Introduction:

For this report I have implemented a collection of C functions that are all fully functioning. All functions will be tested for various inputs and compared to other relevant functions implemented. All files will be in the .zip file with the raw data gathered from test ran with Modi.
All results in this report has used the given benchmark file to print runtimes. As the bench_transpose and bench_matmul does measure both the time taken to allocate memory and compute the expected golden result, the timings will not be what was intended. But all results are still valid for comparisons as they are just considered to be ran on a slow computer.

How the tests are run:

To run the tests shown in this rapport, simply just compile and run the benchmark file with './benchmark'.
For the test where it is required to have the input scale relative to the thread, I have used the file 'submit-all.sh' to pass in the same value as number off threads run. The benchmark file is changed so that it takes one int argument. This argument is used to scale the inputs for the tests. If no input is given the default value '1' is set.
Running the tests on Modi can be done by setting the value 'MODI_ACTIVE = 1' in the benchmark file so that the correct tests will be available to be run Modi. Then following the instructions in the '' submit-all.sh' file will make the test get submitted. This will produce an output file '`slurm-12345.out` which I then have used the python file 'Data_cleaning.py' to process and export to .csv. The .csv file is then used to plot graphs in Excel. All plots together with the data used is to be found in the 'Scaling_plots.xlsx' file. I have also listed what dataset is used for each graph. These raw dataset are also to be found in the folder 'Data_sets'.

All tests are marked with either "local test" or "Modi test" to clarify where they have been run. Tests run on "local test" are run on a **MacBook Pro: Dual-Core Intel Core i5, 2,3 GHz, 8 GB RAM.**

Testing all transpose functions for various inputs:

For this input size I tried different T-values and could conclude that the best T-value for this input size on this machine was T=30.

''Local test'', Runs = 50

N=900	Transpose (7ms)	Transpose_blocked (3 ms)
M=1800	Transpose_parallel (4 ms)	Transpose_blocked_parallel (2 ms) (T =30)

Calculating speedups:

$$\frac{\text{transpose}}{\text{transpose parallel}} = \frac{7\text{ms}}{4\text{ms}} = 1{,}75 \qquad \frac{\text{transpose}}{\text{transpose blocked}} = \frac{7\text{ms}}{3\text{ms}} = 2{,}3\ldots$$

$$\frac{\text{transpose}}{\text{transpose blocked parallel}} = \frac{7\text{ms}}{2\text{ms}} = 3{,}5 \qquad \frac{\text{transpose blocked}}{\text{transpose blocked parallel}} = \frac{3ms}{2ms} = 1{,}5$$

From these speedups we get an idea of how much of an improvement the different implementations are. This is although calculations are made with ms which is rounded off, so they are not precise, but they can still tell something about the relationship between each function.

‘’Local test’’, Runs = 100

N=3300	Transpose (6ms)	Transpose_blocked (4 ms)
M=900	Transpose_parallel (5 ms)	Transpose_blocked_parallel (3 ms) (T =30)

$$\frac{\text{transpose}}{\text{transpose parallel}} = \frac{6\text{ms}}{5\text{ms}} = 1{,}2$$

$$\frac{\text{transpose}}{\text{transpose blocked}} = \frac{6\text{ms}}{4\text{ms}} = 1{,}5$$

$$\frac{\text{transpose}}{\text{transpose blocked parallel}} = \frac{6\text{ms}}{3\text{ms}} = 2$$

$$\frac{\text{transpose blocked}}{\text{transpose blocked parallel}} = \frac{4ms}{3ms} = 1{,}33\ldots$$

‘’Local test’’, Runs = 10

N=12000	Transpose (1839ms)	Transpose_blocked (491 ms)
M=12000	Transpose_parallel (1059 ms)	Transpose_blocked_parallel (348 ms) (T =30)

$$\frac{\text{transpose}}{\text{transpose parallel}} = \frac{1839\text{ms}}{1059\text{ms}} = 1{,}737$$

