JACOBI ITERATIVE SOLVER

STEP1

A serial CPU version of the Jacobi Iterative solver is developed, and the solver is run for the given number of iterations. The solution vector computed is compared against the expected vector and the absolute summation of the errors is measured. The criteria for the verification of the Jacobi Iterative solver (CPU / GPU code) is that the final error reported must be less than unity.

STEP2

The i-loop is parallelized by creating a simple GPU kernel that uses CUDA Unified Memory. Each k-iteration is implemented as a kernel launch. The CPU must synchronize with the Device before copying the intermediate solution vector from the Unified memory. The CPU then proceeds with the next k-iteration.

90% of the computation time is spent in executing the kernel. The rest 10% is utilized in copying the intermediate result back to the input for the next k-iteration.

STEP3

The CPU copies the initial matrix and vector into the device memory once. For each k-iteration, the CPU launches the solver kernel and then, does a device-device memory copy of the resultant vector (to be used for next iteration). Since the memory copies in a CUDA stream are synchronous, it does not begin until the kernel execution is complete. All the k-iterations are launched in the default stream.

Device-Device memory copy is several orders faster in data transfer than Device-Host memory copy followed by Host-Device memory copy. 99.80% of the computation time is used in kernel execution. Thus, Optimization Step3 outperformed Optimization Step2 for all Input Sizes and Block Sizes.

STEP4

Each k-iteration is launched in two different streams and asynchronous device-device memory copy is used. The activities within a CUDA stream itself are in-order but can be interleaved with the activities from different streams. In this way, the memory copies are overlapped with the kernel execution to improve performance. The default stream synchronizes the other streams.

As memory copy is interleaved with the kernel execution, the algorithm falls between Gauss-Siedel and Jacobi. As GPU is still underutilized, streams did not offer performance benefit for this algorithm.

STEP5

Memory copies after each k-iteration can be avoided by launching the solver kernel again with the pointers to the vectors X and Y switched and running only for half the original iterations.

Since GPU Device-Device memory copy is incredibly fast even for larger chunks of data, this optimization technique did not offer any performance benefit.

STEP6

This optimization is a hybrid and combines Step4 and Step5. This did not offer performance boost as expected from Step4 and Step5.

STEP7

It is observed that the vector X is frequently accessed from the global memory at least N^2 times. Thus, it motivates the use of shared memory for the vector X. However, this will require synchronization within the thread blocks for producing the correct result.

The thread divergence leads to serialization of the branches and increases the computation time of the kernel. For very large inputs (and large iterations), performance can be improved by reducing the thread divergence in the solver kernel. To avoid the thread divergence, the computation for the vector Y is slightly different than the prior implementations. This, along with the use of shared memory improves the performance by a significant margin.

Shared memory is 100x faster than global memory. Avoiding thread divergence saved significant computation time on larger Input Sizes and larger iterations. Thus, significant improvement was observed.

STEP8

It is observed that prior implementations do not utilize the compute and memory bandwidth of the GPU to full extent. All the threads and the thread blocks are assigned to the outer i-loop. To explore more parallelism, the thread blocks are assigned to the outer i-loop. The threads within the thread block are assigned to the inner j-loop. The idea behind this is two-folds. First, the threads within a thread block (also a warp) can now access adjacent locations in memory and hence, improve memory coalescing. This will lead to lesser accesses to the global memory and hence, improving the memory latency. Second, more parallelism as more thread blocks can be assigned to the outer i-loop and therefore, drastically cutting down on the computation time. Thread divergence within a thread block is avoided as much as possible, as in Step7.

There are 56 SMs in the GPU with each capable of supporting up to 2048 active threads or 32 thread blocks, 48KB of shared memory per thread block (64KB per SM). This information helped in tuning the kernel configuration to run 112 thread blocks with up to 1024 active threads/thread block. All SMs are fully utilized for the computation in this optimization.

