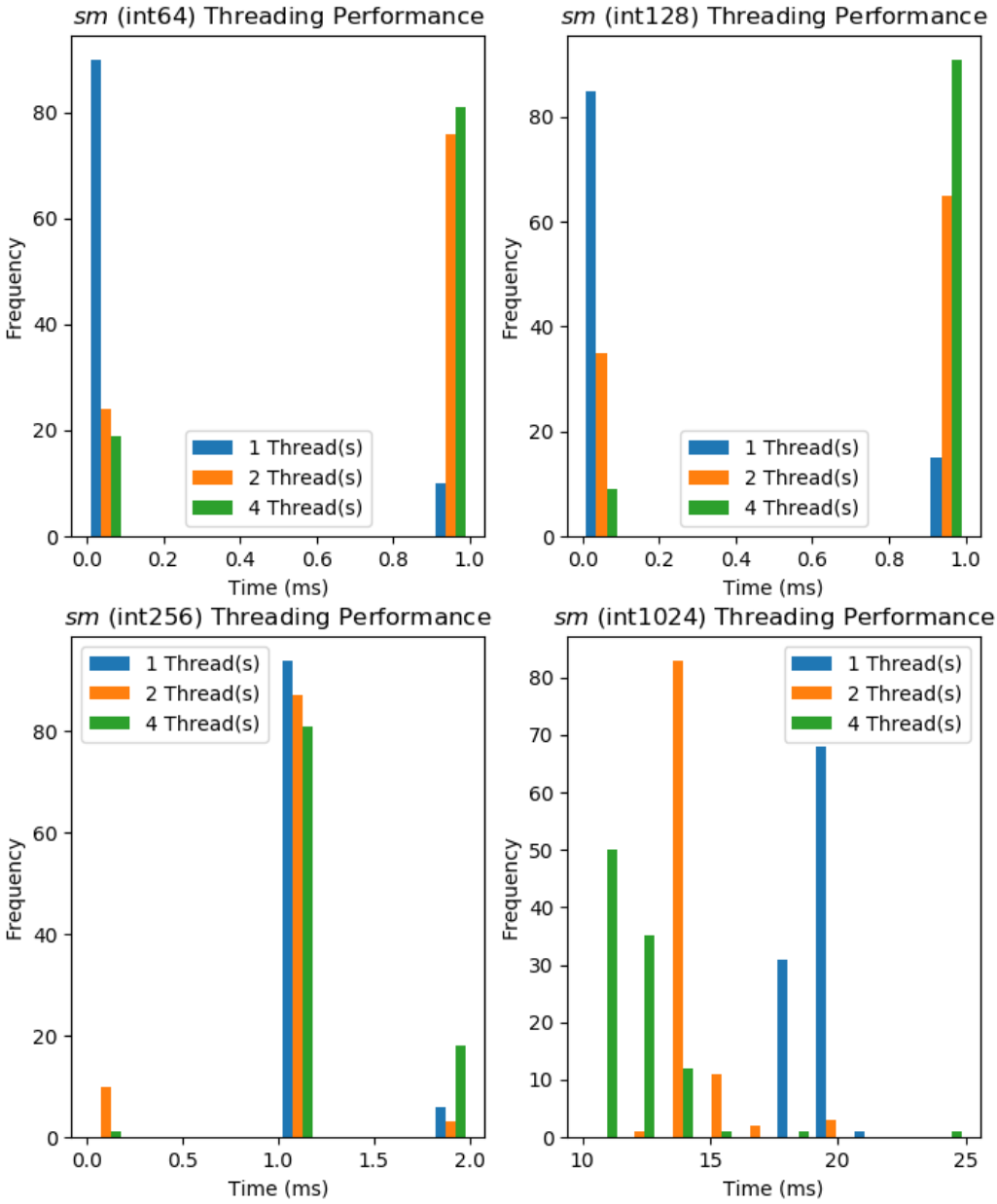


Figure 6: Trace Operation Threading Performance

