CS433: Parallel Programming
Assignment III Report

Aditi Goyal - 190057
Yatharth Goswami - 191178

April 27, 2022

Contents

I Gauss-Seidel Solver

In this problem, we were asked to perform Gauss Seidel iterations on a square matrix using CUDA. In each iteration, we replace the value with the average of it's neigbours and itself. We do this till convergence (convergence set by tolerance and max iterations = 1000). Assume that the matrix A has size $n \times n$. We pad it to make the computations easier. Let the number of threads be P. Also, we have flattened out the matrix A. A global diff variable is maintained which contains the sum of local differences at each index before and after iteration. Also, in all the trials mentioned below (including the final one), we have 1 dimensional blocks with number of threads per block = 32. To successfully run the final attempt, one needs to make sure that number of threads provided are at least 32 and n^2/P is divisible by TILE SIZE (16 in our case)

1.1 Previous attempts

1.1.1 Divide n+2 × n+2 grid among threads

We divided the work of threads in $n+2 \times n+2$ grid in a block fashion i.e. the first $(n+2)*(n+2)/P$ to the first thread, next $(n+2)*(n+2)/P$ to the next thread and so on. We check if the current index that the thread is working on, is a padded one, or the inner one on which computation needs to be done. In case, the computation needs to be done, we update A. To update global diff, we atomically add local diff. Synchronization is done by sense reversing barrier.

Problems with this approach: Adding if branch in the code hampers the performance because threads in a warp execute in a lock step fashion. The thread which is assigned the first row sits idle because it is completely padded.

1.1.2 Divide n × n grid among threads

We assign threads in the inner $n \times n$ grid only. We just changes the offset indexing to incorporate the fact that we have padded. Row i column j becomes $n*i+j$ when flattened in normal $n \times n$ matrix. In this case, it would becomes $(n+2)*(i+1)+j+1$. To update global diff, we atomically add local diff. Synchronization is done by sense reversing barrier.

Problems with this approach The global diff is updated sequentially, it is done by atomic add. This also needed to be parallelized.

1.1.3 Divide n × n grid among threads + tree reduction

All other things are as the approach above. The only addition done is tree reduction for updating global diff.

Problems with this approach: The matrix A would be stored in L2 cache and below. Accessing any variable in A would take a lot of time. We would like to bring A to L1 cache and make the accesses by threads to shared memory. For this, we consider two cases:

1.2 Final attempt

In the final attempt we implement the shared memory version for improvement in performance with tree based reduction. There are two cases which we handled separately.

- Size of square matrix (n) is greater than nthreads (P). In this case, we assign each thread n/P rows to work on. Each thread works on it's assigned rows in a cyclic fashion. See below figure. In this case, thread id 1 operates over the first row blue rectangle first and then in it's next turn will move to operate on first row of the green rectangle (rows are assigned in cyclical fashion to threads). Now, consider the first thread block. The threads in a block compute their rows in a phase wise manner. The rows are tiled (TILE SIZE = 16 in our code). In the first phase, the threads operate on columns within 0 to TILE SIZE-1. For computation in this phase, they need the a rectangle sized (number of threads per block +2) × (tile size +2) as shown in the figure. Then, they compute on the columns TILE SIZE to 2 × TILE SIZE-1. Once all the threads in a block have computed their respective rows, they move on to the next set of rows assigned to them (green rectangle in the figure).

