CITS3402 Project 1:Sparse Matrices Report

Séamus Mulholland-Patterson

September 2019

Contents

1 Introduction

Matrix operations are incredibly important in many areas of computer science and software engineering. Examples such as machine learning, graphics, computer vision benefit immensely from fast matrix computations. This project examines the case of sparse matrices, where matrices contain a large number of zeroes. This is particularly useful for computation in neural networks as many weights in the graph are often zeroed. Although worst time complexity cannot be altered in many of these cases, it will be shown that with threading, and efficient storage of the matrices, a tangible improvement in computation time can be observed.

2 Sparse Formats

2.1 Coordinate Format (COO)

Coordinate format is the simplest form of sparse format. It consists of a list of elements such that each element contains 3 values: The row of the element, the column of the element, and the value of the element. Constructing this is a matter of iterating over the rows and columns of a dense matrix, checking for a non-zero value. If such a value is found, construct an element consisting of its row, column and value and add it to the list of elements. Time complexity for this is $O(r \cdot c)$ where r is number of rows and c is number of columns

2.2 Compressed Sparse Row (CSR)

Compressed sparse row consists of 3 arrays. A list of non-zero elements (nnz), a list of the columns corresponding to the non-zero elements (ja), and finally, an array of the cumulative sum of nonzero elements for each row with element 1 starting at 0 (ia). This format allows for parallelisation across rows as the non-zero values and corresponding column values in a row may be retrieved by indexing the array of non-zero elements from ia[row_index] to ia[row_index + 1]. Construction can be achieved by iterating across all rows first setting ia[row_index + 1] to be the same as ia[row_index] then iterating across all columns in the row to search for non-zero elements. If a non-zero element is found, the value of ia[row_index + 1] is incremented, the non-zero value is appended to nnz and the current column is appended to ja. Time complexity for this is $O(r \cdot c)$ where r is number of rows and c is number of columns.

2.3 Compressed Sparse Column (CSC)

Compressed sparse column is very similar to compressed sparse row. The format consists of the same 3 arrays of nnz, ia, and ja, however ia in this case now contains a cumulative sum of nonzero elements for each column rather than row. The list ja is similarly switched to now comprise of the row corresponding to each nonzero element. Construction is similar whereby instead of iterating over rows then columns, instead iteration occurs over columns then rows. Similarly ia is set every iteration of a column. Time complexity for this is $O(r \cdot c)$ where r is number of rows and c is number of columns.

3 Matrix Operations

3.1 Single Matrix Operations

For any given matrix M of dimensions r rows and c columns the following operations are required.

3.1.1 Trace

For non-square matrices the result is undefined. Otherwise:

$$\text{Trace} = \sum_{n=1}^{r} M_{n,n} \tag{1}$$

A sparse implementation of this iterates over the sparse elements, possibly in a clever way, checking for cases where row and column are the same.

3.1.2 Transpose

Given resultant matrix Y

$$Y_{j,i} = M_{i,j} \forall 1 \leq i \leq r, 1 \leq j \leq c \tag{2}$$

A sparse implementation either builds a new structure representing rows and columns, with rows and columns switched, or simply swaps existing values for rows and columns for every non-zero element

3.1.3 Scalar Multiplication

Given resultant matrix Y and scalar s

$$Y_{i,j} = s \cdot M_{i,j} \forall 1 \leq i \leq r, 1 \leq j \leq c \tag{3}$$

A sparse implementation of this simply iterates over the non-zero elements, applying the scalar to each element.

3.2 Double Matrix Operations

For any given matrices N with dimensions r_n rows and c_n columns and M with dimensions r_m and c_n columns

3.2.1 Matrix Addition

For M and N where either $r_m \neq r_n$ or $c_m \neq c_m$ the result is undefined. Otherwise for a given resultant matrix Y

$$Y_{i,j} = M_{i,j} + N_{i,j} \forall 1 \leq i \leq r_m, 1 \leq j \leq c_m \tag{4}$$

A sparse implementation of this would be iterating over the non-zero elements in each array such that non-zero elements with corresponding i and j values are summed and added to the resulting sparse representation, and elements without corresponding i and j values are added to the resulting sparse representation.

3.2.2 Matrix Multiplication

For MN where $c_m \neq r_n$ the result is undefined. Otherwise for a given resultant matrix Y

$$Y_{i,j} = \sum_{n=1}^{c_m} M_{i,n} \cdot N_{n,j} \forall 1 \leq i \leq r_m, 1 \leq j \leq c_n \tag{5}$$

A sparse implementation of this would be iterating over rows of M then columns of N before using a clever way of iterating over the non-zero values in the current row of M and current column of N. The column of the element from M would be compared with the row of N and if a match was found then resultant multiplication of the non-zero elements would be added to the current value of Y indexed by the row of M and column of N. Any comparisons that don't match are ignored as they are zeroed.

4 Implementation

4.1 Operating systems

The program has been written such that it is portable between MacOS, Windows and Linux for distributions of linux using the glibc library.