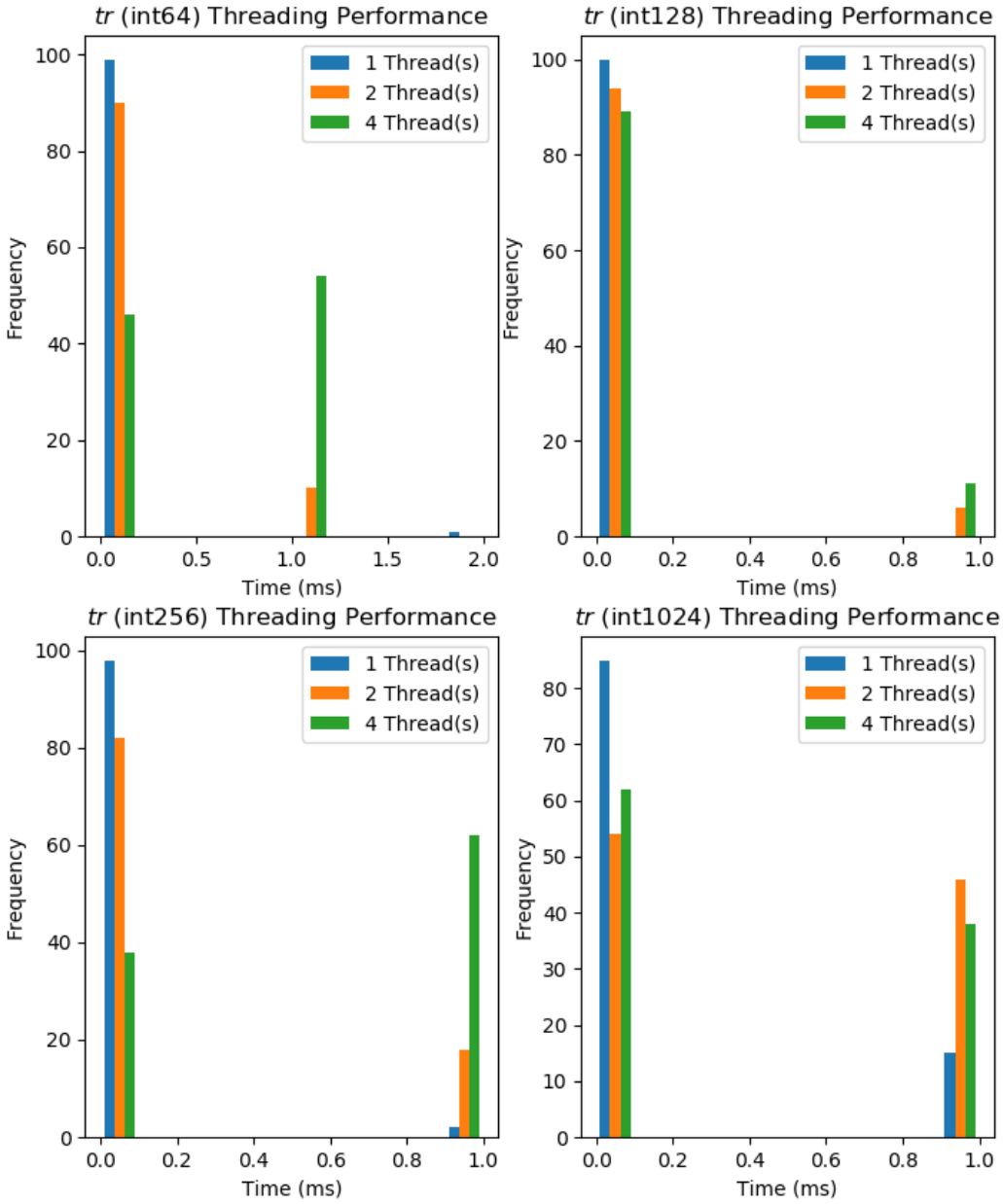


Figure 7: Transpose Operation (First Approach) Threading Performance