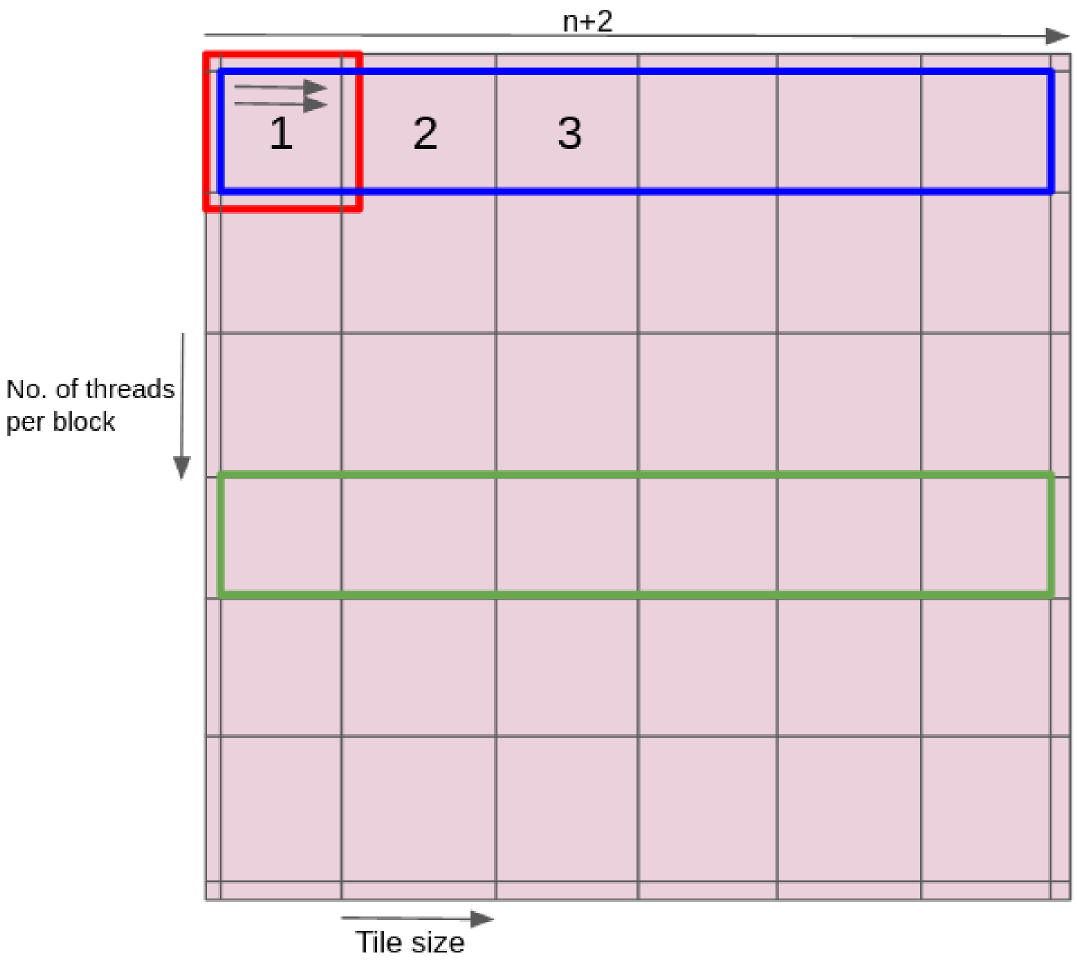

Figure 1: Gauss-Seidel shared memory with n greater than nthreads

- Size of matrix (n) is less than number of threads (P) but number of threads are bounded above by $n \times n$. In this case, each thread will operate on a section $(n \times n)/P$ of the matrix. Hence, in this case we divide the rows into pieces each of size $(n \times n)/P$. Each row of blue rectangle shown in the below shows one such piece. Each such piece is assigned to one thread. Each thread completes the task assigned to it in certain phases. In each phase, each thread brings a part of the row assigned to it in the shared memory for fast access first and then starts computing the new values for the next iteration. All the threads in a block work on rows in one of the bigger rectangles. In the figure, the first thread of the block will be working on first row of blue rectangle, second thread will be working on the row below it in blue rectangle and so on. Similar to the previous case, the rows are tiled (TILE SIZE = 16 in our code). In the first phase, the threads operate on columns within 0 to TILE SIZE-1. For computation in this phase, they need the a rectangle sized (number of threads per block +2) × (tile size +2) as shown in the figure. The threads on upper and lower boundaries also bring the row above and below them respectively into shared memory as well. Then, they compute on the columns TILE SIZE to 2 × TILE SIZE-1. This pattern repeats until all $(n \times n)/P$ elements are done by a thread.

