CS433
Design Assignment - 3

190772
Sarthak Rout
sarthakr@iitk.ac.in
Group - 21

April 27, 2022

Note: Machine specification is at the end of the report.

Q1: Gauss-Seidel Iterative Grid Solver

For grid-wise synchronization, we have used multiple kernel launches. n used: 4096

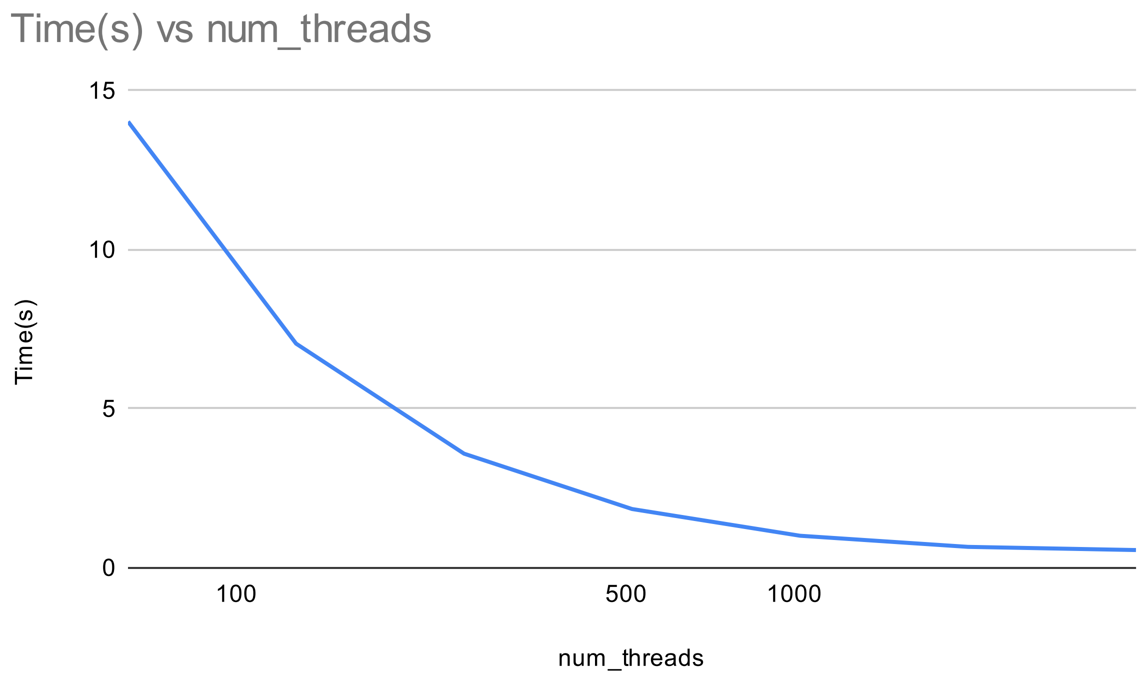

Figure 1: Time vs Number of Threads (Logarithmic Scale)

For N = 4096, performance of program for different thread counts:

num_threads	Time(s)
64	14.03
128	7.05
256	3.59
512	1.85
1024	1.01
2048	0.66
4096	0.56

Evaluating Optimizations with N = 4096, T = 4096:

- With naive solution with no optimizations – One atomic update of $diff$ variable per update per iteration : **7.94s**
- Implementing $local\_diff$ variable to accumulate local diffs and update at the end of the iteration: **1.29s**
- With tree reduction optimization to accumulate the diffs for threads in a warp in a tree-like fashion: **0.99s**

- With shared memory optimization: **0.56s**
- Comparing with OMP - Gauss Seidel (cyclic) implementation: **37.42s**

Final Algorithm:

- We generate 16K random values on the CPU in an managed array and copy them to the GPU to initialize the grid. We don't use the cudaRandom() function as it is expensive with time.
- The grid is allocated with help of cudaMallocManaged() function and we use cudaMemAdvise() to tell the CUDA APIs that we intend to use the grid on the GPU.
- Then, We initialize the grid on the GPU with the above random values. We do cudaDeviceSynchronize() and start the timer.
- We are asked to use the tolerance value as $1e-5$ and maximum number of iterations to 1000. In each iteration, we launch the upd kernel and synchronize all threads across the device. If the value of total $diff$ is greater than the given tolerance value, we continue or break.
- In the update loop, we create 2D tiles of size $(THREADS\_PER\_BLOCK + 2) * (TILE\_SIZE + 2)$ where THREADS_PER_BLOCK = TILE_SIZE is set to 16. These tiles are shared across threads. We load the grid into these tiles and synchronize threads with __syncthreads(). Then, we perform the update computations on the tile element and update the grid element simultaneously. On each update, we accumulate the local diff and at the end, use tree reduction to accumulate the local diffs and atomically add to the global diff across the device.
- After the update loop stops, we stop the timer. We report the total time taken, the number of iterations taken to converge and the average diff after the the algorithm stops.
- We currently only allow number of threads to be upper bounded by n.