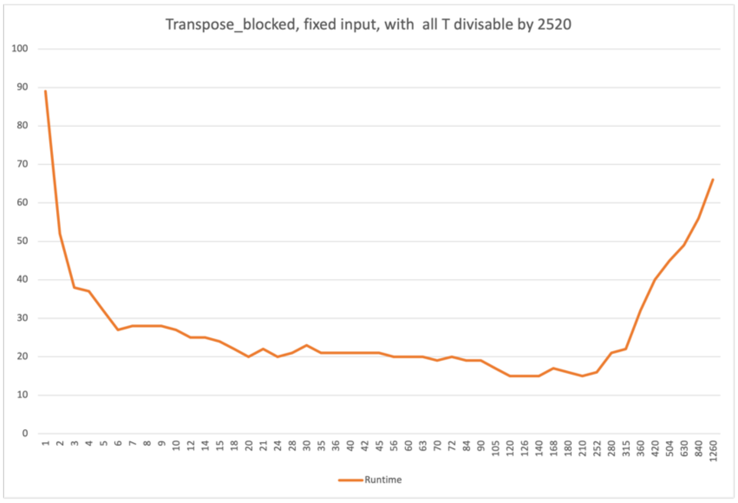
$$\frac{\text{transpose}}{\text{transpose blocked}} = \frac{1839\text{ms}}{491\text{ms}} = 3{,}745$$

$$\frac{\text{transpose}}{\text{transpose blocked parallel}} = \frac{1839\text{ms}}{348\text{ms}} = 5{,}284$$

$$\frac{\text{transpose blocked}}{\text{transpose blocked parallel}} = \frac{491\text{ms}}{348\text{ms}} = 1{,}411$$

I see that for various values my tests seem to run and produce the right result. For the last test for n=m=12000 I get speedup values that are closer to the actual value. And as I have scaled the input size, I see how the parallel functions really start to be efficent.
I see that just parallelizing the transpose function made it roughly 1,7 times faster. But as for transpose the improvement made it around 3,7 times faster, and parallelizing that, it went to about 5,3 times faster.

Testing for the best T-value:
To get the best T-value I have changed the benchmark file a bit. It does now take for all transpose functions an int T. By iterating over every divisible number of m=n=2520. This gives me the data that I have plotted in the following graph.
runs=500
‘’Modi test’’



From this is see that the most optimal T-value must be around 105-250 for tests run on Modi with this input size. I see that a small T-value makes bad runtimes as well as a too large value does.

Transpose
As it is a nested forloop it will scale O(n^2). And because its cache performance is a bit weakened by the fact that it must fetch the values in the A-array in column major it will have higher chance of cache misses.

Transpose parallel
For Transpose parallel I used '**pragma omp parallel for**' as this in tests was the fastest. It makes sense to use this clause as all loops and calculations are independent.

Transpose blocked
For transpose-blocked, T can be seen as a value that brakes up the matrix into pieces, hence the CPU have faster access to the data as it is cached. This is what enhances the performance as we want to use as much data as possible from the cache-line to reduce cache misses.
We don't want T to be too big as we will have an area that will not fit into cache.
For a small T we will have a too tiny area instead and are not using very much of each cache-line. This is also what the graph is showing in my test for best T-value.

Transpose blocked parallel
For transpose-blocked-parallel I used the cluse 'omp parallel for schedule(dynamic)' as I know from the lecture 'concurrent programming 1' that 'it assigns each thread an iteration and is given more iterations when its finishes'. This was the most optimal result when run, and as transpose blocked parallel is a quadruple forloop where the inner forloops depend on the outer, it makes good sense to assigning one thread to each outer forloop to ensure a correct result each time.