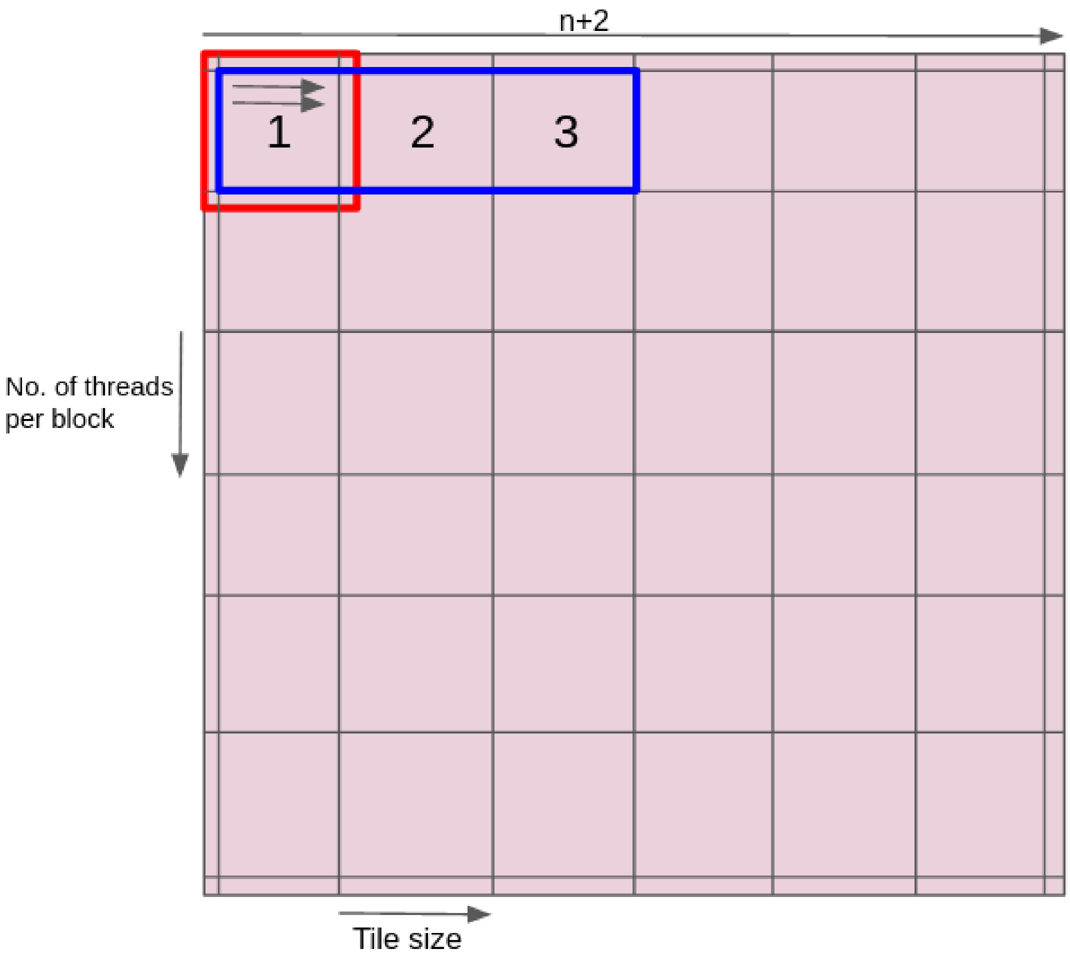

Figure 2: Gauss-Seidel shared memory with n less than nthreads

1.3 Performance Evaluation

The code was run on the cse machine gpu@cse.iitk.ac.in. Here is the overall experimental setup:

- The execution time is only taken for the kernel 'gauss_seidel_kernel' in the code.
- Fixed matrices initialised using 'init_kernel' were taken for testing and evaluation.
- The execution times are measured for thread count from 32 to 16384. Although we tried for bigger thread counts as well but it was taking much more time for evaluation of omp programs on our personal machine.
- The execution times are measured as average of 5 executions.

Here are the performance statistics for the program described above (and submitted as final submission) Note: the data taken below varied with the load on gpu. At some instance when we could not take the complete data, shared memory results were quite better than the simple version.
Execution times for $N = 1024$:

Number of Threads	Execution Time (in μs)
32	11894254
64	9359321
128	6922969
256	3934277
512	1094841
1024	396698
2048	253744
4096	211053
8192	318226
16384	492636

Here the best time was 211053 microseconds and 4096 thread count. On running it using tree reduction, we got 66142. On running it using our shared memory, it got reduced to 34026 microseconds.

Execution times for $N = 2048$:

Number of Threads	Execution Time (in μs)
32	159543469
64	80011419
128	44636138
256	24154594
512	13042267
1024	8066832
2048	4164467
4096	3587819
8192	2972586
16384	10233532

Here the best time was 2972586 microseconds and 8192 thread count. On running with tree reduction, we got 1680631 microseconds. On running it using our shared memory, it got reduced to 721775 microseconds.

