Assignment 2 COMP4300

Abhaas Goyal (u7145384)

June 4, 2021

Table 1: Table abbreviations

$\mathbf{A_T}$	Advection Time
$\mathbf{G_F}$	Gigaflops per second
$\mathbf{P_C}$	Per core (with G_F)
$\mathbf{N_T}$	Number of threads
NP	Number of processes
$\mathbf{C_T}$	Calculated time
$\mathbf{I_T}$	Instruction type

Kindly see the above table for table headings given later

1 Parallelization via 1D Decomposition and Simple Directives

1.1 Updation policies

1. `omp1dUpdateAdvectField()` - updated via the metrics described below

2. `omp1dCopyField()` - same as the best performant (1) from metrics. (Although loop fusion can be done to significantly reduce the load of copying in each iteration[2]) - even better optimization is done at the end of Q4 than the approach just mentioned, where `omp1dCopyField` is only called after the last iteration.

3. `omp1dBoundaryRegions()` - parallelized only the top and bottom halo exchange and not left and right exchange, because cache for only 2 elements in each row (leftmost and rightmost block) would be loaded for different threads so there will be a lot of read/write cache misses. Whereas in top and bottom exchange parallelizing would lead to only 2 rows of cache blocks (one for top and another for bottom) to load for each thread so performance would be worth it (for large values of `N`).

1.2 Metrics

Let `t = No of threads`

1. **Maximize performance** - (a) **Code**

```
#pragma omp parallel for default(shared) private(j)
  for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++)
      // Update V(v,i,j)
  }
```

Analysis

Best practice is to parallelize the outer loop (i.e. rows) - which significantly reduces overheads with minimal parallel regions being created. Also, parallelize the outer loop, while iterating through the columns privately gives us maximal performance (by minimizing cache reads/writes) and creates `t` separarate parallel regions (each of size around `(N * M)/t` blocks).

2. **Maximize parallel region entry/exits** `(b)` **Code**

```
for (i = 0; j < M; i++) {
  #pragma omp parallel for default(shared)
  for (j = 0; j < N; j++)
    // Update V(v,i,j)
}
```

Analysis

To create more parallel regions, we just parallelize the innermost loop (in this case j). In each iteration of i, we would be getting `t` parallel regions of size around `N/t`, so after all iterations, one would be getting around ration, I would be getting around `M * t` parallel regions, which is maximal. (This is assuming that `M >= N` - if the opposite is true then switch `i` and `j`, keep `i` in innermost loop and parallelise over `i`)

3. **Maximize cache misses involving coherent reads** `(c)`

Visualization

```
t   | 1 2 3 4 |
----+----------|
i   |          |
0   | *        |
1   |   *      |
2   |     *    |
3   |       *  |
4   | *        |
5   |   *      |
6   |     *    |
7   |       *  |
.   | *        |
.   |          |
n   |       *  |
    |----------|
   (inspired from [1])
```

Code

```
#pragma omp parallel for default(shared) private(j) schedule(static,1)
  for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++)
      // Update V(v,i,j)
  }
```

Analysis

- To maximize cache misses I applied a policy of statically scheduling parallel regions to 1 element.
- In this case loading at a thread in each iteration (in a round robin fashion) is done.
- For the distribution of various parts, here's a visualization. Assume no of threads are 4. Then we divide the threads in static blocks of size i, like- for every iteration, a new cache block for each new thread has to be loaded considering the size of each thread's row. Although once it has been loaded, writing becames faster - so can't really use this in metric 4.
- To implement this, one does the usual of parallelizing over outer loops but add `schedule(static,1)` to it.

4. **Maximize cache misses involving coherent writes** `(d)`

   Visualization

```
j |  0 1 2 3 4 5 6 7 .   .    .   .   .   n
--+-----------------------------------------
t |
1 |  *       *       *       *       *
2 |    *       *       *       *       *
3 |      *       *       *       *       *
4 |        *       *       *       *       *
```

Code

```
#pragma omp parallel for default(shared) private(i) schedule(static,1)
for (j = 0; j < N; j++) {
  for (i = 0; i < M; i++)
    // Update V(v,i,j)
}
```