Observations:

- We observe in Figure 1 that the with increase in t, we achieve massive speedup and improvement of program performance times.
- For some values of n and t, tree reduction didn't provide significant improvement over the normal atomic add operation.
- OMP program could only be run on small number of cores, due to which it couldn't provide the same level of performance as that of CUDA programs.
- It was surprising that loading the grid data into shared memory array could give a performance boost, as apparently, we are just doing extra load operations and earlier too we used to do local updates. But, the results show that software managed shared memory in L1 cache improves the performance by avoiding random cache evictions.

Q2: Matrix-Vector Product

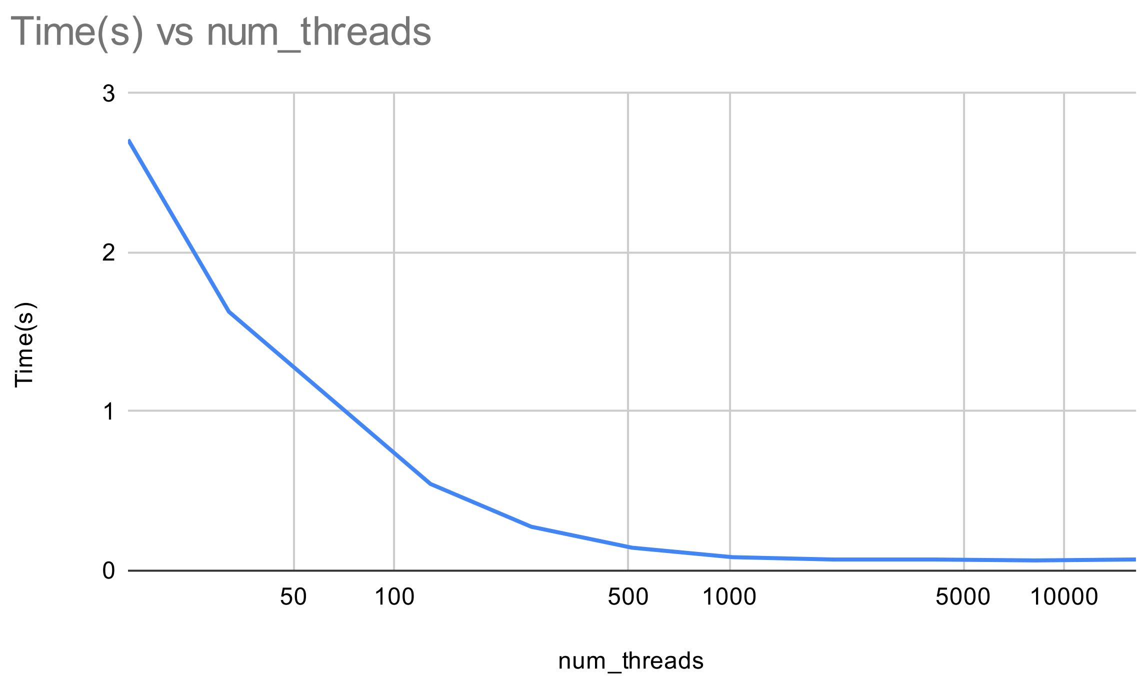

Figure 2: Time vs Number of Threads (Logarithmic Scale)

For N = 32768, performance of program for different thread counts:

num_threads	Time(s)
16	2.71
32	1.628
64	1.087
128	0.545
256	0.2759
512	0.144
1024	0.0843
2048	0.0697
4096	0.0697
8192	0.064
16384	0.07

Evaluating Optimizations for N = 32768:

- Without shared memory optimization, for 8192 threads, the program ran for 0.09s and for 4096 threads, the program ran for 0.12s.
- **Comparison with OMP:**