Execution times for $N = 4096$:

Number of Threads	Execution Time (in μs)
32	600046270
64	331698303
128	154349309
256	76893749
512	40414211
1024	20984766
2048	11317690
4096	10979202
8192	12091804
16384	32648503

Here the best time was 10979202 microseconds and 4096 thread count. On running using tree reduction, we got 4592772 microseconds. On running it using our shared memory, it got reduced to 2611938 microseconds.

Execution times for $N = 8192$:

Number of Threads	Execution Time (in μs)
32	495882300
64	239968863
128	206909381
256	68959296
512	33355758
1024	18798857
2048	10222742
4096	6766725
8192	9282447
16384	29763202

Here the best time was 6766725 microseconds and 4096 thread count. On running it using tree reduction, we got time 2980417 microseconds. On running it using our shared memory, it got reduced to 2222667 microseconds.

Execution times for $N = 16384$:

Number of Threads	Execution Time (in μs)
32	1153785637
64	607697555
128	316902773
256	177559517
512	111850937
1024	142352797
2048	414157658
4096	27457699
8192	20702155
16384	71905334

Here the best time was 20702155 microseconds and 8192 thread count. On running it using tree reduction, we obtained 20819295 microseconds as the time. On running it using our shared memory, it got reduced to 10562892 microseconds.

Next, we visualize performance of our basic program.

Visualizing performance:

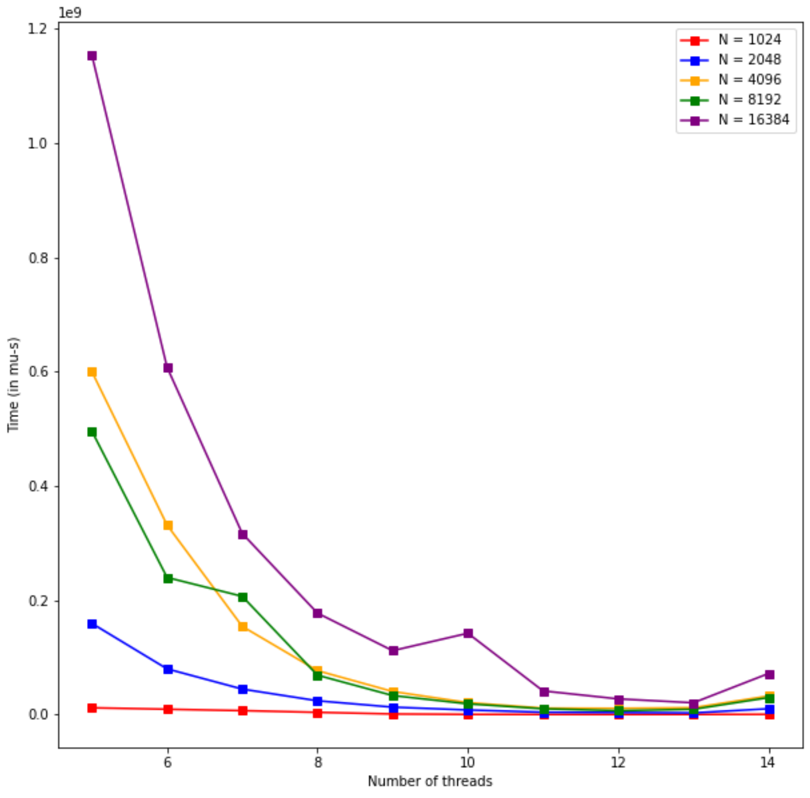

Figure 3: Time vs number of threads (logarithm scale)