4.1.1 Windows

A windows executable is included, however if you wish to compile it the command is stored in windows_make_cmd.txt. For compilation navigate to the directory of the project in a visual studio developer command prompt and enter the command.
Usage: `sparse_matrix.exe <operation> [%f|%d] -f <file1> [file2]`

4.1.2 MacOS

This project was tested on Darwin, however it complies with the C99 standard such that any version of gcc installed with OpenMP support will be able to compile the executable. For compilation simply `make`. If an error with OpenMP is thrown, install a version of gcc that supports OpenMP and change the `GCC_CMD`

variable to be that of the alias for the version of gcc supporting OpenMP. There is an option to `make install` to install unit tests that check exit values and example input files to run those unit tests.
Usage: `./sparse_matrix.bin <operation> [%f|%d] -f <file1> [file2]`

4.1.3 Linux

This project was tested on Ubuntu, however it complies with the C99 standard such that any version of gcc installed with OpenMP support will be able to compile the executable. The only environment setup was defining the macro `_DEFAULT_SOURCE` to enable the timezone variable in time.h. For compilation simply `make`. Similarly to MacOS if an error with OpenMP is thrown, install a version of gcc that supports OpenMP and change the `GCC_CMD` variable to be that of the alias for the version of gcc supporting OpenMP. The same option exists to `make install` for unit tests and the corresponding input files.

4.2 Flags

Order of flags is unimportant
Operations (non-optional):
  `--tr` Trace
  `--ts` Transpose
  `--sm [%d|%f]` Scalar Multiplication'
  `--ad` Addition
  `--mm` Matrix Multiplication
Input files (non-optional):
  `-f [file1] [file2]`
Number of threads (optional):
  `-t %d`
Log Output (optional)
  `-l`
No threading - ignores number of threads specified (optional)
  `--nothreading`
Format (optional):
  `--format [COO|CSR|CSC]` specify COO, CSR, or CSC format
Print final output structure to stdout (optional)
  `-p`
Print timing information to stdout (optional)
  `--timing`

4.3 Structures

4.3.1 COO

COO is stored as a struct containing several elements. The rows and columns of the dense matrix are stored as these are impossible to determine simply from the array of elements. The edge case of completely zeroed rows or columns at the extremities of the matrix is what makes this an impossibility. pointer to a 1-dimensional array of a structure containing i, j, the type of the element, and a union storing either a long double or an integer depending on type. Finally the COO struct also contains the type of the matrix as this allows the 0 element case to still be typed.

4.3.2 CSR

CSR is stored as a struct containing several elements. Ia is stored as a pointer to an integer array with length rows + 1. Ja is stored as a pointer to an integer array with length ia[rows + 1]. nnz is stored as a union storing either a long double pointer or an integer pointer either of which will have length ia[rows + 1]. Num_vals stores ia[rows + 1] ie the number of non-zero values, for readability. The rows and columns of the dense matrix are stored as columns are impossible to determine from just the ja, and rows provides a length for ia. Finally the struct contains the type of the matrix such that the right pointer in the union is accessed.

4.3.3 CSC

CSC is a typedef of CSR as the structures are identical and it allows for easy implementations concerning treating one type of matrix as the other.

4.3.4 MAT_RV

MAT_RV is a struct containing the output information required for logging, and most importantly the resultant dense matrix of the sparse representation. It contains error information, timing, whether to treat the resultant matrix as a matrix or a single value, the rows and columns of the resultant, and a union containing the resultant matrix. For the purposes of this project, this structure has been designed to be able to be returned from every operation function. Writing the functions such that they could perform multiple matrix operations, would simply require the operations to return a structure of the format they were operating with, and later convert that format to the dense format and populate MAT_RV

4.4 Trace

The COO implementation iterates across all elements in the array, searching for cases where i and j are equivalent to add them to the result. Complexity is $O(n)$ where n is the number of non-zero elements.

For CSR and CSC the implementation iterates across the elements in ja by indexing ja with ia for each row for CSR and column for CSC. If a value for ja $\geq$ row_index for CSC or col_index for CSR then the value of nnz is added to the result only for the equal case, and ja is then indexed by the next row. Although the complexity of this is still $O(n)$ it provides a two times speed up in the average case due to only needing to iterate over half the elements on average.

An ideal sparse representation for Trace would have the ability to index along diagonals, more specifically, the leading diagonal. Turning complexity into $O(r)$ where r is the number of rows and as in a square matrix this has a quadratic relationship to the worst case number of elements, a factor of r speed is observed.

4.5 Transpose

The COO implementation iterates across all elements in the array, swapping the i and j values. Time complexity is $O(n)$ where n is the number of non-zero elements.

CSR and CSC are implemented similarly, however are able to be indexed by rows or columns depending on either being CSR or CSC. This does not give a decrease in time complexity, however arranges the elements correctly in the representations. Time complexity is still $O(n)$ where n is the number of elements.

The default implementation does not require any processing time as it simply treats a CSR as a CSC, and by swapping rows and columns, the matrix is now transposed. Time complexity in this case is constant $O(1)$ and this is already an optimal solution.

4.6 Scalar multiplication

Implementation in COO involves iterating across all elements in the array, applying the scalar to the non-zero element in the array, changing the type if need be. Time complexity is $O(n)$ where n is the number of non-zero elements.

CSR and CSC again have a similar implementation, however due to the constraints of a union, a second structure must be created to store the result. Time complexity is again $O(n)$ where n is the number of non-zero elements.

An ideal implementation would not realise the scalar until after all operations where completed. Hence an extra time constant would be added to the construction of the dense format from the sparse, but the time complexity for the operation of adding a scalar to the representation would be constant $O(1)$.