Analysis

- j is parallelized so in each iteration while reading it's cool because on the left and right sides `a[i-1] and a[i + 1]` the cache line would already have been loaded but the problem happens when writing data.
- Shared data on left and right side for each thread's update - ie `a[i-1] amd a[i + 1]` is being accessed by multiple threads at the same time. The modifications of the same cache happen in rapid succession by different threads. All of these conditions lead to what's called false sharing, where each update will cause the cache line to "ping-pong between the threads" (idea gained from [2] slide 35 and seen lecture slides for more details)

1.3 Testing Methodolgy

- Performance model was tested in one node through strong scaling on `OMP_NUM_THREADS` for all metrics.
- The parameters were chosen to distribute all the nodes equally so M and N were both `divisible by 48`. The focus was on computation aspect so suitably large value of M and N were chosen. So for testing purposes I chose `M=N=2160` and remain unchanged since strong scaling was needed.
- Number of reps was taken to be 100 (sufficiently large)

1.4 Results

`M = 2160 N = 2160 reps = 100`

Table 2: Strong scaling with different number of threads on single node on different metrics

Metric	$\mathbf{N_T}$	1	3	6	12	24	48
(a)	$\mathbf{A_T}$	1.93e+00s	7.20e-01s	4.32e-01s	3.34e-01s	4.95e-01s	2.52e+00s
	$\mathbf{G_F}$	4.83e+00	1.30e+01	2.16e+01	2.79e+01	1.89e+01	3.71e+00
	$\mathbf{P_C}$	4.83e+00	4.32e+00	3.60e+00	2.33e+00	7.86e-01	7.72e-02
	Speedup	1	2.68	4.66	5.77	3.89	0.76
(b)	$\mathbf{A_T}$	5.84e+00s	1.86e+00s	1.14e+00s	1.02e+00s	3.57e+01s	4.73e+00s
	$\mathbf{G_F}$	1.60e+00	5.02e+00	8.18e+00	9.19e+00	2.61e-01	1.97e+00
	$\mathbf{P_C}$	1.60e+00	1.67e+00	1.36e+00	7.66e-01	1.09e-02	4.11e-02
	Speedup	1	3.13	5.12	5.72	1.63	1.2
(c)	$\mathbf{A_T}$	1.93e+00s	7.41e-01s	4.58e-01s	3.99e-01s	5.30e-01s	2.72e+00s
	$\mathbf{G_F}$	4.84e+00	1.26e+01	2.04e+01	2.34e+01	1.76e+01	3.43e+00
	$\mathbf{P_C}$	4.84e+00	4.20e+00	3.39e+00	1.95e+00	7.34e-01	7.14e-02
	Speedup	1	2.60	4.21	5.77	3.64	0.74
(d)	$\mathbf{A_T}$	5.78e+00s	3.38e+00s	1.62e+00s	1.25e+00s	3.61e+01s	0
	$\mathbf{G_F}$	1.62e+00	2.76e+00	5.76e+00	7.46e+00	2.59e-01	0
	$\mathbf{P_C}$	1.62e+00	9.20e-01	9.60e-01	6.22e-01	1.08e-02	0
	Speedup	1	1.71	3.56	4.62	0.16	0

- In (d) I have inputted 0 since it the operation was timed out (Walltime used was 1 minute)
- Metric (a) and (c) perform similarly and the same holds for metric (b) and (d)
- Metric (a) and (c) perform significantly faster than (b) and (d), which little differences within them
- That would be indicative of the fact that when a simple OpenMP policy is applied, optimizing it becomes less worth it.
- After `OMP_NUM_THREADS > 12` we see a heavy performance drop in most cases (since `numactl` has a limit of 12)
- So best version would use 12 threads (note for the future questions)

2 Performance Modelling of Shared Memory Programs

2.1 TODO

3 Parallelization via 2D Decomposition and an Extended Parallel Region

3.1 Approach

- To work on a single parallel region first, I calculated the starting and ending indices of each thread and then manually assigned different loop indices to different threads. Then I used those indices to work with `omp1dUpdateAdvectField()`, `omp1dCopyField()`, `omp1dBoundaryRegions()` and `updateBoundary`. Finally, I also used the result of `OMP_NUM_THREADS=12` from previous question to use in this one.