Calculating the speedup for **transpose parallel** and **transpose blocked parallel** all for, n =m= 12000 and runs = 10:
''Local test''

N=M	Threads	transpose_parallel	transpose_blocked_parallel (T=30)
12000	1	1810 ms	675 ms
12000	2	1025 ms	349 ms

To see what fraction of the code that can be parallelized for transpose-parallel and transpose-blocked-parallel ill use Amdahls law. I'll use the speedups I have calculated and then isolate p in the equation:

$$S(N) = \frac{1}{(1-p)+\frac{p}{N}} \Leftrightarrow p = \frac{N \cdot (S(N)-1)}{S(N) \cdot (N-1)}$$

For transpose parallel:

$$speedup = \frac{\text{thread=1}}{\text{thread=2}} = \frac{1810\text{ms}}{1025\text{ms}} = 1{,}766 \qquad p = \frac{2\cdot(1{,}766-1)}{1{,}766\cdot(2-1)} = 0{,}87$$

Meaning that 87% of my program could be parallelized in this specific example.

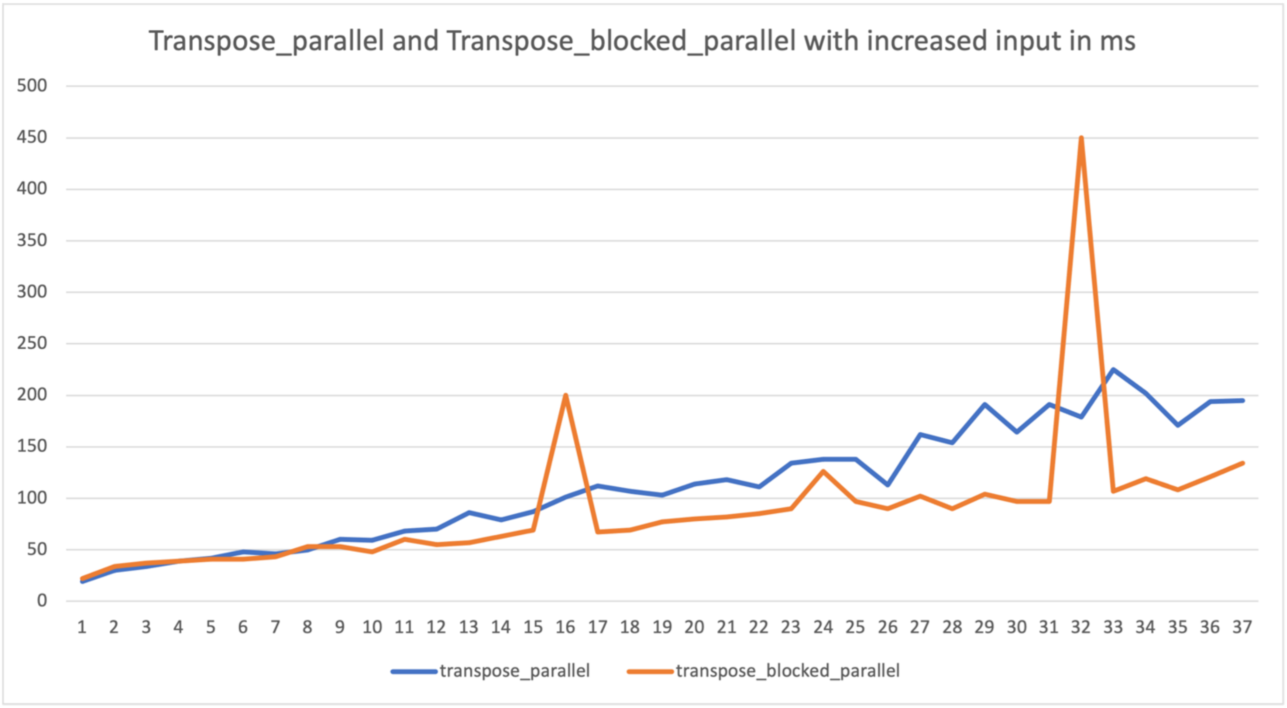
For transpose blocked parallel (T=30)

$speedup = \frac{\text{thread=1}}{\text{thread=2}} = \frac{675\text{ms}}{349\text{ms}} = 1{,}934$ $\quad\quad p = \frac{3\cdot(1{,}93-1)}{1{,}93\cdot(3-1)} = 0{,}73$

Meaning that 73% of my program could be parallelized in this specific example.