4.7 Addition

Addition is implemented in COO by using a two pointer method, where each pointer is catching up to the other unless they are equivalent, in which case both pointers move. This technique results in a time complexity of $O(n)$ where $n = (set(i,j) \in N) \cup (set(i,j) \in M)$ for matrices N and M

CSR and CSC have similar implementations, except accessing each element by iterating over each row. This provides a constant number of loops which may be threaded later. Time complexity is still the same as COO

The only necessary computation is the intersection of the two matrices. Hence if there was a structure that was able to determine the intersection of two matrices efficiently, and append the other elements to the nnz there could be a small speedup. It is likely that this approach contains a lot of far higher time constants than that which it saves. $O(n)$ is likely the best that can be done

4.8 Matrix multiplication

Matrix multiplication with the COO format requires one matrix sorted in row column order and the other sorted in column row order. This is implemented with a quick sort for compatibility however for a lower worst case a merge_sort would be more applicable. This is not an issue however as the sort is not the dominating factor in complexity. The computation is implemented again with a two pointer approach, however, a row in matrix M, if the multiplication is MN, is required to be iterated over multiple times, hence the location of the pointer to the start of the current row of M is stored such that it may be reused without having to search again. Thus the time complexity for the multiplication MN would be $O(rows_M \cdot columns_N \cdot rows_N)$. Whether $rows_N$ or $columns_M$ are used, in complexity, is irrelevant as they are equivalent by the definition of matrix multiplication.

For CSC and CSR multiplication the representations must be transformed in such a way that for the multiplication MN M is a sparse matrix in CSR format and N is a sparse matrix in CSC format. Once this is the case, the ia arrays of both may be iterated over for the rows of M and columns of N then a two pointer approach may be used for the ja values of each. Time complexity is the same as COO, with opportunities to parallelise as rows and columns are determinable loops.

The default for this operation is to create a CSR representation for the initial matrix, and create a CSC representation for the second matrix.

5 Threading

5.1 Cache Coherence

In order to avoid issues with cache coherence, any variable required to be edited by multiple threads stores a local copy of the result of that variable on it's local stack, before going to a single thread section where the same operation is performed across all the local copies to give a result. Unfortunately this does not result in a factor of complexity being removed ie an $O(n^2)$ operation does not become $O(n)$ with n number of threads, instead it remains $O(n^2)$ the optimal complexity for this operation becomes $O(n \log n)$ with $\log n$ threads

5.2 Resource Sharing

In order to avoid issues where threads are blocking consistently waiting for access to a resource, resources accessed by multiple threads are copied onto the stack, whether that be by dereferencing an address to retrieve a value from the heap or copying the value from the shared stack.

6 OpenMP

6.1 Directives

6.1.1 parallel

The parallel directive builds threads to be used by the block under the parallel directive. Clauses may be used to specify exactly how the threads are constructed as well as how many, otherwise the compiler will attempt to optimise.

6.1.2 for

For is used specifically for for loops to split the loop into sections handled by each thread. Clauses may be used to specify how this splitting should occur, as well as what loops may be sectioned.

6.1.3 critical

Critical locks a block such that only one thread may execute that block at any one time. This allows for operations requiring updating a single value updated by multiple threads, to be shared correctly

6.2 Clauses

6.2.1 shared

Shared specifies that one instance of a variable be shared amongst all threads. This prevents cache coherency issues as there are no other instances of the variable that are required to be updated.

6.2.2 private

Private specifies that a local instance of a variable should be made available to the block, under which the clause is called, for each thread. This local copy is not expected to be the same amongst all threads, hence cache coherence is not an issue, more importantly this enables the same variable to take on multiple values amongst threads, enabling partitioning of operations.

6.2.3 reduction

Reduction specifies a special case of for loop where a single operation is occurring multiple times to a single variable. OpenMP provides the reduction clause for builtin compiler optimisation for this case.

6.2.4 schedule

for parallelising some for loops, operations in each iteration of the loop may be differing. Rather than split into same-size, contiguous blocks of iterations, as is default for the for directive, schedule provides an option to size your iteration blocks.

6.2.5 num_threads

num_threads forces a parallel section to run with a specified number of threads, irregardless of what is optimal

6.3 Use

6.3.1 Trace

The project uses a parallelised section with a sectioned for loop, into a critical section. This achieves the desired outcome, however a more applicable approach would have been to use a reduction as this is simply a sum. Running time with parallelisation is optimal under $\log n$ threads, potentially limiting scalability. In addition, using a schedule call, specifically for CSR and CSC, would allow a more efficient sectioning of the loop being parallelised.

6.3.2 Transpose

The implementation in the project can thread COO with a simple parallel for directive, however cannot parallelise CSC or CSR as the construction of these structures have dependencies that are not parallelisable. of course this isn't necessarily an issue as transposing them can be a matter of treating a CSR as a CSC and a CSC as a CSR, which is constant time.

6.3.3 Scalar multiplication

COO is able to be parallelised with parallel for directives across the array of elements. CSC and CSR are able to be parallelised over the rows for CSR and columns for CSC. Hence complexity with n threads is able to reduce COO to constant time $O(1)$. For a matrix with r rows and c columns, CSR is able to be reduced to $O(c)$ with r threads, and CSC is able to be reduced to $O(r)$ with c threads.

