MAP55616-02 - Cylindrical Radiator

Finite Differences model

Yu Liu（liuy41@tcd.ie）
Git Repository Link: https://github.com/afkbjb/cuda2_cylindrical_radiator

Task 1 - calculation in the cpu

The CPU version simulates heat propagation on a 2D matrix over p iterations.
Two functions were implemented:

- `propagate_step`: updates the matrix for one time step, considering periodic boundary conditions.
- `compute_averages`: calculates the average temperature of each row.

A double-buffering approach was used (two matrices swapping roles) to avoid overwriting data during updates.

```
liuy41@cuda01:~$ ./radiator_cpu
Completed 10 iterations on 32x32 matrix.
Overall average temperature: 0.179616
liuy41@cuda01:~$
```

```
liuy41@cuda01:~$ ./radiator_cpu -n 10 -m 10 -p 20 -a
Row averages:
  row 0: 0.00913908
  row 1: 0.0365563
  row 2: 0.0822517
  row 3: 0.146225
  row 4: 0.228477
  row 5: 0.329007
  row 6: 0.447815
  row 7: 0.584901
  row 8: 0.740265
  row 9: 0.913908
liuy41@cuda01:~$ ./radiator_cpu -n 5 -m 5 -p 5 -a
Row averages:
  row 0: 0.032884
  row 1: 0.131536
  row 2: 0.295956
  row 3: 0.526144
  row 4: 0.8221
```

Task 2 - parallel implementation

The GPU version follows the same main steps as the CPU version: matrix initialization, p-step propagation, and row average calculation. The default block size is 16x16.

Two CUDA kernels were used:

- `propagate_kernel`: one thread updates one matrix element with periodic boundaries.
- `average_kernel`: one thread computes one row's average.

Memory copies and synchronization were managed properly. CUDA events measured GPU allocation, transfer, and kernel times. Options `-c` and `-t` were added.

Result

```
liuy41@cuda01:~$ ./radiator_gpu -c -t -n 64 -m 64 -p 20
GPU alloc: 0.085056 ms
GPU H2D: 0.030176 ms
GPU propagate: 0.321472 ms
GPU average: 0.022912 ms
GPU D2H: 0.046848 ms
liuy41@cuda01:~$ ./radiator_gpu -t -n 64 -m 64 -p 20
CPU compute: 0.863016 ms
GPU alloc: 0.086016 ms
GPU H2D: 0.026624 ms
GPU propagate: 0.286112 ms
GPU average: 0.030592 ms
GPU D2H: 0.047648 ms
Matrix mismatches (>1e-4): 0, max diff: 4.17233e-07
Average mismatches (>1e-4): 0, max diff: 1.78814e-07
Speedup: 2.72499
liuy41@cuda01:~$ ./radiator_gpu -t -n 32 -m 32 -p 20
CPU compute: 0.206836 ms
GPU alloc: 0.092128 ms
GPU H2D: 0.019712 ms
GPU propagate: 0.30224 ms
GPU average: 0.031584 ms
GPU D2H: 0.047968 ms
Matrix mismatches (>1e-4): 0, max diff: 4.17233e-07
Average mismatches (>1e-4): 0, max diff: 2.08616e-07
Speedup: 0.619596
liuy41@cuda01:~$ ./radiator_gpu -t -n 64 -m 128 -p 20
CPU compute: 1.76643 ms
GPU alloc: 0.084256 ms
GPU H2D: 0.045056 ms
GPU propagate: 0.284224 ms
GPU average: 0.024832 ms
GPU D2H: 0.072096 ms
Matrix mismatches (>1e-4): 0, max diff: 4.17233e-07
Average mismatches (>1e-4): 0, max diff: 1.49012e-07
Speedup: 5.71556
liuy41@cuda01:~$ ./radiator_gpu -t -n 128 -m 64 -p 20
CPU compute: 1.7249 ms
GPU alloc: 0.088192 ms
GPU H2D: 0.046464 ms
GPU propagate: 0.289248 ms
GPU average: 0.022528 ms
GPU D2H: 0.047552 ms
Matrix mismatches (>1e-4): 0, max diff: 4.17233e-07
Average mismatches (>1e-4): 0, max diff: 1.78814e-07
Speedup: 5.53248
liuy41@cuda01:~$ ./radiator_gpu -t -n 64 -m 64 -p 100
CPU compute: 4.30844 ms
GPU alloc: 0.08432 ms
GPU H2D: 0.02816 ms
GPU propagate: 0.780288 ms
GPU average: 0.023008 ms
GPU D2H: 0.048224 ms
Matrix mismatches (>1e-4): 0, max diff: 1.43051e-06
Average mismatches (>1e-4): 0, max diff: 7.7486e-07
Speedup: 5.36346
liuy41@cuda01:~$ ./radiator_gpu -t -n 70 -m 70 -p 20
Error: block (16x16) must divide matrix (70x70)
```