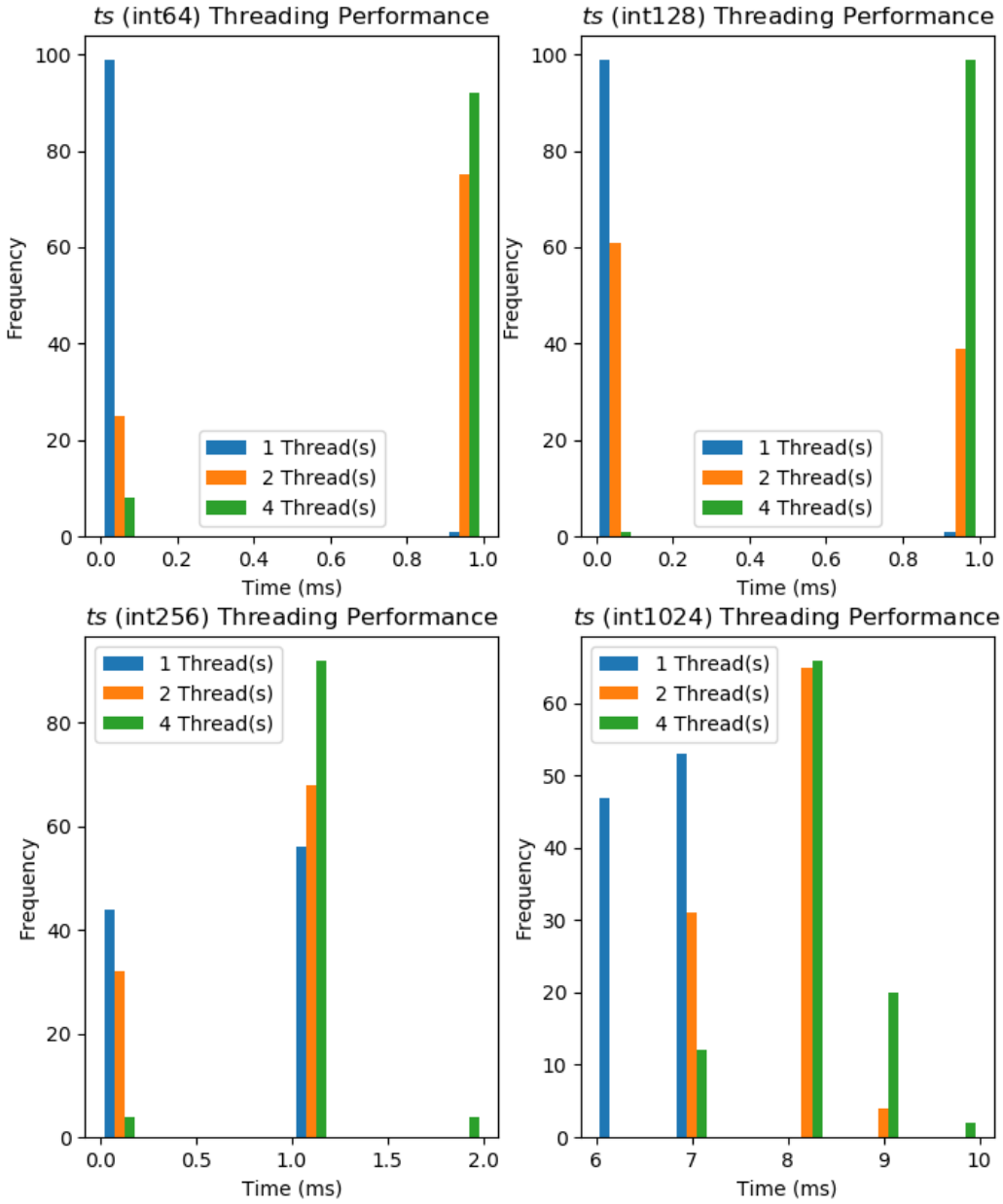


Figure 8: Scalar Multiplication Operation (Second Approach) Threading Performance