- To work only in one boundary at a time in a single parallel region at once - this time I also parallelized the left and right halo exchange (at the cost of potential cache misses - although the indices are already loaded and top and bottom halo can have slight load imbalance)
- Generally, no barrier is needed after left and right halo exchange in 2D grids, however when `(Q=1 and P > 1)` 1D case it is needed, so I added an edge case of a conditional barrier in the extreme case of `Q=1`

3.2 Results

- `M = N = 2160 (2 * L_3 cache has around 70 MB memory) OMP_NUM_THREADS=12`

Table 3: Computation for 2D process grids (1 Node) with Q >= 1

P	**Q**	$\mathbf{A_T}$	$\mathbf{G_F}$	$\mathbf{P_C}$	**Speedup**
1	12	4.48e-01	2.08e+01	1.74e+00	0.72
2	6	3.75e-01	2.49e+01	2.07e+00	0.86
3	4	3.57e-02	2.62e+01	2.18e+00	0.91
4	3	3.48e-01	2.68e+01	2.24e+00	0.93
6	2	3.40e-01	1.74e+01	2.28e+00	0.95
12	1	3.27e-01	2.86e+01	2.38e+00	1

- A significant decrease of of minimum 9-11% was found when distributing evenly across columns
- Distributing in row fashion after threads have been initialized still gives us the maximum speedup. This would be because each thread gets maximum cache reads
- Here, speedup is taken w.r.t Q = 1 unlike in previous table

4 Further Optimizations OpenMP

- One of the main optimizations was that The copying overhead is removed for larger number of iterations. (inspired from Piazza post of potential optimizations in assignment 1)
- This Improved performance by a lot

4.1 Results

- On `P = Q = 2160` with division of `P,Q 12,1` - we see a speedup of around 50%

Table 4: Results on optimized version of OpenMP

M	**N**	$\mathbf{A_T}$	$\mathbf{G_F}$	$\mathbf{P_C}$	$\mathbf{A_T}$ **(-o)**	$\mathbf{G_F}$ **(-o)**	$\mathbf{P_C}$ **(-o)**	**Speedup**
1080	1080	4.88e-02	4.78e+01	3.98e+00	5.34e-02	4.37e+01	3.64e+00	0.91
2160	2160	3.29e-01	2.84e+01	2.36e+00	1.86e-01	5.03e+01	4.19e+00	1.77
4320	4320	1.59e+00	2.35e+01	1.96e+00	8.08e-01	4.62e+01	3.85e+00	1.63

- We see that in the optimized version the per core speed pretty much remains the same (whereas in unoptimized version the per core speed decreases - because of more writing in memory in each iteration).

5 Baseline GPU Implementation

5.1 Testing methodology

- Worked with first copying data into GPU main memory and copying back to host memory when operation is completed (to maximize bandwidth)
- In general, GPUs work well with large sizes (and not small ones). So we take large parameters in this case `M=N=4096` (divisible by `32 * 32` - warp size)
- Varying block sizes with 2D warp size of 1024 (that means `Bx * By = 1024`)
- For `Gx and Gy`, it has to be in the powers of 2 (maximum even block distribution) and more number of blocks would be needed. To see the increase in varying block sizes, `Gx and Gy` start from 1,1 and go all the way to 64,64
- We also take `r=10` to not slow down the tests for small values of `Gx and Gy` (at the same time also hiding the time to copy back and forth from memory - as tested in nvprof)

5.2 Results

`M = 4096 N = 4096 reps = 10`

- Gx and Gy start to make sense to apply after `(Gx, Gy) >= 32`
- 3 variants of `(Bx,By)` make the most sense here consistently after `Gx` and `Gy` are decided - (8,128) (32,32) and (128,8)
- The best performing variant for this particular case would be `(Gx, Gy, Bx, By) = (32,32,8,128)`. However, for the more general case - `(Gx,Gy) = (32,32)` gave the best results

5.3 Comparing with CPU and serial GPU

- Finally, comparing with the results of CPU and serial GPU implementation at this speed

$\mathbf{I_T}$	$\mathbf{A_T}$	$\mathbf{G_F}$	**Speedup**
-s	5.05e+01s	6.64e-02	760.54
-h	1.09e+00s	3.09e+00	28.47

- We see a huge speedup from serial implementation of the GPU (around 750x)
- We also see around 30x increase from the initial CPU variant.