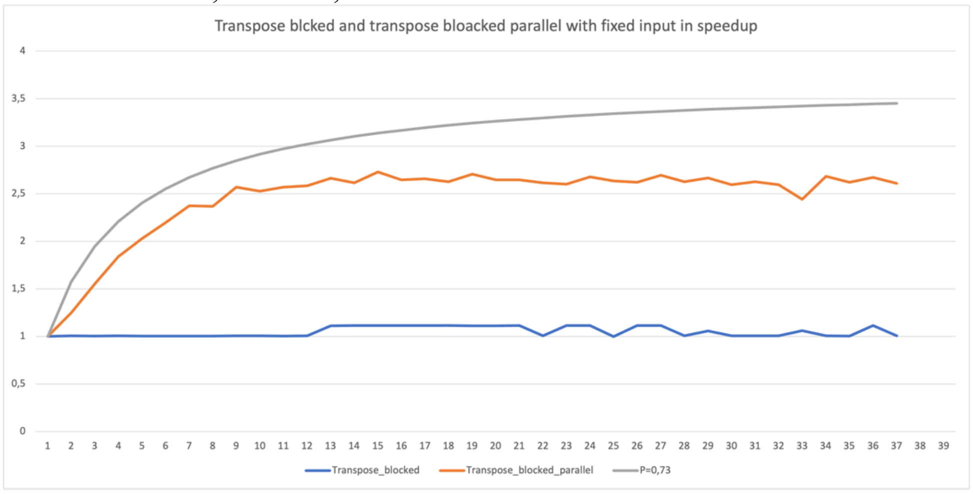
For these values it seems that **transpose parallel** has the best scalability compared to **transpose blocked parallel** as it has the highest p-value which indicates that more of the code can be parallelized. To examine this further I will run a test on Modi where I will increase the number of threads and workload for both functions.

Graph for **Transposed Blocked Parallel** (T=100) and Transpose Parallel
Input : M=20000, N=500*threads, runs = 100
''Modi test''



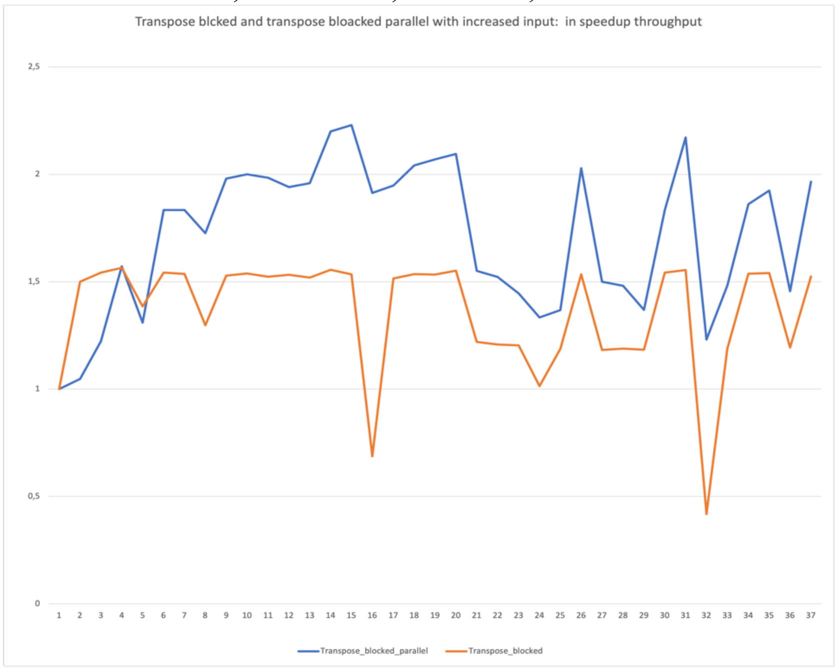
I see that for an increased input for the functions transpose-parallel and transpose-blocked-parallel, I get the best runtime for transpose-blocked-parallel. This is even though I estimated a higher p-value for transpose-parallel. The reason for this could be that the cache-performance is better for transpose-blocked-parallel, as discussed earlier, and good cache-performance seem to have a higher importance in the matter of scaling.

