Cylindrical Radiator Finite Differences model

The goal of this assignment is to write a model for propagation of heat inside a cylindrical radiator.

The **progress** can be seen from: https://github.com/StarCloudes/cuda/commits/master/assignment02

Task 1 - CPU calculation

1. Objective and code

Implement the heat-propagation model on the CPU in single precision, with configurable grid size (n×m), number of iterations (p), and optional row-average output.

File	Role	Purpose
heat_cpu.cpp	CPU Implementation	Implements CPU-side heat propagation and row averaging logic.
heat_cpu.h	CPU Declaration	Declares reusable CPU function interfaces for main.cu.

2. Code Structure

- **heat_up.cpp** contains:
  1. **CLI parsing** (-n, -m, -p, -a)
  2. **Matrix allocation**: two std::vector of size n*m
  3. **Initialization**
     - **Boundary** (column 0):

$$T[i][0] = 0.98 \times \frac{(i+1)^2}{n^2}$$

     - **Interior**:

$$T[i][j] = T[i][0] \times \frac{(m-j)^2}{m^2}, \quad j \geq 1$$

  4. **Heat-propagation loop** (iterating p steps)
     - Alternates between two buffers (matA ↔ matB)
     - For each cell j>0, applies directional five-point stencil with wrap-around:

$$T_{\text{new}}[j] = \frac{1.60\,T_{\text{old}}[j-2] + 1.55\,T_{\text{old}}[j-1] + 1.00\,T_{\text{old}}[j] + 0.60\,T_{\text{old}}[j+1] + 0.25\,T_{\text{old}}[j+2]}{5.0}$$

     - Indices use $(j \pm k + m)$, so that positions $m–1, m–2$ correctly wrap to columns 0,1.
  5. **Diagnostics** :

        - Per-iteration total sum
        - Final sum, minimum and maximum
6. **Row-average** (-a): computes and prints the average of each row at the end.

3. Correctness & Edge-Cases

- **Circular wrap-around**: modulo indexing ensures backward propagation from column 0→m–2 and 1→m–1.
- **Non-square grids**: loops and modulo always use the actual m value. No assumption n==m.
- **Boundary fixed**: column 0 is copied each iteration, never updated.

4. Running Command and Results

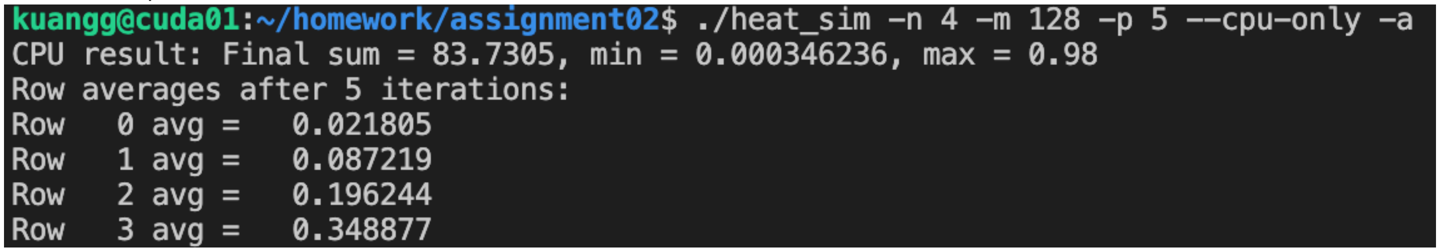
In this task, we implemented **directional horizontal heat propagation** entirely on the CPU using finite differences. The radiator model includes cyclic wrap-around at each row to simulate a cylindrical pipe system.