6.3.4 Addition

For COO parallelising is not worth the extra computation time, as creating the structures to parallelise COO requires single threading over the same complexity that is attempting to be reduced. In the case that there was a larger time constant than addition as the operation this may be a useful case, but as addition is a single, constant time operation the parallelisation demonstrated in the project is not useful.

CSC and CSR already have this structure built in with ia. As this is the case parallelising over the rows in CSR and columns in CSC reduces the complexity to $O(((set(i,j) \in N) \cup (set(i,j) \in M))/r)$ with r threads. Similarly in CSC the complexity is reduced by a factor of c when using c threads.

6.3.5 Matrix multiplication

The complexity of the single threaded operation to create a way to parallelise COO is now worthwhile as it is not the dominating factor in complexity. As this is the case, building a version of ia for the row entries (as COO is in row, column order in this program) of the matrix M in the multiplication MN gives the ability to parallelise over the number of rows, additionally ia can be created for the matrix N for the column entries. Both may be parallelised over. However, again with the single threaded complexity required to build the structures, parallelising over both will not lead to a reduction to a linear time complexity, the time complexity can never be less than $O(n)$ where n is the number of non-zero elements. However, the complexity has the potential to be reduced by up factors of rows of M and columns of N ie has linear time complexity $O(rows_N)$ or $O(cols_M)$ as they are equivalent. This assumes running $rows_M \cdot cols_N$ threads and that $n \leq rows_N$ otherwise n will dominate.

For CSR and CSC, the time complexity is also limited by construction time. The transformation from CSR to CSC and vice versa is also $O(n)$ and although with iterating over rows of M and columns of N the potential reduction in time complexity is up to $row_M \cdot cols_N$, it is again limited by the number of non-zero elements.

The implementation of CSC $\cdot$ CSR does away with the precomputation time, as the sparse representations are already in the correct format. Hence the reduction in time complexity is not limited by n and truly becomes $O(rows_N)$ or $(cols_M)$

7 Testing

7.1 Timing

Timing was done using the function `timespec\_get()` in windows sending the `TIME_UTC` macro, and the function `clock_gettime()` in unix sending the `CLOCK_REALTIME` macro. This returned an accurate timing that did not rely on the time a process had spent on the cpu, as this would not represent the results of threading accurately. In testing the project timed construction and destruction time of the matrix as time to construct, but omitted fileio as there are more efficient ways of storing a matrix and file io is tedious, not reflecting the construction time accurately. This is not the case for logging, where file io is included in the construction time. process timing was based on the total execution time of a function, including memory allocation and freeing.

7.2 Systems

This project was tested on two machines. Specifications of both are listed below:

7.2.1 2019 Macbook air

- Operating System: MacOS Mojave
- CPU: 1.6GHz Intel Core i5-8210Y (dual-core, 4 threads, 4MB cache)
- RAM: 16 GB 2133 MHz LPDDR3
- Storage: 512GB PCIE SSD

7.2.2 Custom PC

- Operating System: Windows 10 (64 bit)
- CPU: 4.0GHz Intel Core i7-6700k (quad-core, 8 threads, 8MB cache)
- RAM: 16GB 2400MHz DDR4
- Storage: 256 GB SATA SSD

7.3 Method

The software was run on two machines until completion. Only square matrices were tested hence complexity should scale polynomially. Every permutation of format, file, operation were tested running with the no threading option as well as on threads of value 1-16. Each of these cases were run 100 times and logged to a json. The jsons are included as timing_mac.json and timing_win.json

8 Results

Testing took approximately 14-16 hours, as timing the runtime of the test script was not setup correctly there was not an accurate result. However, the runtime of the software was not timed by the script and instead timed within the program and logged to stdout using the timing flag.

From the results where threading was implemented there was a speed up observed in some cases. Trace is an exception to this, at the same time it is tough to measure as the time constants for building threads are higher than the complexity that's saved by threading, hence all observable data shows no benefit in threading. This may prove to be different for larger matrices. Hence a noticeable slowdown occurred just from building the threads.

Scalar multiplication observed a slow down with the COO format, this appears to be due to an issue with sharing resources. CSR and CSC both show speedups, with optimal speeds at 4 threads. This is to be expected as there is miniscule blocking of each thread, hence the most efficient use of computation is the least number of process swaps, ie 4 threads.

Addition observed significant slow downs in all 3 cases. COO's slow down was due to the time constants creating the data structures for threading negating any potential speed up from threading. However the threaded speeds do scale with the number of threads in an expected fashion, hence there is an observable speedup from threading. CSR and CSC likely also had large time constants for allocation as well as copying the resultant matrix into a the correct memory. Again the speeds scale with threading. Large time constants prevented this from speeding up.

Matrix Multiplication's results differ to all others due to the sheer number of extra operations that are required to be computed. It's now observable that large numbers of threads have little impact on the speed, likely due to threads exceeding their time quantum so many times that the difference in number of swap times required between using all available threads (4) and using excessive amounts (16) is negligible. The same problem would likely show up if attempting to use larger numbers of processors

9 Remarks

Threading was not able to be implemented for construction of the matrix formats due to time restrictions, specifically with regards to while testing was still possible.

10 Conclusion

The software demonstrated the ability to compute sparse matrix operations efficiently, using threading where appropriate. Aside from addition and scalar multiplication for the COO format, all operations where threading was implemented realised a speed up in performance. It's clear that the relationship between time and the number of cores is inversely proportional, due to the reduction in complexity by the factor of the number of cores. The optimal number of threads is not to exceed the number of cores if there is no blocking io required. This does not manifest as well when there is such a huge number of operations that all threads exceed their time quantum regularly.