Shared memory is also used for storing the partial product/sums. Threads within a thread block can now compute the partial product/sum in parallel. The partial sums are then reduced in parallel (conflict-free manner) in at most 10 ($\log_2(1024)$) steps. All thread blocks allocate up to 20KB of shared memory for the inputs and partial results. Hence, smaller block size may be resource limited on the shared memory capacity. It is to be noted that up to 40KB of shared memory (per SM) is used for larger inputs in the tuned kernel configuration. Hence, further increase in the number of thread blocks or block size may be limiting in both the shared memory capacity and active threads per SM.

The 5120 x 5120 Input Size takes about 8.7 hours to compute the solution vector in CPU. However, This Optimization Step run on GPU computed the solution vector in less than a minute. The following is the ranking of different optimization techniques employed based on the relative difficulty of the approach, starting with kernel switching (the easiest). *Kernel Switching* < *Shared Memory* < *CUDA Streams* < *Memory Coalescing* < *Avoiding Thread Divergence* < *Explore Parallel Algorithms (Sum Reduction)* < *Tuning Kernel Configuration for Maximum Occupancy.*

RESULTS

	2	3	4	5	6	7	8
256.txt	7.7	16.32	8.49	29.8	16.37	16.52	16.39
1024.txt	51.14	96.59	95.25	99.3	101.17	146.35	149.99
5120.txt	211.37	248.4	244.39	249.74	245.15	370.65	625.3

Table 1: Speedup over CPU for various Input Sizes and GPU Optimizations

	CPU (ms)	1024 (ms)
8.txt	0.012992	0.551776
256.txt	784.355	47.846401
1024.txt	82854.33	552.38086
5120.txt	31010578	49593.453

Table 2: Absolute Timing for Fastest Implementation (STEP8)