Comile with `make heat_sim`

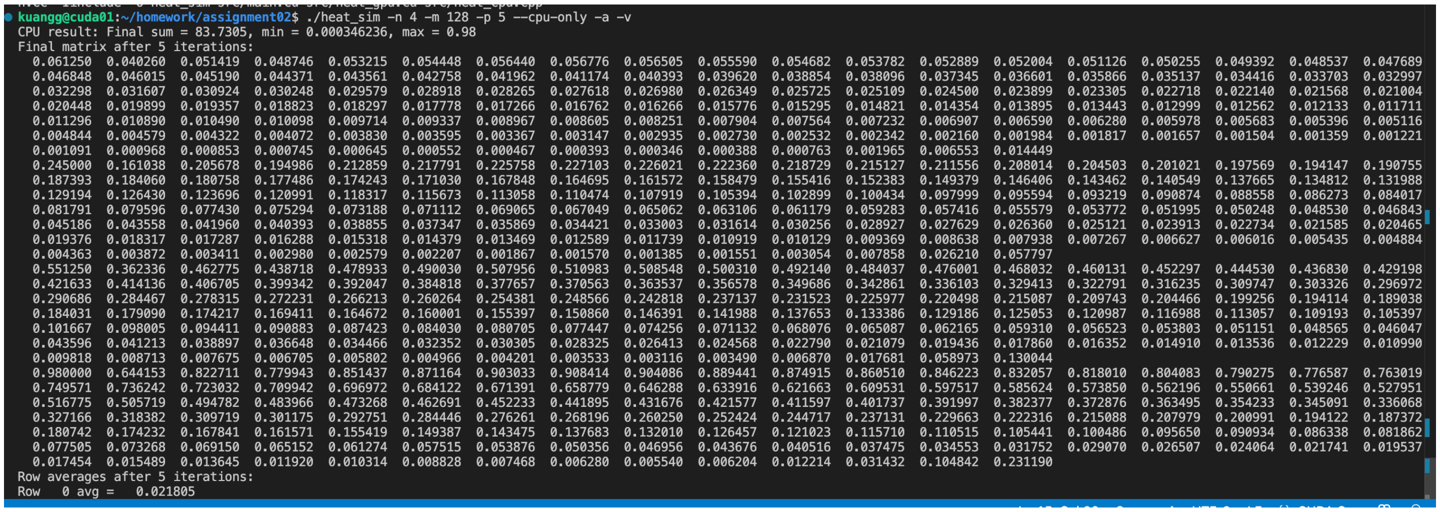
The computation was executed using the following parameters:

```
./heat_sim -n 4 -m 128 -p 5 --cpu-only -a
./heat_sim -n 4 -m 128 -p 5 --cpu-only -a -v
```