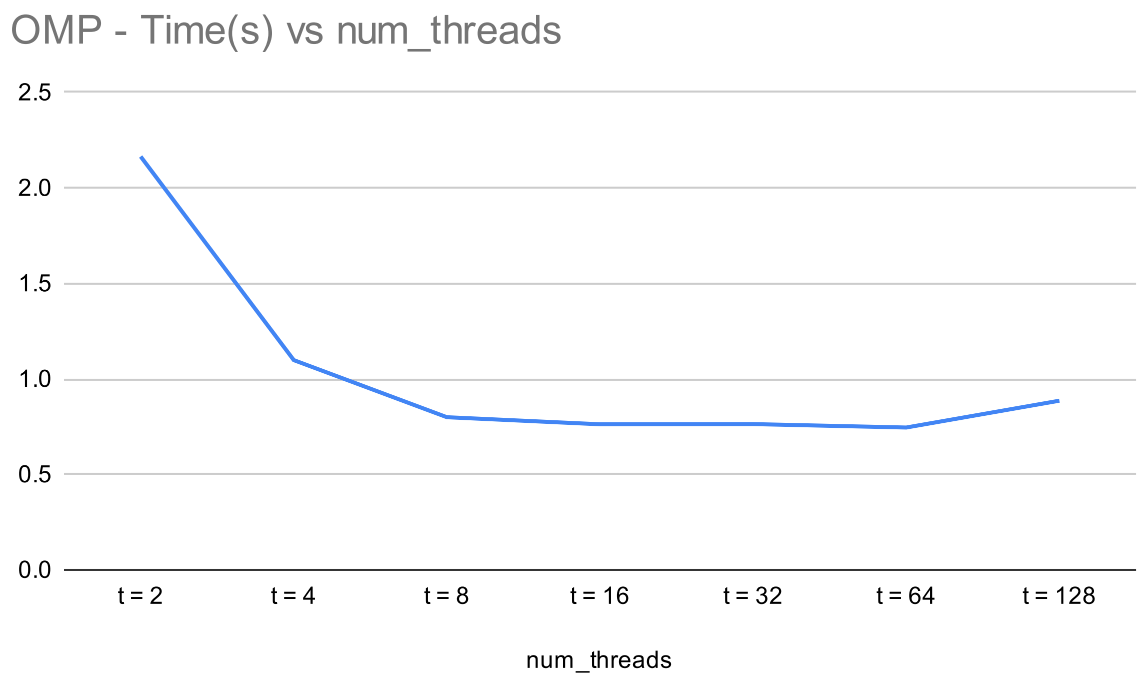

Figure 3: OMP : Time vs Number of Threads (Logarithmic Scale)

OMP, n = 32768	**Time(s)**
t = 2	**2.164**
t = 4	**1.098**
t = 8	**0.799**
t = 16	**0.762**
t = 32	**0.763**
t = 64	**0.745**
t = 128	**0.8855**

Final Algorithm:

- We initialize with a $init$ kernel, that gives some hard-coded values into the matrix and vector x. We avoided the cudaRandom() function, as it took large amount of time.
- We had allocated memory for the $matrix$, x and y arrays using cudaMallocManaged() and advised using cudaMemAdvise() that we intend to use the arrays on the GPU.
- In the $update$ kernel, we use the shared memory to store the values of x vector as it is being used by multiple threads as opposed to the matrix values which is only used by one corresponding thread. We store 64 values of the x vector.
- We load the value of x vector via each thread and do __syncthreads(). Then, we maintain a variable $party$ to accumulate the y vector element. After iterating over a row, we update the value of y. As we have taken $t < n$, we need not use atomicAdd to update the value of y. The $party$ variable is needed to avoid cache issues.
- The $matmul$ function finds out the difference between the two vectors after computing the y vector on the CPU.
- We currently implement the case where $t < n$.

Observations:

- We observe in Figure 2 that the with increase in t, the time taken is reduced due to the program being highly parallelisable.
- Sharing the memory for x vector improves the program performance, as we exploit that fact that multiple rows are multiplied with the same value of x. Infact each column of the matrix is multiplied with only one value of x.
- We could run the OMP program for small thread values, but the the programs have comparable performance with that of the CUDA programs.

Machine Details

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              12
On-line CPU(s) list: 0-11
Thread(s) per core:  2
Core(s) per socket:  6
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               79
Model name:          Intel(R) Core(TM) i7-6850K CPU @ 3.60GHz
Stepping:            1
CPU MHz:             3697.940
CPU max MHz:         4000.0000
CPU min MHz:         1200.0000
BogoMIPS:            7195.85
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            15360K
NUMA node0 CPU(s):   0-11
```