‘’Modi test’’
M=N=K=20000, runs=10, T=50



Strong scaling means using parallelizing code to run a fixed problem size faster. So if I want to calculate a matrix that took an hour to calculate, I will by throwing more threads at it, make it run in less time.
Ideally, I want that if threads double I want speedup to double. This meaning I get linear speedup. This is in practice not possible, as there is an upper bound for how fast It will be able to run depending on how much of the code is parallelable.
I my case is see that I do not have the desired strong scaling.

‘’Modi test’’
N = 400 * threads, M = 10000, runs = 10, T=50



Weak scaling is what we refer to when we're talking about using parallel computing to run a larger problem. If I wanted to run a bigger matrix in the same amount of time that would be an example of weak scaling. This means that as input must increase relative to number of threads, I should get a linear graph, if I was to have weak scaling. In my case I do not have weak scaling.

Testing all matmul functions for various inputs and calculating speedups:
''Local tests''

Runs = 100		
N = 200	Matmul = 7(ms)	Matmul_transpose = 7 (ms)(T=25)
M = 300	Matmul_locality = 2(ms)	Matmul_locality_par = 1 (ms)
K = 100	Matmul_parallel= 3(ms)	Matmul_transpose_par = 2(ms)(T=25)

Runs = 50		
N = 400	Matmul = 67 (ms)	Matmul_transpose = 60 (ms)(T=25)
M = 200	Matmul_locality = 18(ms)	Matmul_locality_par = 10 (ms)
K= 650	Matmul_parallel= 25(ms)	Matmul_transpose_par = 19(ms)(T=25)

Runs = 10		
N = 1200	Matmul = 3858 (ms)	Matmul_transpose= 2982 (ms)(T=25)
M = 1200	Matmul_locality = 1693(ms)	Matmul_locality_par = 991(ms)
K= 1200	Matmul_parallel= 1698(ms)	Matmul_transpose_par = 1657(ms)(T=25)

$$\frac{\text{Matmul}}{\text{Matmul locality}} = \frac{3858\text{ms}}{1693\text{ms}} = 2{,}279 \qquad \frac{\text{Matmul}}{\text{Matmul parallel}} = \frac{3858\text{ms}}{1698\text{ms}} = 2{,}272$$

$$\frac{\text{Matmul}}{\text{Matmul transpose}} = \frac{3858\text{ms}}{2982\text{ms}} = 1{,}294 \qquad \frac{\text{Matmul}}{\text{Matmul locality par}} = \frac{3858\text{ms}}{991\text{ms}} = 3{,}893$$

$$\frac{\text{Matmul}}{\text{Matmul transpose par}} = \frac{3858\text{ms}}{1657\text{ms}} = 2{,}328$$

I see that the locality changes in matmul-locality decreases the runtime tremendously. The speedup is almost the same or just above matmul transpose, matmul transpose parallel and matmul parallel. This is as, for matmul-locality we have fixed this issue by moving the A matrix out to the second forloop. This means that I add on the p value in the loop, hence a smaller chance of cache miss and a significantly increased performance as seen in the data and the speedup calculations.