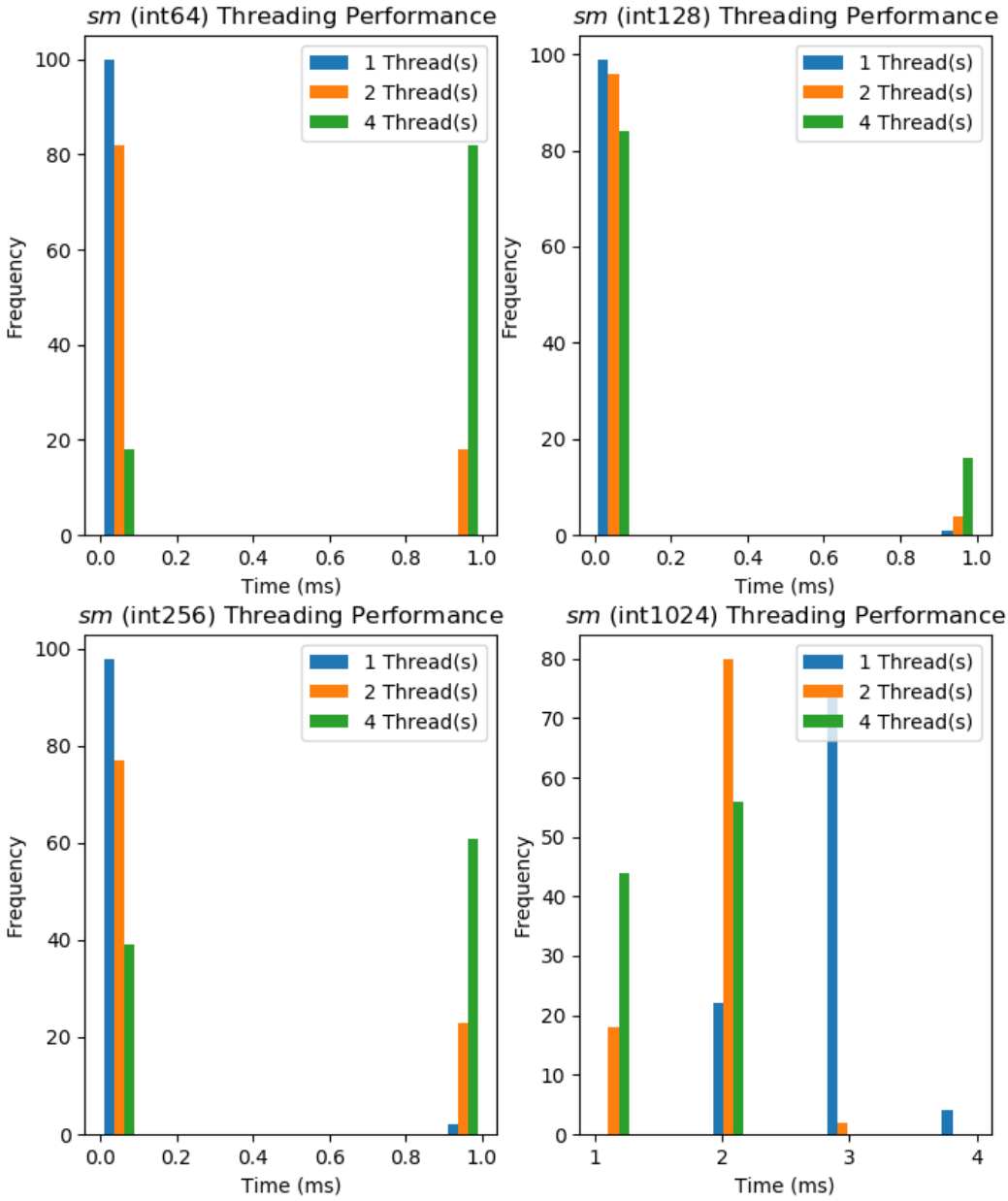


Figure 9: Transpose Operation (Second Approach) Threading Performance