Test	(n, m, p)	CPU compute (ms)	GPU propagate (ms)	Max matrix diff	Max avg diff	Speedup
1	64×64, p=20 (GPU only)	N/A	0.3215	N/A	N/A	N/A
2	64×64, p=20	0.8630	0.2861	4.17×10^{-7}	1.79×10^{-7}	2.725
3	32×32, p=20	0.2068	0.3022	4.17×10^{-7}	2.09×10^{-7}	0.620
4	64×128, p=20	1.7664	0.2842	4.17×10^{-7}	1.49×10^{-7}	5.716
5	128×64, p=20	1.7249	0.2892	4.17×10^{-7}	1.79×10^{-7}	5.532
6	64×64, p=100	4.3084	0.7803	1.43×10^{-6}	7.75×10^{-7}	5.363
7	128×128, p=20	3.4860	0.3339	4.77×10^{-7}	1.49×10^{-7}	9.769
8	256×256, p=20	19.8525	0.3571	5.36×10^{-7}	2.68×10^{-7}	51.881
9	70×70, p=20	—	—	—	—	—

speedup = CPU_compute / GPU_compute

- The GPU results were compared against the CPU. No mismatches larger than 10^{-4} were found.
- Significant speedup was achieved for larger matrix sizes. The table summarizes key results.
  If the matrix size is not a multiple of the block size (e.g., 70x70), an error will occur.

Task 3 - performance improvement

The test was conducted at the reference scale with different block sizes using `benchmark_task3.sh`. Due to the large scale, only the GPU was tested, and the test was performed twice.
For the small scale, 16x4 was selected for the CPU/GPU comparison.
To keep it short, not all the output is included below.

```
liuy41@cuda01:~$ chmod +x benchmark_task3.sh
./benchmark_task3.sh
Done. Results in bench.txt
liuy41@cuda01:~$ cat bench.txt
bx,by,Propagate_ms,Average_ms,Total_ms
16,16,5086.71,11.225,5097.94
16,8,4694.81,11.1712,4705.98
16,4,4528.65,11.1849,4539.83
16,32,5856.71,11.2311,5867.94
32,16,5362.82,11.203,5374.02
32,8,4742.91,11.1964,4754.11
32,4,4540.25,11.2284,4551.48
32,32,6529.69,11.1989,6540.89
64,16,6092.64,11.2312,6103.87
64,8,4923.74,11.296,4935.04
64,4,4567.81,11.3395,4579.15
64,32,0.764896,11.2067,11.9716
8,16,5339.41,11.3716,5350.78
8,8,5063.87,11.3938,5075.26
8,4,7324.81,11.316,7336.13
8,32,6081.1,11.3583,6092.46
```

```
./radiator_gpu_task3 -t -n 64 -m 64 -p 20 -bx 16 -by 4
./radiator_gpu_task3 -t -n 128 -m 128 -p 20 -bx 16 -by 4
./radiator_gpu_task3 -t -n 256 -m 256 -p 20 -bx 16 -by 4
./radiator_gpu_task3 -t -n 512 -m 512 -p 20 -bx 16 -by 4
./radiator_gpu_task3 -t -n 1024 -m 1024 -p 20 -bx 16 -by 4
```

```
// Reference size test
liuy41@cuda01:~$ ../radiator_gpu_task3 -c -t -n 15360 -m 15360 -p 1000 -bx 16 -by 4
GPU alloc: 0.314976 ms
GPU H2D: 301.574 ms
GPU propagate: 4529.31 ms
GPU average: 11.272 ms
GPU D2H: 412.849 ms
```