For all the multiplication functions I see that **matmul** is the slowest one. This is because I have bad locality when the functions must fetch the values for B. Here we are multiplying the p value which is the inner most forloop. This means that it have to make bigger jumps in the dynamic array to get the desired value hence an increased chance of cache miss and thereby a slowed cache performance.

For **matmul-transpose** I get an increase in speedup as well. This is because we do the transposing first so that it now can access the columns in the B matrix in row major form. This is as discussed earlier better for cache performance. But for this function we do have two steps, both the transposing and the multiplication part which does take some time. This means the function in overall do outperform the matmul function with some speedup, but it is not as good as the matmul-locality for this specific input.

Choosing my OMP clause:
For matmul parallel I use the clause reduction. I here explicitly tell OpenMP the what operator to use with the syntax: '#pragma omp parallel for reduction(+:acc)'. This ensures that I don't get a race condition when adding up the 'acc'.
For matmul locality parallel I use the syntax '#pragma omp parallel for' on the forloop that sets the output matrix C to 0 and for the multiplication calculations I use the '#pragma omp parallel for reduction(+:a)'.

For the Transpose blocked parallel I used the clause 'schedule(dynamic)' as discussed earlier. This means that each thread is assigned an iteration.

To see what fraction of the code that can be parallelized for matmul-locality-parallel and matmul-transpose-parallel ill use Amdahls law. I'll use the speedups and insert in the formula for p.

''Local test''

N=M	Threads	**matmul_locality_parallel**	**matmul_transpose_parallel** (T=25)
500	1	59 ms	166 ms
500	2	38 ms	97 ms

$$p = \frac{N \cdot (S(N) - 1)}{S(N) \cdot (N - 1)}$$

For matmul locality parallel:

$speedup = \frac{\text{thread=1}}{\text{thread=2}} = \frac{59\text{ ms}}{38\text{ ms}} = 1{,}5526$ $\quad p = \frac{2\cdot(1{,}5526-1)}{1{,}5526\cdot(2-1)} = 0{,}712$

Meaning that around 71% of my program could be parallelized in this specific example.

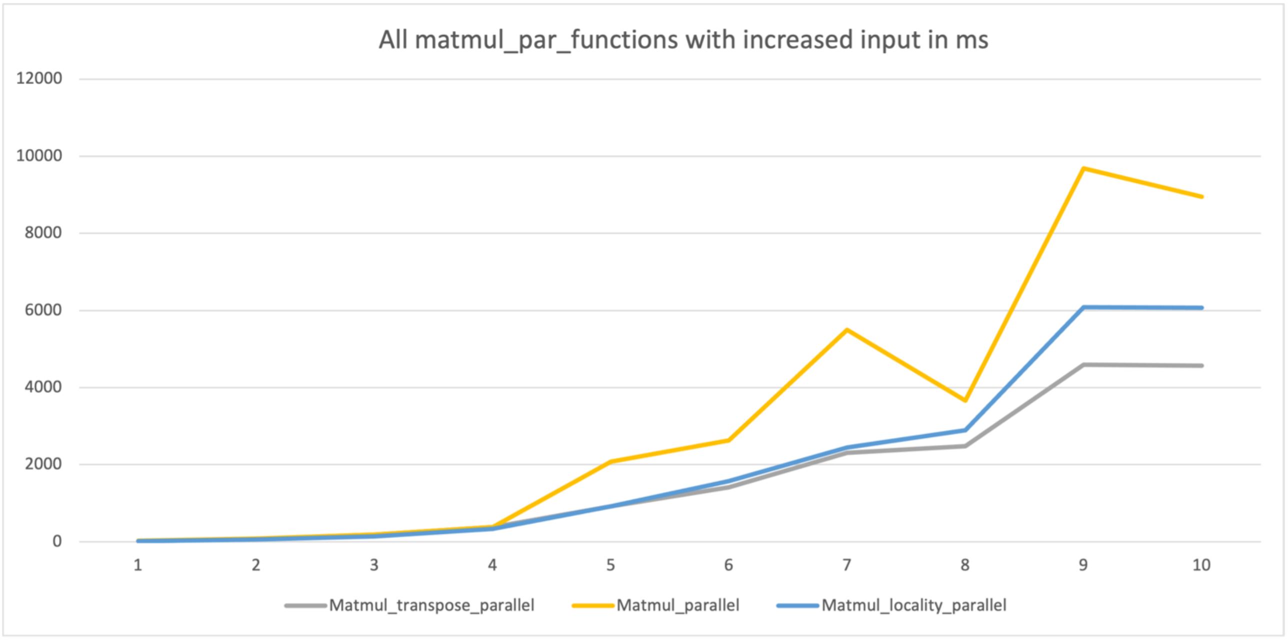
For matmul transpose parallel (T=25)