A Tables

All tables included are the results for the Macbook Air's testing

A.1 Trace

A.1.1 COO

num_threads	float64.in	float128.in	float256.in	float512.in	float1024.in
0	0.00000	0.00000	0.00002	0.00008	0.00034
1	0.00004	0.00005	0.00007	0.00013	0.00043
2	0.00015	0.00017	0.00020	0.00024	0.00045
3	0.00017	0.00021	0.00024	0.00027	0.00043
4	0.00023	0.00025	0.00027	0.00030	0.00044
5	0.00025	0.00027	0.00029	0.00033	0.00049
6	0.00027	0.00028	0.00031	0.00033	0.00050
7	0.00028	0.00031	0.00032	0.00036	0.00051
8	0.00031	0.00032	0.00034	0.00037	0.00053
9	0.00032	0.00033	0.00036	0.00039	0.00056
10	0.00033	0.00035	0.00037	0.00041	0.00057
11	0.00035	0.00037	0.00040	0.00043	0.00059
12	0.00038	0.00040	0.00042	0.00045	0.00062
13	0.00039	0.00040	0.00043	0.00047	0.00064
14	0.00042	0.00044	0.00045	0.00049	0.00067
15	0.00043	0.00044	0.00047	0.00050	0.00067
16	0.00044	0.00047	0.00049	0.00052	0.00069

A.1.2 CSR

num_threads	float64.in	float128.in	float256.in	float512.in	float1024.in
0	0.00000	0.00000	0.00001	0.00004	0.00014
1	0.00004	0.00005	0.00006	0.00010	0.00022
2	0.00016	0.00020	0.00020	0.00021	0.00033
3	0.00017	0.00019	0.00022	0.00024	0.00034
4	0.00026	0.00025	0.00027	0.00029	0.00037
5	0.00025	0.00027	0.00029	0.00031	0.00039
6	0.00027	0.00030	0.00030	0.00032	0.00040
7	0.00028	0.00030	0.00031	0.00034	0.00043
8	0.00030	0.00034	0.00033	0.00036	0.00044
9	0.00033	0.00034	0.00035	0.00037	0.00046
10	0.00034	0.00035	0.00037	0.00040	0.00048
11	0.00036	0.00037	0.00039	0.00042	0.00051
12	0.00037	0.00039	0.00041	0.00045	0.00053
13	0.00040	0.00041	0.00043	0.00046	0.00055
14	0.00041	0.00044	0.00045	0.00048	0.00056
15	0.00043	0.00045	0.00046	0.00050	0.00057
16	0.00044	0.00046	0.00048	0.00051	0.00059

A.1.3 CSC

num_threads	float64.in	float128.in	float256.in	float512.in	float1024.in
0	0.00000	0.00001	0.00001	0.00004	0.00015
1	0.00005	0.00005	0.00006	0.00011	0.00024
2	0.00016	0.00018	0.00020	0.00025	0.00044
3	0.00017	0.00021	0.00023	0.00028	0.00042
4	0.00023	0.00025	0.00028	0.00031	0.00043
5	0.00025	0.00028	0.00030	0.00032	0.00042
6	0.00027	0.00029	0.00030	0.00034	0.00046
7	0.00028	0.00031	0.00031	0.00036	0.00048
8	0.00030	0.00033	0.00034	0.00038	0.00048
9	0.00032	0.00034	0.00035	0.00040	0.00051
10	0.00036	0.00035	0.00037	0.00041	0.00053
11	0.00036	0.00038	0.00039	0.00045	0.00056
12	0.00037	0.00040	0.00041	0.00046	0.00058
13	0.00039	0.00043	0.00043	0.00049	0.00060
14	0.00041	0.00044	0.00045	0.00050	0.00061
15	0.00046	0.00043	0.00048	0.00052	0.00063
16	0.00044	0.00047	0.00049	0.00054	0.00064

A.2 Transpose

A.2.1 COO

num_threads	float64.in	float128.in	float256.in	float512.in	float1024.in
0	0.00000	0.00001	0.00002	0.00010	0.00041
1	0.00004	0.00005	0.00008	0.00018	0.00058
2	0.00014	0.00017	0.00019	0.00027	0.00056
3	0.00015	0.00018	0.00021	0.00028	0.00052
4	0.00019	0.00023	0.00024	0.00029	0.00050
5	0.00021	0.00023	0.00025	0.00031	0.00055
6	0.00022	0.00025	0.00025	0.00032	0.00055
7	0.00024	0.00026	0.00027	0.00033	0.00056
8	0.00025	0.00027	0.00029	0.00035	0.00058
9	0.00025	0.00029	0.00030	0.00036	0.00061
10	0.00028	0.00029	0.00031	0.00037	0.00062
11	0.00029	0.00032	0.00033	0.00040	0.00063
12	0.00030	0.00032	0.00034	0.00040	0.00065
13	0.00031	0.00034	0.00036	0.00041	0.00067
14	0.00032	0.00034	0.00038	0.00043	0.00068
15	0.00034	0.00038	0.00040	0.00045	0.00069
16	0.00035	0.00038	0.00041	0.00046	0.00071