5.4 Kernel overhead

- I saw that the best way to determine kernel overhead would be to minimize the time within the function being actually called and at the same time it should be possible to invoke the kernel itself, so I called `M = 1, N = 1 r=100`. Note that I used a good value for `r` to give consistent results
- The result was 1.63e-03s in advection time for 100 repetions - which means around `1.6us` is spent for each iteration
- There are 4 kernels being called, so on average each kernel takes `0.4us` to load in each iteration.

Table 5: Performance comparision between various values of (Gx, Gy) and (Bx, By)

Gx	**Gy**	**Bx**	**By**	$\mathbf{A_T}$	$\mathbf{G_F}$	**Speedup**
1	1	16	64	1.46e+00s	2.30e+00	1.02
1	1	32	32	1.50e+00s	2.24e+00	1 (ref.)
1	1	64	16	1.55e+00s	2.17e+00	0.96
2	2	16	64	3.97e-01s	8.46e+00	3.77
2	2	32	32	4.00e-01s	8.39e+00	3.74
2	2	64	16	4.36e-01s	7.70e+00	3.43
4	4	16	64	2.14e-01s	1.57e+01	7
4	4	32	32	2.80e-01s	1.20e+01	5.35
4	4	64	16	2.87e-01s	1.17e+01	5.22
8	8	8	128	9.28e-02s	3.62e+01	16.16
8	8	16	64	2.90e-01s	1.16e+01	5.17
8	8	32	32	5.80e-01s	5.78e+00	2.58
8	8	64	16	6.39e-01s	5.25e+00	2.34
16	16	4	256	2.67e-02s	1.26e+01	5.625
16	16	8	128	4.04e-02s	8.31e+01	37.09
16	16	16	64	1.48e-01s	2.27e+01	10.13
16	16	32	32	4.99e-01s	6.73e+00	3.00
16	16	64	16	6.26e-01s	5.36e+00	2.39
16	16	128	8	5.88e-01s	5.71e+00	2.54
16	16	256	4	4.11e-01s	8.16e+00	3.64
32	32	1	1024	1.72e+00s	1.95e+00	0.87
32	32	2	512	9.70e-01s	3.46e+00	1.54
32	32	4	256	5.15e-01s	6.52e+00	2.91
32	32	8	128	3.81e-02s	8.80e+01	39.28
32	32	16	64	5.35e-02s	6.27e+01	2.79
32	32	32	32	2.14e-01s	1.57e+01	7.00
32	32	64	16	3.81e-01s	8.81e+00	3.93
32	32	128	8	3.69e-01s	9.08e+00	4.05
32	32	256	4	7.33e-01s	4.58e+00	2.04
32	32	512	2	9.78e-01s	3.43e+00	1.53
32	32	1024	1	1.59e+00s	2.12e+00	0.94
64	64	16	64	4.72e-02s	7.11e+01	31.74
64	64	32	32	3.84e-02s	8.75e+01	39.06
64	64	64	16	4.83e-02s	6.95e+01	31.02

5.5 Other optimizations

- Use it in conjunction with MPI (in which MPI is used in communication and OpenMP is used in Computation)

6 Optimized GPU Implementation

6.1 Implementation

- I used the concept of a custom tiled stencil, where each tiles is of size `(Bx,By)`. Each thread is mapped to one location of the tiled memory during one iteration of i and j (padding is also done for the cases of uneven block distribution not being divisible by `Bx or By` - intuition given in [3]). There are 3 major steps to this algorithm:

  1. Load into `__shared__` memory the answer of applying a part of the stencil on 3 contiguous memory addresses by all threads in a single block at one time. The total reads from each memory address would be <=3 (in the best case 1) since contiguous memory is being loaded so most accessed memory would be in each block's register (just a hypothesis).

     ```
     V_(aData, By, 1 + tdx, tdy) = cjm1 * V(u, tp_i, tp_j - 1) +
                                   cj0 * V(u, tp_i, tp_j) +
                                   cjp1 * V(u, tp_i, tp_j + 1);
     ```

     Corner cases of top and bottom rows would also need to be handled for correctly doing step 2.

  2. Since `__shared__` memory is scrachpad memory, accessing every element in it takes equal time. Do the remaining operation of accessing (j-1), j and (j+1) element - which already contains the weighted sum of 3 contiguous elements, and reuse it to do the remaining operation.