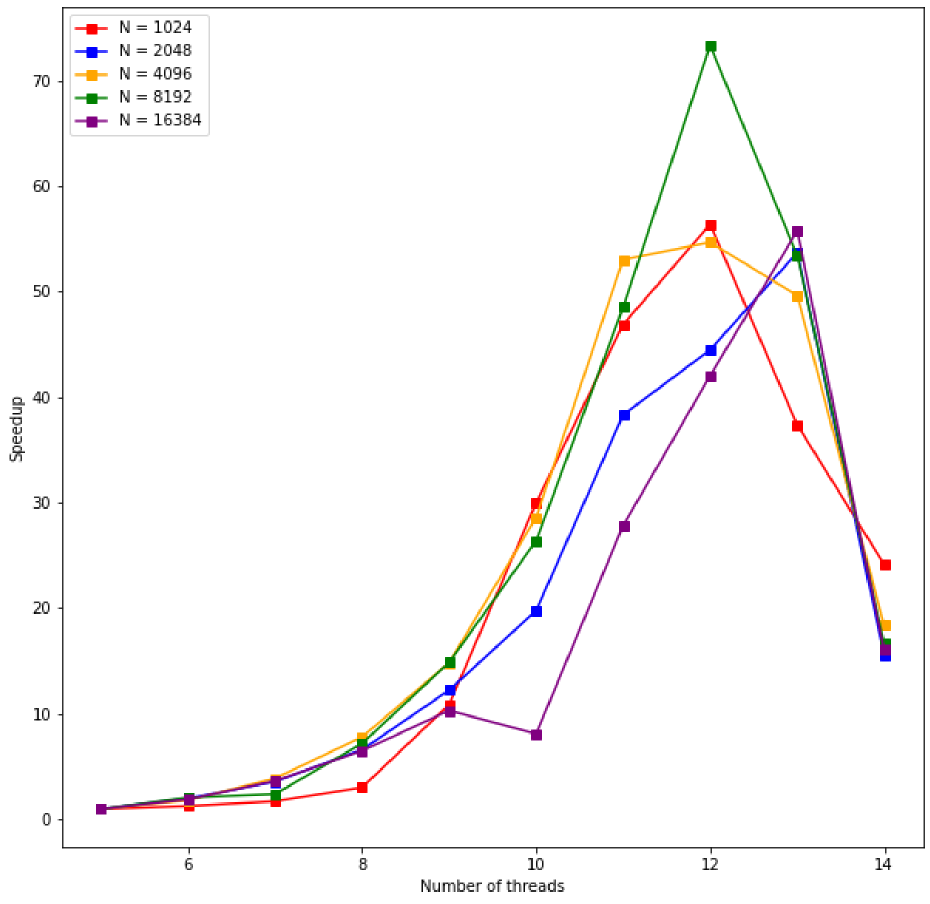

Figure 4: Speedup vs number of threads (logarithm scale)

1.4 Comparison with OMP

We compared our best times using shared memory implementation and OMP parallel program. Here is the summarised result

Size of matrix	Execution Time for CUDA (in μs)	Execution Time for OMP (in μs)
1024	34026	4258580
2048	721775	1584025
4096	2611938	6372769
8192	2222667	24588804
16384	10562892	102565264

Notice that from the above table, we observe that the performance of GPU programs are almost always better than OMP ones when amount of computation is large enough and algorithm is parallelised to run on GPU.

II Matrix-Vector Multiplication

In this problem, we were asked to perform Matrix Vector multiplication on a square matrix and column vector using CUDA. We are given an $n \times n$ matrix A and a column vector. We need to output column vector such that $y = Ax$. We have flattened out the matrix A. Here, we have kept number of threads (let us denote this by P) to be minimum of the number of threads given by user and the number of rows in matrix A i.e. n

2.1 Previous attempts

2.1.1 Assign threads to rows

The computation of first n/P rows of y is done by thread 1, next n/P rows of y by thread 2 and so on.

Problems with this approach: The matrix A and x accessed by threads will be in L2 cache. To make these accesses fast, we need to bring them to shared memory.

2.2 Final attempt

In this attempt, we tiled the rows of the matrix to implement shared memory. Each thread is assigned multiple rows in a cyclical fashion. A thread assigned row i computes $y[i]$. This computation proceeds in a phase wise manner. In the first phase, first TILE SIZE rows of x are multiplied with corresponding entries in A. In the next phase, next TILE SIZE rows of x are multiplied with corresponding entries in A. The phases of all threads in a thread block are synchronized to utilise shared memory. A rectangle of size number of threads per block $\times$ tile size is brought into shared memory by the threads operating on this tile (each thread brings the row it operates on). Once all the threads in a thread block are done, they move on to the next set of rows assigned to them. For example, in the figure below, the first phase of threads in the first thread block is depicted by the red square.