$speedup = \frac{\text{thread=1}}{\text{thread=2}} = \frac{166\text{ ms}}{97\text{ ms}} = 1{,}7113$ $\quad p = \frac{2\cdot(1{,}7113-1)}{1{,}7113\cdot(2-1)} = 0{,}8313$

Meaning that 83% of my program could be parallelized in this specific example.

From these p-values I see that more of the matmul-transpose-parallel function could be parallelized hence it should have the best scalability. So even though the matmul-locality-parallel function is faster on these test, matmul-transpose-parallel should be able to outperform matmul-locality-parallel for a large enough input.

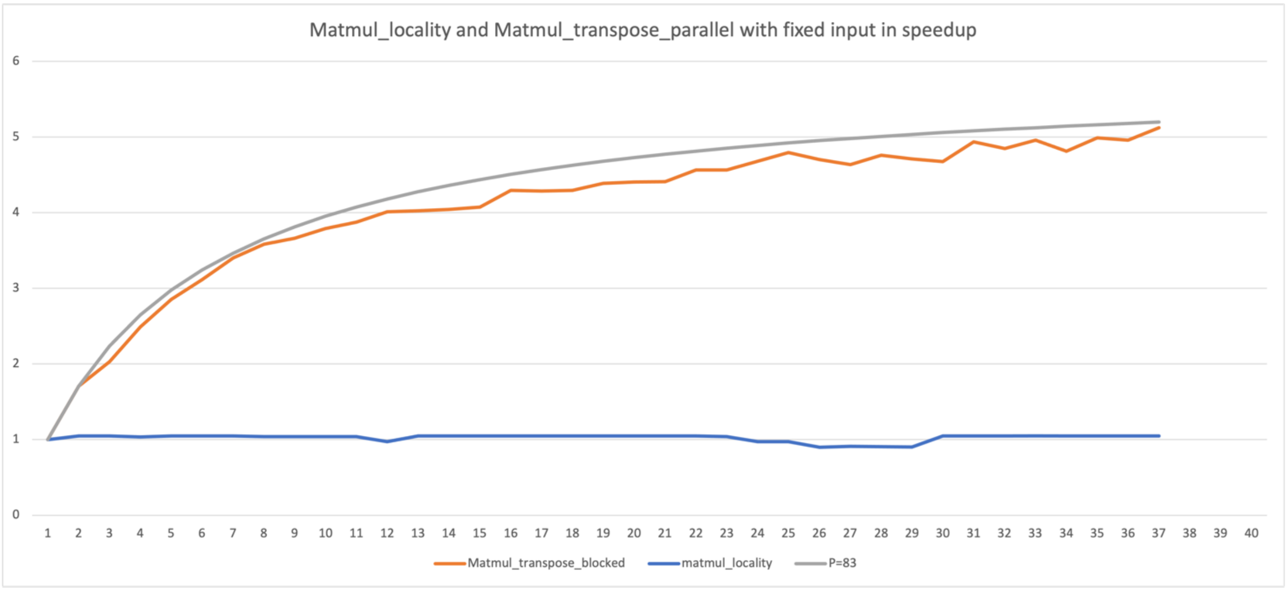
Input = 250*threads
‘’Modi test’’