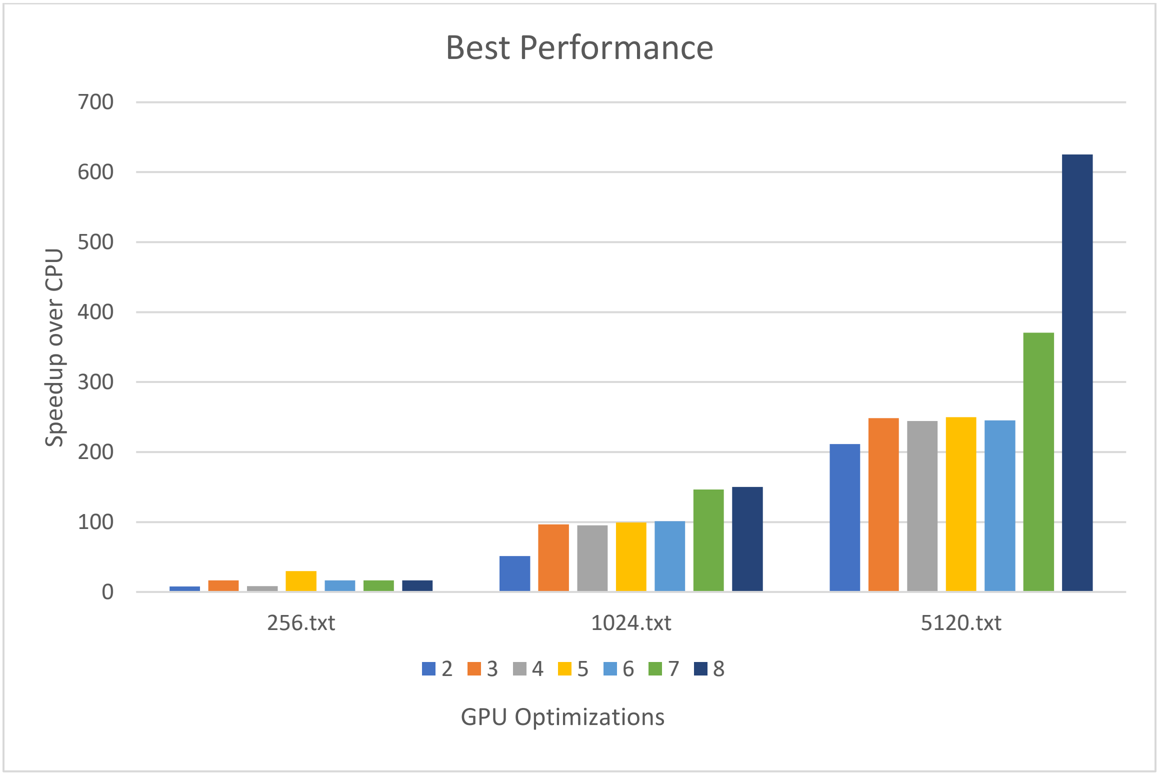

Figure 1: Speedup over CPU for various Input Sizes and GPU Optimizations

The speedup values in Table 1 indicate the best performance numbers obtained for the specific Input Size and Optimization Step from among the block sizes of 32, 64, 128 and 256. The fastest implementation employs shared memory, more thread-level parallelism, parallel computation of partial sums followed by conflict-free reduction, avoids thread divergence and improves memory coalescing. The Optimization Step8 run on GPU is 625 times faster than the CPU in computing the solution vector. It is also noted that speedup on GPU increases with the Input Size. In addition, block sizes of 32 and 64 offer better performance results than block sizes of 128 and 256 for most of the Optimization Steps.

APPENDIX

Input	32 (ms)	64 (ms)	128 (ms)	256 (ms)
8.txt	3.53936	3.149152	3.297184	3.06448
256.txt	128.2851	101.8702	108.2857	146.9553
1024.txt	2683.994	1792.665	1620.047	2611.85
5120.txt	148332.2	150716.3	146712.7	186918.1

Table 3: Absolute Timing for various Block Sizes (STEP2)

Input	32 (ms)	64 (ms)	128 (ms)	256 (ms)
8.txt	0.401408	0.402848	0.582592	0.403392
256.txt	48.0521	48.66746	48.58711	89.79495
1024.txt	857.7621	887.1097	1176.911	2043.731
5120.txt	124843.5	125720.1	125166.4	164374.9

Table 4: Absolute Timing for various Block Sizes (STEP3)