A.2.2 CSR

num_threads	float64.in	float128.in	float256.in	float512.in	float1024.in
0	0.00002	0.00005	0.00016	0.00057	0.00221
1	0.00002	0.00005	0.00016	0.00056	0.00218
2	0.00002	0.00005	0.00015	0.00056	0.00216
3	0.00002	0.00005	0.00016	0.00057	0.00214
4	0.00002	0.00006	0.00016	0.00057	0.00216
5	0.00002	0.00005	0.00016	0.00057	0.00219
6	0.00002	0.00005	0.00016	0.00058	0.00218
7	0.00002	0.00006	0.00016	0.00057	0.00224
8	0.00002	0.00006	0.00016	0.00057	0.00219
9	0.00002	0.00005	0.00016	0.00058	0.00217
10	0.00002	0.00005	0.00016	0.00056	0.00218
11	0.00002	0.00005	0.00015	0.00057	0.00218
12	0.00002	0.00005	0.00016	0.00058	0.00222
13	0.00002	0.00005	0.00016	0.00057	0.00220
14	0.00002	0.00006	0.00016	0.00058	0.00218
15	0.00002	0.00005	0.00016	0.00057	0.00223
16	0.00002	0.00006	0.00016	0.00057	0.00217

A.2.3 CSC

num_threads	float64.in	float128.in	float256.in	float512.in	float1024.in
0	0.00001	0.00003	0.00016	0.00057	0.00210
1	0.00003	0.00005	0.00016	0.00062	0.00206
2	0.00002	0.00005	0.00016	0.00060	0.00207
3	0.00002	0.00005	0.00016	0.00062	0.00207
4	0.00002	0.00005	0.00017	0.00062	0.00209
5	0.00002	0.00005	0.00017	0.00061	0.00210
6	0.00002	0.00005	0.00017	0.00061	0.00204
7	0.00002	0.00005	0.00016	0.00061	0.00204
8	0.00002	0.00005	0.00017	0.00062	0.00210
9	0.00002	0.00005	0.00017	0.00062	0.00212
10	0.00003	0.00005	0.00017	0.00061	0.00208
11	0.00002	0.00005	0.00017	0.00061	0.00209
12	0.00002	0.00005	0.00016	0.00062	0.00208
13	0.00002	0.00005	0.00017	0.00060	0.00207
14	0.00002	0.00005	0.00016	0.00061	0.00205
15	0.00002	0.00005	0.00017	0.00060	0.00208
16	0.00002	0.00005	0.00017	0.00062	0.00207

A.3 Scalar Multiplication

A.3.1 COO

num_threads	float64.in	float128.in	float256.in	float512.in	float1024.in
0	0.00000	0.00001	0.00003	0.00013	0.00054
1	0.00004	0.00005	0.00008	0.00019	0.00063
2	0.00016	0.00017	0.00022	0.00028	0.00061
3	0.00019	0.00022	0.00025	0.00030	0.00057
4	0.00023	0.00025	0.00028	0.00034	0.00055
5	0.00026	0.00027	0.00030	0.00036	0.00062
6	0.00028	0.00029	0.00031	0.00037	0.00062
7	0.00031	0.00031	0.00033	0.00039	0.00063
8	0.00031	0.00032	0.00035	0.00042	0.00066
9	0.00032	0.00034	0.00037	0.00043	0.00069
10	0.00036	0.00036	0.00041	0.00045	0.00070
11	0.00037	0.00039	0.00042	0.00047	0.00072
12	0.00039	0.00041	0.00045	0.00050	0.00075
13	0.00039	0.00043	0.00045	0.00052	0.00077
14	0.00046	0.00045	0.00047	0.00054	0.00079
15	0.00044	0.00049	0.00051	0.00056	0.00081
16	0.00046	0.00047	0.00052	0.00058	0.00082

A.3.2 CSR

num_threads	float64.in	float128.in	float256.in	float512.in	float1024.in
0	0.00001	0.00003	0.00013	0.00051	0.00197
1	0.00005	0.00008	0.00017	0.00054	0.00194
2	0.00018	0.00023	0.00030	0.00053	0.00151
3	0.00021	0.00023	0.00032	0.00052	0.00130
4	0.00027	0.00029	0.00037	0.00053	0.00118
5	0.00029	0.00032	0.00039	0.00060	0.00135
6	0.00031	0.00035	0.00041	0.00061	0.00132
7	0.00034	0.00035	0.00044	0.00062	0.00131
8	0.00036	0.00039	0.00046	0.00066	0.00137
9	0.00038	0.00041	0.00049	0.00068	0.00143
10	0.00040	0.00046	0.00052	0.00072	0.00142
11	0.00043	0.00045	0.00054	0.00072	0.00145
12	0.00046	0.00049	0.00057	0.00077	0.00150
13	0.00051	0.00050	0.00060	0.00077	0.00153
14	0.00051	0.00056	0.00063	0.00081	0.00153
15	0.00053	0.00057	0.00065	0.00084	0.00154
16	0.00055	0.00059	0.00070	0.00087	0.00158