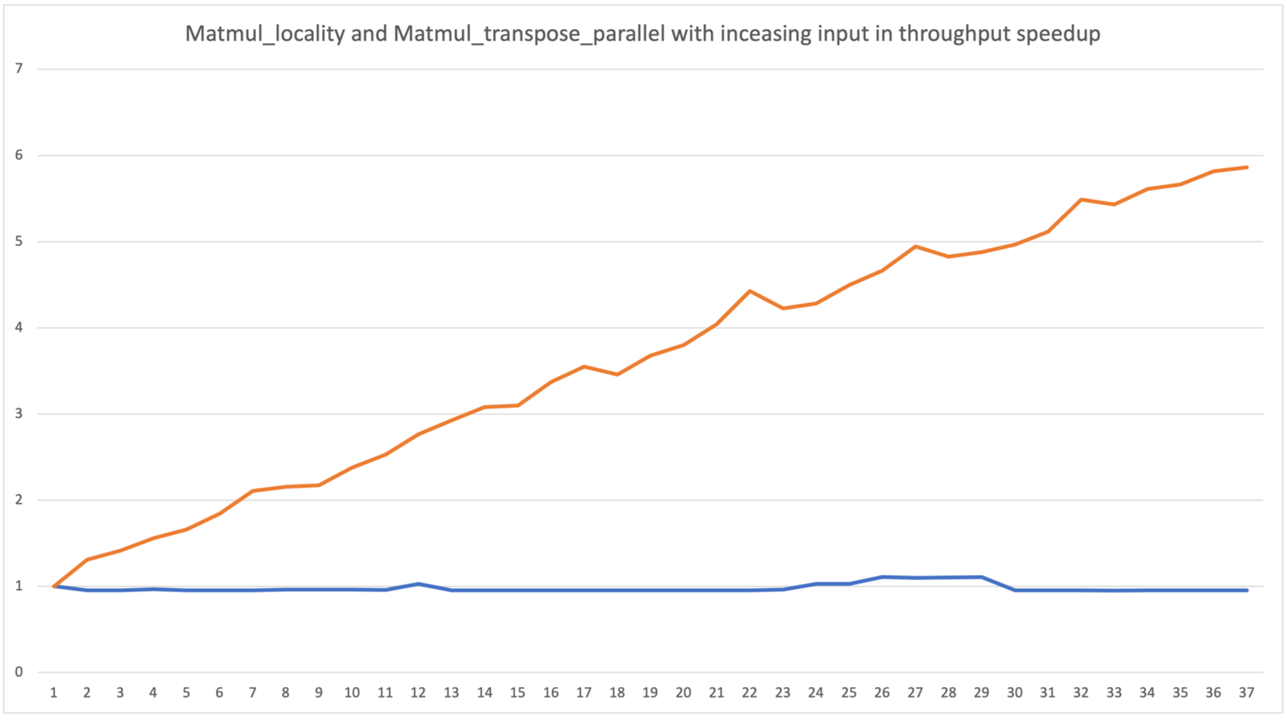
I see that the matmul transpose parallel has the best scalability for a large enough input.

So, when choosing the workload to examine strong scalability I will use an input size above $250 * 6 = 1500$ as this is the first point, I see the matmul-transpose-parallel is above matmul-locality-parallel.

For the comparison I have run Matmul-locality and matmul-transpose-parallel for a fixed input size N=M=K=2500.
‘’Modi test’’



Here I see that my data follows the expected values for p=83. This means that I do have strong scaling in this case. If I zoom in and take a look at the first 8 threads, I get a speedup at almost 4 which is pretty decent strong scaling.



For matmul-transpose-parallel I see that I have weak-scalability. I get an almost linear graph which means that for an increasing input relative to number of threads It will run in about the same time.