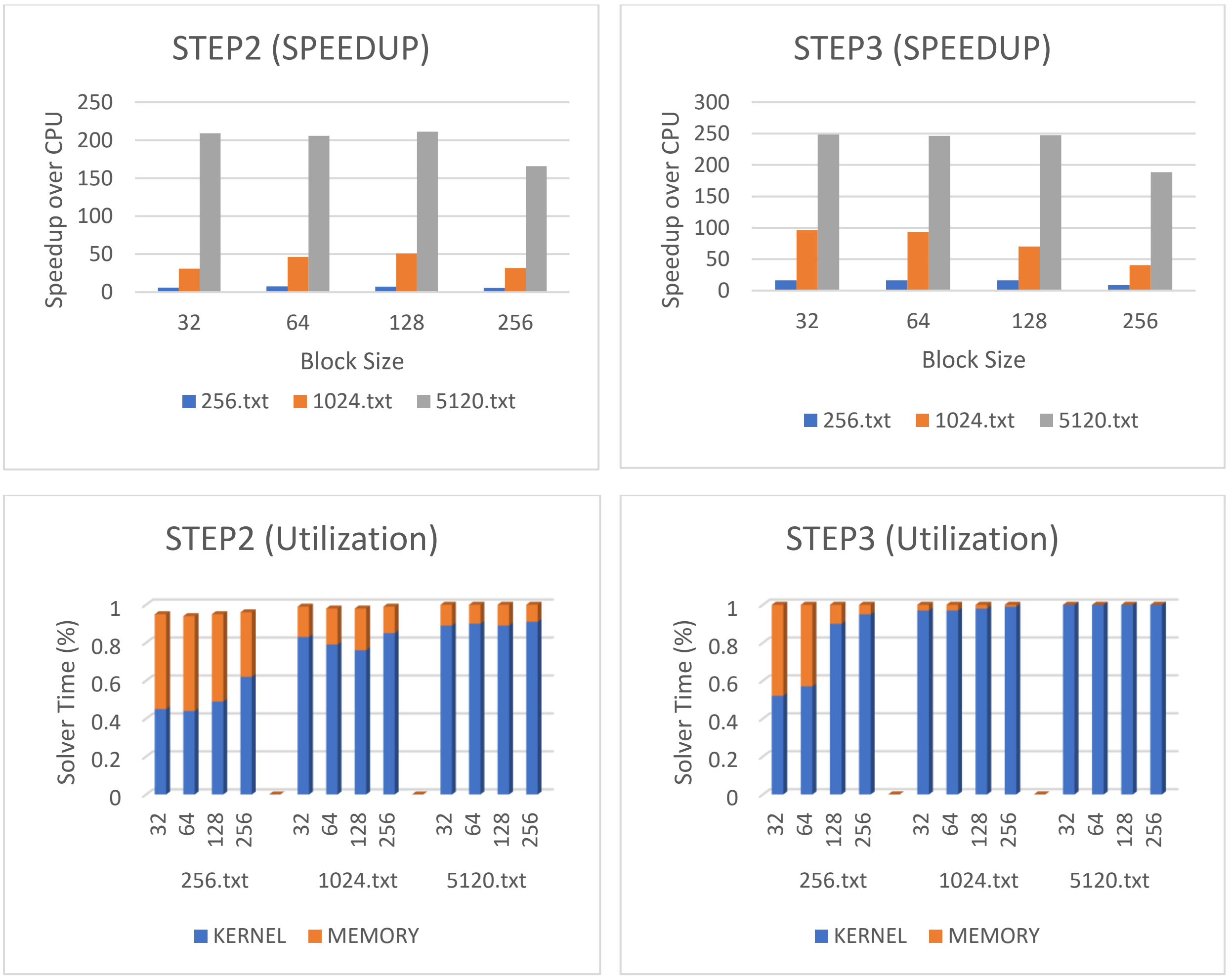

Figure 2 a) Speedup over CPU (STEP2), b) Speedup over CPU (STEP3), c) Solver Time Utilization (STEP2) and d) Solver Time Utilization (STEP3) for various Block sizes and Input sizes.

Input	32 (ms)	64 (ms)	128 (ms)	256 (ms)
8.txt	0.710976	0.718816	0.703968	0.707392
256.txt	93.734398	92.37523	93.07063	92.73725
1024.txt	869.828	901.0532	1183.86	2064.492
5120.txt	127138.05	128969.6	126887.2	165823.3

Table 5: Absolute Timing for various Block Sizes (STEP4)

Input	32 (ms)	64 (ms)	128 (ms)	256 (ms)
8.txt	0.421376	0.24336	0.239744	0.241824
256.txt	26.316608	28.51443	45.01331	86.09914
1024.txt	834.36487	858.931	1151.496	2029.731
5120.txt	124172.45	125342.8	124695	163906.9

Table 6: Absolute Timing for various Block Sizes (STEP5)