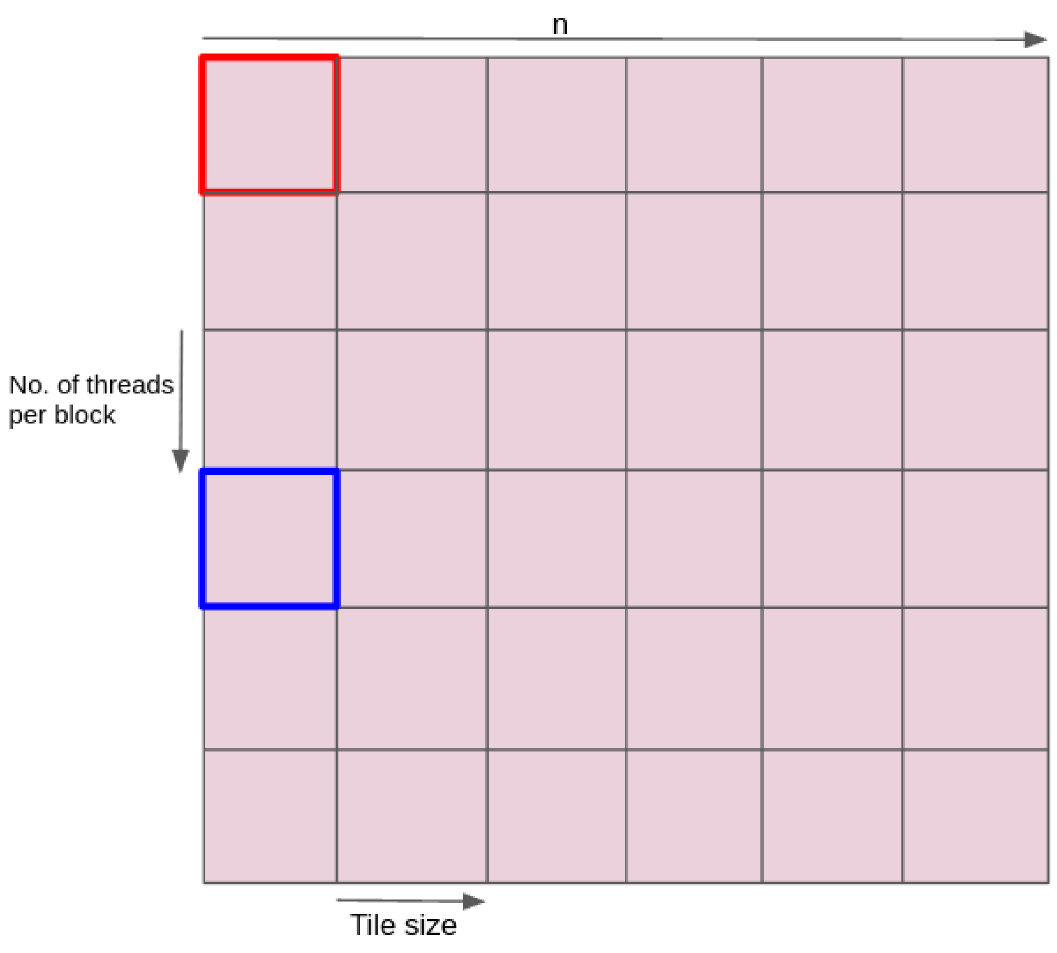

Figure 5: Matrix vector multiplication using shared memory

2.3 Performance Evaluation

The code was run on the cse machine gpu@cse.iitk.ac.in. Here is the overall experimental setup:

- The execution time is only taken for the kernel 'matmul_kernel' in the code.
- Fixed matrices initialised using 'init_kernel' were taken for testing and evaluation.
- The execution times are measured for thread count from 32 to 16384. Although we tried for bigger thread counts as well but it was taking much more time for evaluation of omp programs on our personal machine.
- The execution times are measured as average of 5 executions.

Note: The way we have parallelized the program allows calculation of matrix vector product with number of threads upto size of matrix only. Beyond this size, the program had to use reduction which was not supposed to be done in this problem. So, we capped the number of threads by size of matrix (n). Here are the performance statistics for the program described above (and submitted as final submission) Note: the data taken below varied with the load on gpu. At some

instance when we could not take the complete data, shared memory results were quite better than the simple version.
Execution times for $N = 1024$:

Number of Threads	Execution Time (in μs)
32	13173
64	5773
128	2979
256	1556
512	861
1024	466
2048	532
4096	552
8192	539
16384	522

Here the best time was 466 microseconds and 1024 thread count. On running it using our shared memory, it got reduced to 223 microseconds.