bx	by	Propagate (ms)	Average (ms)	Total (ms)
16	16	5086.1 – 5086.7	11.23 – 11.26	5097.4 – 5097.9
16	8	4694.8 – 4696.5	11.17 – 11.22	4706.0 – 4707.8
16	4	4528.7 – 4530.3	11.18 – 11.24	4539.8 – 4541.6
16	32	5855.9 – 5856.7	11.23 – 11.25	5867.2 – 5867.9
32	16	5362.8 – 5367.9	11.19 – 11.20	5374.0 – 5379.1
32	8	4742.9 – 4743.5	11.19 – 11.20	4754.1 – 4754.7
32	4	4540.3 – 4542.6	11.21 – 11.23	4551.5 – 4553.8
32	32	6529.7 – 6532.4	11.20 – 11.24	6540.9 – 6543.6
64	16	6092.6 – 6101.4	11.22 – 11.23	6103.9 – 6112.6
64	8	4923.7 – 4955.9	11.28 – 11.30	4935.0 – 4967.2
64	4	4567.8 – 4577.1	11.31 – 11.34	4579.2 – 4588.4
~~64~~	~~32~~	~~0.76 – 0.78~~	~~11.16 – 11.21~~	~~11.95 – 11.97~~
8	16	5339.4 – 5369.5	11.35 – 11.37	5350.8 – 5380.8
8	8	5063.9 – 5109.1	11.39 – 11.40	5075.3 – 5120.5
8	4	7324.8 – 7378.8	11.32 – 11.38	7336.1 – 7390.2
8	32	6081.1 – 6108.5	11.35 – 11.36	6092.5 – 6119.8

- invalid: 64x32 resulted in Propagate ≈ 0.7 ms, which is unrealistic and should be excluded.

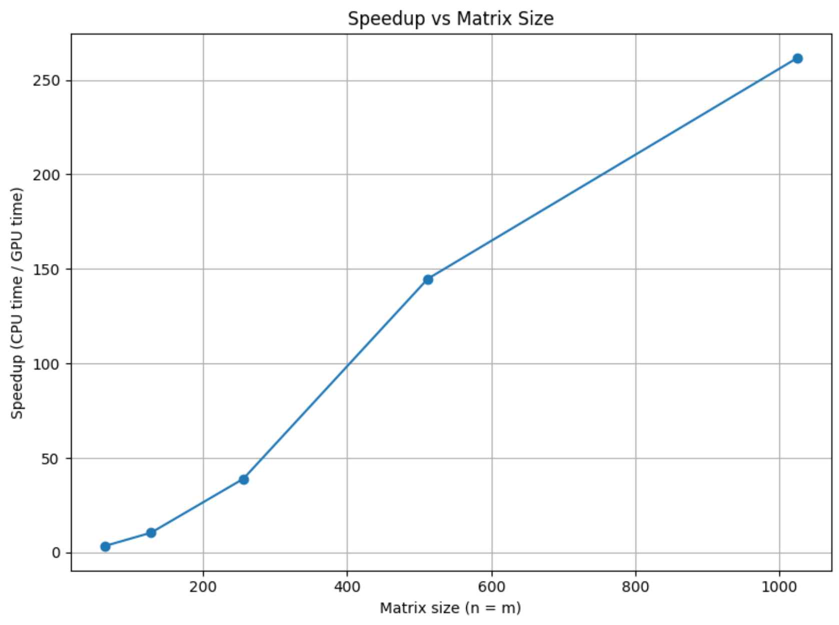
Blocksize

The test was conducted at the reference scale with different block sizes using `benchmark_task3.sh`. Due to the large scale, only the GPU was tested, and the test was performed twice.

For large-scale GPU-only testing (15360×15360×1000) at the reference scale, the total GPU compute time for block size **(16×4)** is the lowest (≈4.54 s), making it the fastest.

- The Propagate kernel is most affected by **block flatness**: Flatter blocks with smaller `by` values align better with row-major memory layout, resulting in significantly lower Propagate times.
- For example: 16×4 (≈4530 ms) < 16×8 (≈4696 ms) < 16×16 (≈5086 ms). different block shapes such as 16×16 and 64×4 (both with 256 threads) resulted in different performance due to differences in memory access alignment and warp scheduling.

- The Average kernel shows consistent runtimes (~11 ms) across all configurations, so it is not a performance bottleneck.



For the small-scale matrix, we set m=n and tested both GPU and CPU using the optimal block size (16×4) found earlier. Speedup curve was measured by running both CPU and GPU on sizes 64^2,128^2,...,1024^2. It shows a significant improvement in GPU acceleration as the matrix size increases.

After testing with similar sizes as the last assignment (such as 1000×1000 and 5000×5000), I found that the new reduce is slower at scales compared to the previous two-stage reduction.

One possible reason is that the new `average_kernel` saves one kernel launch and avoids the intermediate array, but each thread needs to scan the whole row. This increases the pressure on global memory bandwidth. Also, only one thread processes each row, so the parallelism is low.