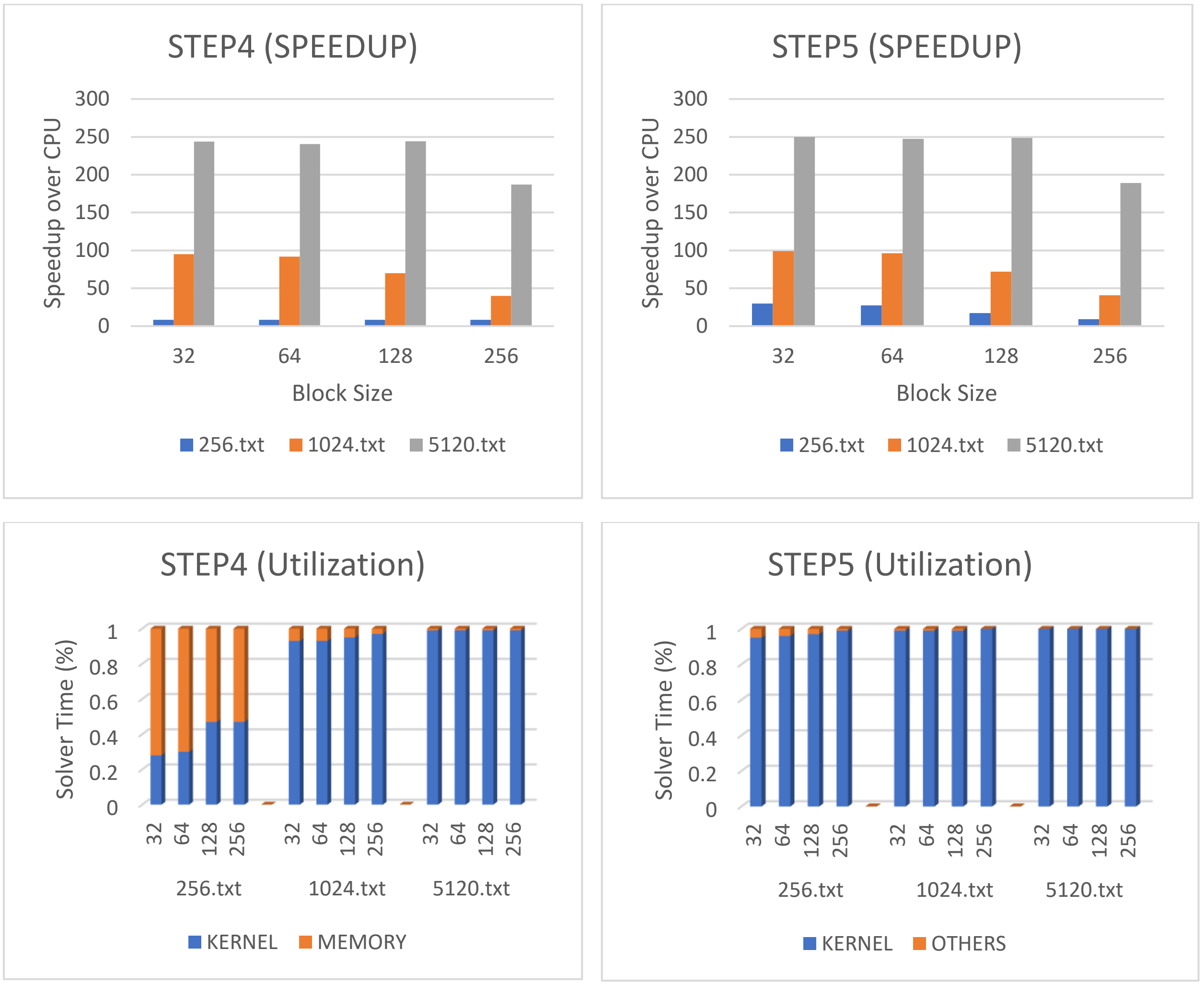

Figure 3 a) Speedup over CPU (STEP4), b) Speedup over CPU (STEP5), c) Solver Time Utilization (STEP4) and d) Solver Time Utilization (STEP5) for various Block sizes and Input sizes.

Input	32 (ms)	64 (ms)	128 (ms)	256 (ms)
8.txt	0.413856	0.408736	0.411456	0.411424
256.txt	47.927326	49.71091	48.43879	48.29952
1024.txt	818.95563	859.1746	1161.765	2027.173
5120.txt	127026.84	129792.2	126498.2	164122

Table 7: Absolute Timing for various Block Sizes (STEP6)

Input	32 (ms)	64 (ms)	128 (ms)	256 (ms)
8.txt	0.40112	0.399392	0.400416	0.402624
256.txt	47.596066	47.63773	47.49258	81.10544
1024.txt	579.96722	566.1398	962.451	1831.276
5120.txt	83665.914	96341.35	100372.4	214687.5

Table 8: Absolute Timing for various Block Sizes (STEP7)