A.3.3 CSC

num_threads	float64.in	float128.in	float256.in	float512.in	float1024.in
0	0.00001	0.00002	0.00013	0.00049	0.00178
1	0.00004	0.00006	0.00017	0.00054	0.00184
2	0.00018	0.00020	0.00029	0.00058	0.00144
3	0.00021	0.00024	0.00032	0.00055	0.00131
4	0.00027	0.00029	0.00038	0.00056	0.00119
5	0.00028	0.00033	0.00040	0.00062	0.00135
6	0.00031	0.00033	0.00041	0.00061	0.00136
7	0.00033	0.00034	0.00043	0.00064	0.00134
8	0.00035	0.00038	0.00046	0.00067	0.00134
9	0.00037	0.00044	0.00049	0.00070	0.00142
10	0.00039	0.00043	0.00051	0.00072	0.00142
11	0.00042	0.00044	0.00055	0.00074	0.00145
12	0.00045	0.00048	0.00057	0.00079	0.00149
13	0.00046	0.00052	0.00059	0.00080	0.00153
14	0.00050	0.00053	0.00061	0.00084	0.00153
15	0.00052	0.00056	0.00064	0.00086	0.00157
16	0.00055	0.00057	0.00067	0.00088	0.00156

A.4 Addition

A.4.1 COO by COO

num_threads	float64.in	float128.in	float256.in	float512.in	float1024.in
0	0.00001	0.00003	0.00019	0.00069	0.00265
1	0.00015	0.00038	0.00097	0.00317	0.00964
2	0.00024	0.00044	0.00086	0.00260	0.00734
3	0.00025	0.00042	0.00079	0.00236	0.00671
4	0.00028	0.00044	0.00077	0.00226	0.00622
5	0.00030	0.00046	0.00083	0.00241	0.00687
6	0.00030	0.00047	0.00083	0.00236	0.00667
7	0.00031	0.00047	0.00082	0.00236	0.00667
8	0.00034	0.00049	0.00086	0.00235	0.00659
9	0.00034	0.00051	0.00087	0.00239	0.00684
10	0.00035	0.00051	0.00086	0.00240	0.00679
11	0.00037	0.00053	0.00087	0.00239	0.00679
12	0.00039	0.00055	0.00091	0.00241	0.00682
13	0.00040	0.00057	0.00093	0.00243	0.00688
14	0.00041	0.00058	0.00093	0.00244	0.00683
15	0.00043	0.00059	0.00097	0.00246	0.00684
16	0.00045	0.00061	0.00098	0.00246	0.00686

A.4.2 CSR by CSR

num_threads	float64.in	float128.in	float256.in	float512.in	float1024.in
0	0.00001	0.00002	0.00008	0.00028	0.00110
1	0.00012	0.00030	0.00096	0.00241	0.00782
2	0.00022	0.00035	0.00087	0.00195	0.00613
3	0.00023	0.00038	0.00081	0.00178	0.00537
4	0.00027	0.00039	0.00081	0.00173	0.00509
5	0.00028	0.00042	0.00086	0.00183	0.00534
6	0.00029	0.00041	0.00087	0.00181	0.00532
7	0.00030	0.00042	0.00085	0.00179	0.00526
8	0.00032	0.00044	0.00088	0.00180	0.00528
9	0.00032	0.00046	0.00090	0.00183	0.00524
10	0.00034	0.00047	0.00091	0.00183	0.00525
11	0.00035	0.00048	0.00092	0.00184	0.00537
12	0.00038	0.00051	0.00094	0.00186	0.00529
13	0.00039	0.00053	0.00094	0.00187	0.00533
14	0.00040	0.00053	0.00097	0.00189	0.00529
15	0.00041	0.00055	0.00098	0.00191	0.00527
16	0.00043	0.00056	0.00099	0.00192	0.00540

A.4.3 CSC by CSC

num_threads	float64.in	float128.in	float256.in	float512.in	float1024.in
0	0.00001	0.00002	0.00008	0.00027	0.00104
1	0.00010	0.00020	0.00050	0.00141	0.00644
2	0.00021	0.00030	0.00056	0.00136	0.00546
3	0.00022	0.00033	0.00056	0.00125	0.00509
4	0.00026	0.00035	0.00057	0.00121	0.00499
5	0.00028	0.00037	0.00059	0.00127	0.00509
6	0.00029	0.00037	0.00059	0.00128	0.00508
7	0.00029	0.00038	0.00061	0.00127	0.00504
8	0.00031	0.00043	0.00061	0.00127	0.00501
9	0.00032	0.00041	0.00063	0.00130	0.00521
10	0.00033	0.00042	0.00066	0.00131	0.00514
11	0.00036	0.00044	0.00065	0.00134	0.00508
12	0.00036	0.00045	0.00070	0.00137	0.00519
13	0.00038	0.00046	0.00070	0.00134	0.00513
14	0.00039	0.00049	0.00071	0.00138	0.00515
15	0.00042	0.00049	0.00074	0.00140	0.00516
16	0.00042	0.00051	0.00075	0.00138	0.00516