Task 4 - double precision version

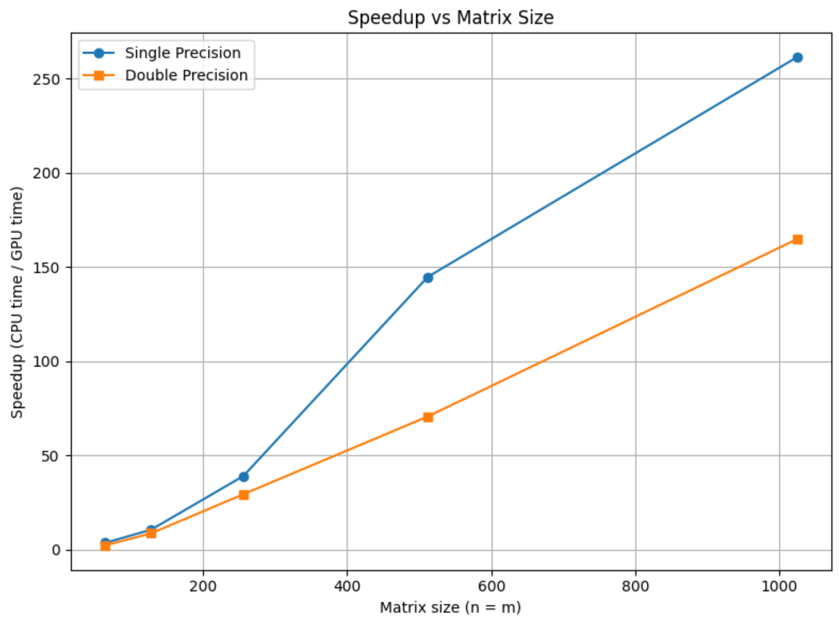
The code was modified from single precision (`float`) to double precision (`double`), then the same tests as in task3 were performed and compared with the single precision results

Reference size test

```
liuy41@cuda01:~$ ./radiator_gpu_dp -c -t -n 15360 -m 15360 -p 1000 -bx 16 -by 4
GPU alloc: 0.388448 ms
GPU H2D: 600.307 ms
GPU propagate: 9099.79 ms
GPU average: 48.3792 ms
GPU D2H: 839.998 ms
liuy41@cuda01:~$
```

Blocksize test

```
liuy41@cuda01:~$ chmod +x benchmark_task4.sh
./benchmark_task4.sh
liuy41@cuda01:~$   cat bench_dp.txt
bx,by,Propagate_ms,Average_ms,Total_ms
16,16,9412.89,48.3537,9461.24
16,8,9138.45,48.3472,9186.8
16,4,9182.73,48.7256,9231.46
16,32,10372.2,48.7484,10420.9
32,16,9971.72,49.1107,10020.8
32,8,9324.66,49.4966,9374.16
32,4,9399.83,51.2613,9451.09
32,32,13701.5,49.4337,13750.9
64,16,13669.8,50.552,13720.4
64,8,9755.11,52.1623,9807.27
64,4,9578.43,52.2142,9630.64
64,32,1.07485,48.5055,49.5803
8,16,9679.02,52.9034,9731.92
8,8,9780.97,52.5641,9833.53
8,4,10590.9,53.2037,10644.1
8,32,10201,53.8603,10254.9
```



For comparison, we used the same 16x4 block size as in task3 for the small-scale tests.

Double precision GPU is much slower than single precision, with a lower speedup (CPU vs GPU).

While double precision offers higher accuracy, no significant mismatches were observed in this problem.

bx	by	Total_ms (float)	Total_ms (double)
16	16	5097.4	9461.24
16	8	4706.0	9186.8
16	4	4539.8	9231.46
16	32	5867.2	10420.9
32	16	5374.0	10020.8
32	8	4754.1	9374.16
32	4	4551.5	9451.09
32	32	6540.9	13750.9
64	16	5415.9	13720.4
64	8	4935.0	9807.27
64	4	4561.3	9630.64
8	16	5350.8	9731.92
8	8	5075.3	9833.53
8	4	5555.8	10644.1
8	32	6092.5	10254.9

- As shown in the table, single precision is fastest with 16x4, while double precision is fastest with 16x8.
- At the reference size (n=15360, m=15360, p=1000) and blocksize(16x4), single precision GPU propagate time is ≈ 4529 ms, while double precision is ≈ 9099 ms, making double precision about 2 times slower than single precision at the compute stage.

Summary

Memory transfer and allocation times are similar for single and double precision, so the main performance difference comes from computation. Average kernel shows stable runtimes (11 ms in `single` and 50ms in `double`) and is not a bottleneck. The main slowdown comes from the propagate kernel, which is sensitive to block shape. This is reasonable, as gpus are usually optimized for single precision.