Execution times for $N = 2048$:

Number of Threads	Execution Time (in μs)
32	52134
64	23764
128	18227
256	6682
512	3423
1024	1603
2048	920
4096	1100
8192	1124
16384	970

Here the best time was 920 microseconds and 2048 thread count. On running it using our shared memory, it got reduced to 443 microseconds.

Execution times for $N = 4096$:

Number of Threads	Execution Time (in μs)
32	161741
64	78523
128	39937
256	21667
512	16427
1024	6850
2048	3186
4096	1934
8192	2250
16384	4976

Here the best time was 1934 microseconds and 4096 thread count. On running it using our shared memory, it got reduced to 1320 microseconds.

Execution times for $N = 8192$:

Number of Threads	Execution Time (in μs)
32	630218
64	315216
128	157667
256	81139
512	41858
1024	21853
2048	18237
4096	9776
8192	7128
16384	7370

Here the best time was 7128 microseconds and 8192 thread count. On running it using our shared memory, it got reduced to 6850 microseconds.

Execution times for $N = 16384$:

Number of Threads	Execution Time (in μs)
32	2720512
64	1433490
128	719598
256	361852
512	182170
1024	96703
2048	51878
4096	33602
8192	27730
16384	30401

Here the best time was 27730 microseconds and 8192 thread count. On running it using our shared memory, it got reduced to 25840 microseconds.

Next, we visualize performance of our basic program.

Visualizing performance:

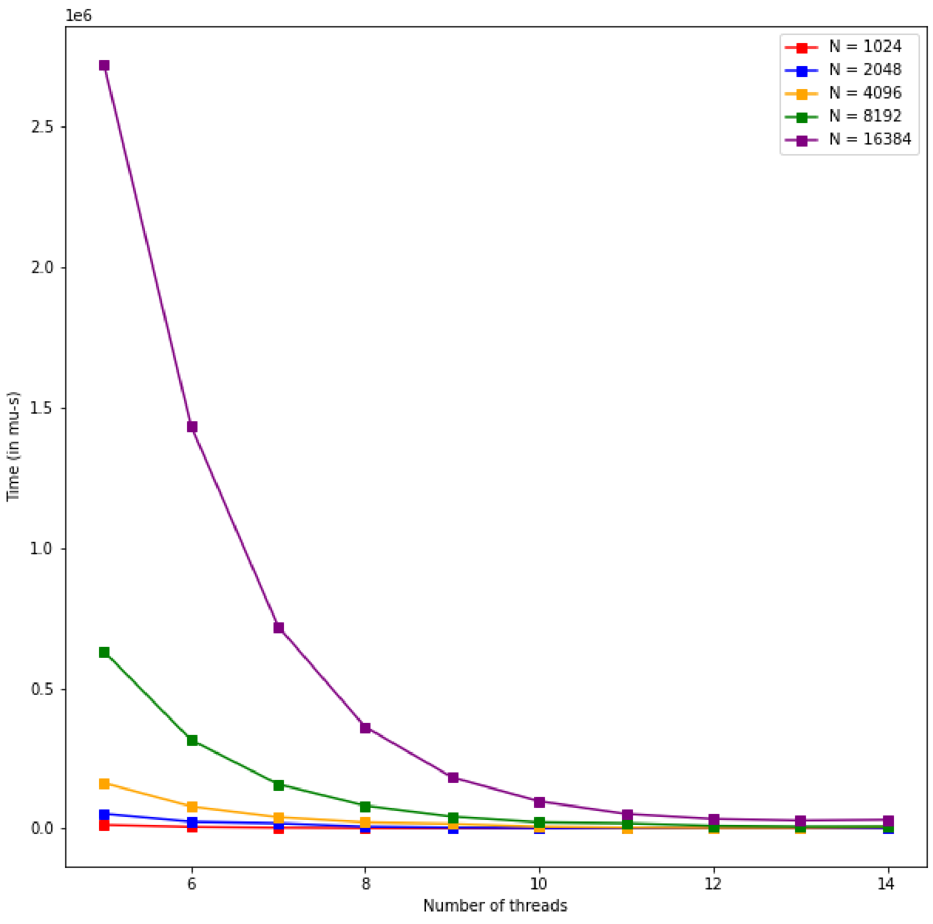

Figure 6: Time vs number of threads (logarithm scale)