A.5 Matrix Multiplication

A.5.1 CSR by CSC

num_threads	float64.in	float128.in	float256.in	float512.in	float1024.in
nothreading	0.00066	0.00484	0.03747	0.27235	2.04424
1	0.00072	0.00531	0.04191	0.32431	2.73768
2	0.00066	0.00336	0.02386	0.18457	1.69653
3	0.00052	0.00270	0.02077	0.16708	1.53351
4	0.00050	0.00247	0.01821	0.14215	1.33409
5	0.00056	0.00278	0.02420	0.16321	1.36455
6	0.00057	0.00272	0.01995	0.15516	1.35996
7	0.00054	0.00260	0.01875	0.14954	1.36590
8	0.00056	0.00256	0.01861	0.15361	1.33894
9	0.00061	0.00274	0.02007	0.15243	1.36390
10	0.00059	0.00269	0.01962	0.15212	1.33382
11	0.00060	0.00272	0.01852	0.14831	1.34023
12	0.00064	0.00276	0.01912	0.15111	1.33759
13	0.00064	0.00272	0.01916	0.14817	1.35969
14	0.00064	0.00280	0.01930	0.15240	1.32151
15	0.00066	0.00271	0.01883	0.15328	1.30709
16	0.00072	0.00275	0.01889	0.15263	1.31976

A.5.2 COO by COO

num_threads	float64.in	float128.in	float256.in	float512.in	float1024.in
nothreading	0.00067	0.00444	0.03552	0.26171	2.03733
1	0.00096	0.00613	0.04930	0.37090	2.88142
2	0.00075	0.00406	0.02861	0.21330	1.70452
3	0.00066	0.00338	0.02585	0.19240	1.51072
4	0.00065	0.00316	0.02221	0.16769	1.32695
5	0.00072	0.00346	0.02480	0.19638	1.33016
6	0.00073	0.00341	0.02445	0.18229	1.32522
7	0.00071	0.00327	0.02368	0.17904	1.33077
8	0.00072	0.00319	0.02312	0.17643	1.32912
9	0.00076	0.00340	0.02497	0.17632	1.32806
10	0.00073	0.00333	0.02404	0.17627	1.32862
11	0.00077	0.00328	0.02356	0.17536	1.32909
12	0.00079	0.00338	0.02394	0.17527	1.32573
13	0.00080	0.00341	0.02456	0.17603	1.32621
14	0.00081	0.00340	0.02390	0.17493	1.33217
15	0.00084	0.00339	0.02375	0.17474	1.33283
16	0.00085	0.00338	0.02363	0.17373	1.33111

A.5.3 CSR by CSR

num_threads	float64.in	float128.in	float256.in	float512.in	float1024.in
nothreading	0.00072	0.00542	0.04241	0.30032	2.13081
1	0.00114	0.00799	0.05864	0.45283	3.35126
2	0.00084	0.00521	0.03236	0.24472	1.94647
3	0.00075	0.00423	0.03158	0.25256	1.88681
4	0.00075	0.00448	0.03161	0.25129	1.80095
5	0.00078	0.00460	0.03323	0.26386	1.79841
6	0.00077	0.00426	0.03191	0.25335	1.77850
7	0.00076	0.00435	0.03250	0.25321	1.77294
8	0.00079	0.00441	0.03215	0.25252	1.78520
9	0.00079	0.00448	0.03211	0.25381	1.76888
10	0.00078	0.00430	0.03171	0.25235	1.72006
11	0.00081	0.00431	0.03154	0.25343	1.75395
12	0.00080	0.00428	0.03198	0.25409	1.75752
13	0.00081	0.00427	0.03221	0.25275	1.73597
14	0.00083	0.00422	0.03163	0.25270	1.79097
15	0.00086	0.00419	0.03113	0.25327	1.73102
16	0.00087	0.00420	0.03167	0.25366	1.76342

A.5.4 CSC by CSC

num_threads	float64.in	float128.in	float256.in	float512.in	float1024.in
nothreading	0.00063	0.00385	0.03177	0.24006	1.90560
1	0.00071	0.00528	0.04200	0.31221	2.63747
2	0.00057	0.00335	0.02345	0.17373	1.61524
3	0.00051	0.00273	0.02074	0.15629	1.38024
4	0.00050	0.00267	0.01686	0.13613	1.23313
5	0.00055	0.00282	0.01973	0.15043	1.20344
6	0.00055	0.00272	0.01993	0.14930	1.19518
7	0.00055	0.00266	0.01838	0.14733	1.20403
8	0.00057	0.00272	0.01832	0.14292	1.20155
9	0.00057	0.00271	0.01961	0.14712	1.19192
10	0.00057	0.00275	0.01941	0.15049	1.15924
11	0.00060	0.00272	0.01815	0.14545	1.18327
12	0.00062	0.00279	0.01939	0.13992	1.16692
13	0.00063	0.00279	0.01952	0.14503	1.17481
14	0.00064	0.00269	0.01887	0.14062	1.17415
15	0.00065	0.00272	0.01834	0.14162	1.19451
16	0.00069	0.00281	0.01890	0.14360	1.20391

A.6 Construction Time

A.6.1 COO

float64.in	float128.in	float256.in	float512.in	float1024.in
0.00121	0.00440	0.01750	0.06945	0.27831

int64.in	int128.in	int256.in	int512.in	int1024.in
0.00076	0.00275	0.01069	0.04234	0.16839

A.6.2 CSR

float64.in	float128.in	float256.in	float512.in	float1024.in
0.00129	0.00451	0.01824	0.07306	0.29106

int64.in	int128.in	int256.in	int512.in	int1024.in
0.00080	0.00277	0.01070	0.04187	0.16412

A.6.3 CSC

float64.in	float128.in	float256.in	float512.in	float1024.in
0.00133	0.00474	0.01876	0.07585	0.30541

int64.in	int128.in	int256.in	int512.in	int1024.in
0.00086	0.00296	0.01126	0.04501	0.18791