     ```
     V_(bData, By, tdx, tdy) = cim1 * V_(aData, By, tdx, tdy) +
                               ci0 * V_(aData, By, tdx + 1, tdy) +
                               cip1 * V_(aData, By, tdx + 2, tdy);
     ```

     (a) Finally, copy back the result in Cuda memory and do the same process in the next iteration

- Similar to OpenMP's optimized implementation, another optimization was inspired from Piazza, where I removed the operation of copying back elements to `u` in each iteration. (pretty nice optimization as seen by the results given below)

6.2 Results

`M = N = 2048 Gx = Gy = Bx = By = 32 reps = 100`

Table 6: Performance for different field sizes in baseline vs optimized version in CUDA

M	N	A_T	G_F	A_T (-o)	G_F	Speedup
2048	2048	1.21e-02s	6.91e+01	1.72e-02	4.87e+01	0.7
4096	4096	3.81e-02s	8.80e+01	4.16e-02	8.06e+02	0.91
8192	8192	1.41e-01s	9.54e+01	1.34e-01	1.00e+02	1.05

- This difference would be propounded by a huge factor if repetitions are large (since I have removed `copyFieldKP` to be done at the last, is only done in GPU memory anymore), the above tests are biased towards `updateAdvectFieldOPN`

- When number of iterations are high, then I gained an efficiency of 70-80% through `copyFieldKP` and `5-10%` through `updateAdvectFieldOPN` (tested using `nvprof`).

7 Comparison of various Programming Models

7.1 Performance

- CUDA's model works better for large size inputs (M, N> 4096) - that means there are limits to bandwidth in the CPU for large scale inputs and highly parallel architecture is best for this problem statement.
- MPI's model works best in smaller range and works in-between in heavy cases (better than OpenMP but worse than CUDA as mentioned)
- OpenMP's model works the worst (since minimum parallelism to a embarissingly parallel problem is being applied here) - even though the CPUs are fast
- The performance metric is different for CUDA and OpenMP vs MPI's model, since the focus of the MPI model relied on efficient communication rather than computation (`updateBoundary`), so there was a huge focus on exchanging halo boundaries. It's the opposite case for OpenMP and CUDA we wanted to efficiently compute the 9 point stencil update in `updateAdvectField` instead.

7.2 Difficulty in implementation (personal views)

- Doing the initial version of CUDA was easy/moderate, however finding potential optimizations and applying it was the hardest to do out of all. Mapping the threads to the correct locations in the tile, finding those optimizations and doing padding for uneven sized inputs was the hardest to do.
- Creating MPI's distributed system was second hardest (similar difficulty for all variants - 1D, 2D, overlapping and wide halo) - once you get the right frame of mind of what exactly to apply and what section to communicate, the corresponding implementation is not that hard.
- OpenMP was easiest to do but it didn't gave the worst results out of the three - with little room on optimization (from their initial policy)

8 CUDA Results on Gadi GPU's

- GPU used in Gadi- `1 x Tesla V100-SXM2-32GB (omg)`
- Testing methodology was to vary M and N with different values and compare the speedup with similar performing version of RTX2080

8.1 Results

`M = N = reps = 100 -o on Tesla V100-SXM2-32GB (omg)`

Table 7: Performance for optimized parameters in Nvidia Tesla

M	N	A_T	G_F	Speedup wrt RTX2080
2048	2048	7.52e-03	1.12e+02	1.62
4096	4096	2.21e-02	1.52e+02	1.72
8192	8192	8.32e-02	1.61e+02	1.68

Around consistent 70% improvement on a single GPU of Tesla. Since the size of Device memory on Tesla is pretty high (32GB), multiple copies of it can be used in larger scales than RTX variant

9 References

[1] `http://jakascorner.com/blog/2016/06/omp-for-scheduling.html`
[2] `http://akira.ruc.dk/~keld/teaching/IPDC_f10/Slides/pdf4x/4_Performance.4x.pdf`
[3] `https://piazza.com/class/kkeyidkqw3h21i?cid=187`

10 Acknowledgements

- Done individually and with heavy help from Piazza/COMP4300 Practicals/Piazza.