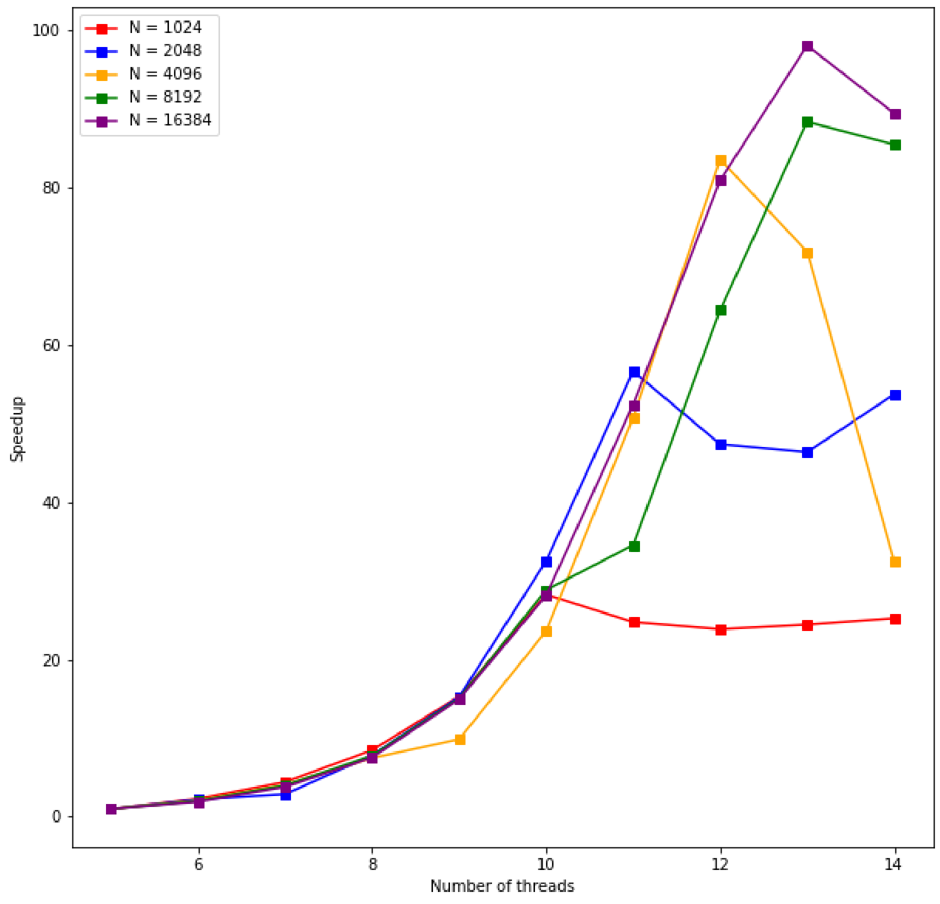

Figure 7: Speedup vs number of threads (logarithm scale)

2.4 Comparison with OMP

We compared our best times using shared memory implementation and OMP parallel program. Here is the summarised result

Size of matrix	Execution Time for CUDA (in μs)	Execution Time for OMP (in μs)
1024	223	180
2048	443	993
4096	1320	4660
8192	6850	17526
16384	25840	72084

Notice that from the above table, we observe that the performance of GPU programs over OMP programs improves significantly on increasing the number of threads.

III Compilation procedure

3.1 For Question 1

Compile with the following command 'nvcc -O3 -DFIX gauss_seidel_shared.cu -o gauss_seidel_shared' 'nvcc -O3 -DFIX gauss_seidel_tree_reduction.cu -o gauss_seidel_tree_reduction' 'nvcc -O3 -DFIX gauss_seidel_simple.cu -o gauss_seidel_simple' 'gcc -O3 -fopenmp omp_gauss-seidel_cyclicrow.c -o omp_gauss-seidel_shared'

Replace the Define with -DCUDA_RANDOM for providing a random initialization in cuda programs.

3.2 For Question 2

Compile with the following command 'nvcc -O3 matmul.cu -o matmul' 'nvcc -O3 -DFIX matmul_shared.cu -o matmul_shared' 'gcc -O3 -fopenmp omp_matmul -o omp_matmul'