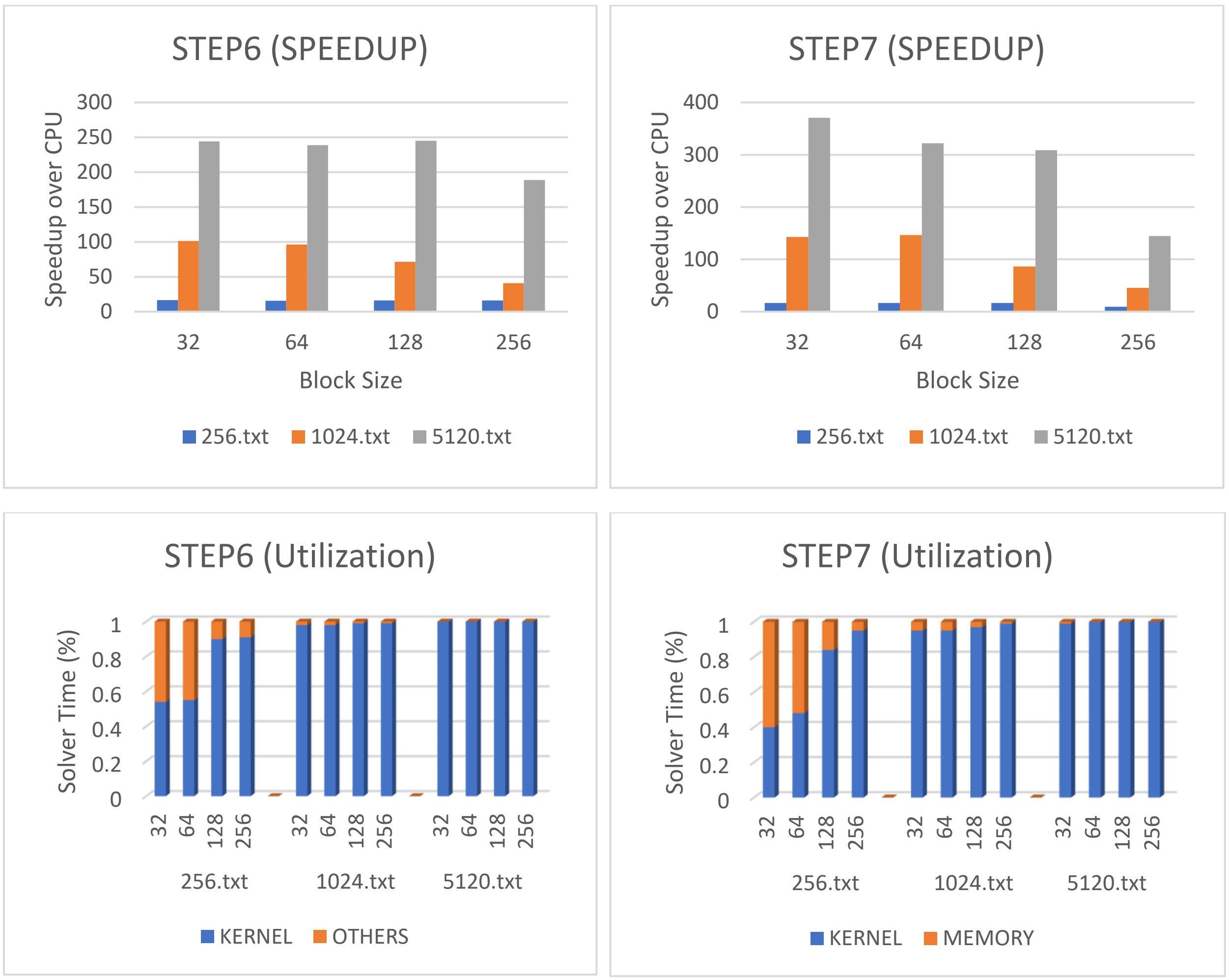

Figure 4 a) Speedup over CPU (STEP6), b) Speedup over CPU (STEP7), c) Solver Time Utilization (STEP6) and d) Solver Time Utilization (STEP7) for various Block sizes and Input sizes.