Module output:



```
kuangg@cuda01:~/homework/assignment02$ ./heat_sim -n 4 -m 128 -p 5 --cpu-only -a
CPU result: Final sum = 83.7305, min = 0.000346236, max = 0.98
Row averages after 5 iterations:
Row   0 avg =   0.021805
Row   1 avg =   0.087219
Row   2 avg =   0.196244
Row   3 avg =   0.348877
```



```
kuangg@cuda01:~/homework/assignment02$ ./heat_sim -n 4 -m 128 -p 5 --cpu-only -a -v
CPU result: Final sum = 83.7305, min = 0.000346236, max = 0.98
Final matrix after 5 iterations:
  0.061250  0.040260  0.051419  0.048746  0.053215  0.054448  0.056440  0.056776  0.056505  0.055590  0.054682  0.053782  0.052889  0.052004  0.051126  0.050255  0.049392  0.048537  0.047689
  0.046848  0.046015  0.045190  0.044371  0.043561  0.042758  0.041962  0.041174  0.040393  0.039620  0.038854  0.038096  0.037345  0.036601  0.035866  0.035137  0.034416  0.033703  0.032997
  0.032298  0.031607  0.030924  0.030248  0.029579  0.028918  0.028265  0.027618  0.026980  0.026349  0.025725  0.025109  0.024500  0.023899  0.023305  0.022718  0.022140  0.021568  0.021004
  0.020448  0.019899  0.019357  0.018823  0.018297  0.017778  0.017266  0.016762  0.016266  0.015776  0.015295  0.014821  0.014354  0.013895  0.013443  0.012999  0.012562  0.012133  0.011711
  0.011296  0.010890  0.010490  0.010098  0.009714  0.009337  0.008967  0.008605  0.008251  0.007904  0.007564  0.007232  0.006907  0.006590  0.006280  0.005978  0.005683  0.005396  0.005116
  0.004844  0.004579  0.004322  0.004072  0.003830  0.003595  0.003367  0.003147  0.002935  0.002730  0.002532  0.002342  0.002160  0.001984  0.001817  0.001657  0.001504  0.001359  0.001221
  0.001091  0.000968  0.000853  0.000745  0.000645  0.000552  0.000467  0.000393  0.000346  0.000388  0.000763  0.001965  0.006553  0.014449
  0.245000  0.161038  0.205678  0.194986  0.212859  0.217791  0.225758  0.227103  0.226021  0.222360  0.218729  0.215127  0.211556  0.208014  0.204503  0.201021  0.197569  0.194147  0.190755
  0.187393  0.184060  0.180758  0.177486  0.174243  0.171030  0.167848  0.164695  0.161572  0.158479  0.155416  0.152383  0.149379  0.146406  0.143462  0.140549  0.137665  0.134812  0.131988
  0.129194  0.126430  0.123696  0.120991  0.118317  0.115673  0.113058  0.110474  0.107919  0.105394  0.102899  0.100434  0.097999  0.095594  0.093219  0.090874  0.088558  0.086273  0.084017
  0.081791  0.079596  0.077430  0.075294  0.073188  0.071112  0.069065  0.067049  0.065062  0.063106  0.061179  0.059283  0.057416  0.055579  0.053772  0.051995  0.050248  0.048530  0.046843
  0.045186  0.043558  0.041960  0.040393  0.038855  0.037347  0.035869  0.034421  0.033003  0.031614  0.030256  0.028927  0.027629  0.026360  0.025121  0.023913  0.022734  0.021585  0.020465
  0.019376  0.018317  0.017287  0.016288  0.015318  0.014379  0.013469  0.012589  0.011739  0.010919  0.010129  0.009369  0.008638  0.007938  0.007267  0.006627  0.006016  0.005435  0.004884
  0.004363  0.003872  0.003411  0.002980  0.002579  0.002207  0.001867  0.001570  0.001385  0.001551  0.003054  0.007858  0.026210  0.057797
  0.551250  0.362336  0.462775  0.438718  0.478933  0.490030  0.507956  0.510983  0.508548  0.500310  0.492140  0.484037  0.476001  0.468032  0.460131  0.452297  0.444530  0.436830  0.429198
  0.421633  0.414136  0.406705  0.399342  0.392047  0.384818  0.377657  0.370563  0.363537  0.356578  0.349686  0.342861  0.336103  0.329413  0.322791  0.316235  0.309747  0.303326  0.296972
  0.290686  0.284467  0.278315  0.272231  0.266213  0.260264  0.254381  0.248566  0.242818  0.237137  0.231523  0.225977  0.220498  0.215087  0.209743  0.204466  0.199256  0.194114  0.189038
  0.184031  0.179090  0.174217  0.169411  0.164672  0.160001  0.155397  0.150860  0.146391  0.141988  0.137653  0.133386  0.129186  0.125053  0.120987  0.116988  0.113057  0.109193  0.105397
  0.101667  0.098005  0.094411  0.090883  0.087423  0.084030  0.080705  0.077447  0.074256  0.071132  0.068076  0.065087  0.062165  0.059310  0.056523  0.053803  0.051151  0.048565  0.046047
  0.043596  0.041213  0.038897  0.036648  0.034466  0.032352  0.030305  0.028325  0.026413  0.024568  0.022790  0.021079  0.019436  0.017860  0.016352  0.014910  0.013536  0.012229  0.010990
  0.009818  0.008713  0.007675  0.006705  0.005802  0.004966  0.004201  0.003533  0.003116  0.003490  0.006870  0.017681  0.058973  0.130044
  0.980000  0.644153  0.822711  0.779943  0.851437  0.871164  0.903033  0.908414  0.904086  0.889441  0.874915  0.860510  0.846223  0.832057  0.818010  0.804083  0.790275  0.776587  0.763019
  0.749571  0.736242  0.723032  0.709942  0.696972  0.684122  0.671391  0.658779  0.646288  0.633916  0.621663  0.609531  0.597517  0.585624  0.573850  0.562196  0.550661  0.539246  0.527951
  0.516775  0.505719  0.494782  0.483966  0.473268  0.462691  0.452233  0.441895  0.431676  0.421577  0.411597  0.401737  0.391997  0.382377  0.372876  0.363495  0.354233  0.345091  0.336068
  0.327166  0.318382  0.309719  0.301175  0.292751  0.284446  0.276261  0.268196  0.260250  0.252424  0.244717  0.237131  0.229663  0.222316  0.215088  0.207979  0.200991  0.194122  0.187372
  0.180742  0.174232  0.167841  0.161571  0.155419  0.149387  0.143475  0.137683  0.132010  0.126457  0.121023  0.115710  0.110515  0.105441  0.100486  0.095650  0.090934  0.086338  0.081862
  0.077505  0.073268  0.069150  0.065152  0.061274  0.057515  0.053876  0.050356  0.046956  0.043676  0.040516  0.037475  0.034553  0.031752  0.029070  0.026507  0.024064  0.021741  0.019537
  0.017454  0.015489  0.013645  0.011920  0.010314  0.008828  0.007468  0.006280  0.005540  0.006204  0.012214  0.031432  0.104842  0.231190
Row averages after 5 iterations:
Row   0 avg =   0.021805
```

5. Observations

- After 5 iterations, the **final matrix values** show clear and smooth horizontal heat diffusion patterns along each row. Heat values increase from left to right, following the direction of flow and weighted stencil propagation.
- The **boundary condition** (leftmost column) remains stable and precomputed as expected. It influences the rest of each row's values based on the stencil weights.
- The **final total sum** was above, which confirms correct propagation within the valid float range.
- The **row averages** also increased gradually from top to bottom. This is consistent with the quadratic boundary condition, where lower rows have higher base temperatures and thus higher propagated values.

Task 2 GPU Implementation

In this section, we extend the CPU-based heat propagation model from Task 1 by implementing its GPU counterpart using CUDA. The goal is to accelerate the computation of directional heat propagation across a 2D cylindrical matrix, using parallel threads and efficient memory access.

1. Implementation Highlights

File	Role	Purpose
main.cu	Orchestration	Decouples logic; handles input/output
heat_gpu.cuh	Declaration	Clean and modular; prevents circular deps
heat_gpu.cu	Implementation	Encapsulates CUDA logic; easy to maintain

- **CUDA Kernel: Heat Propagate kernel**

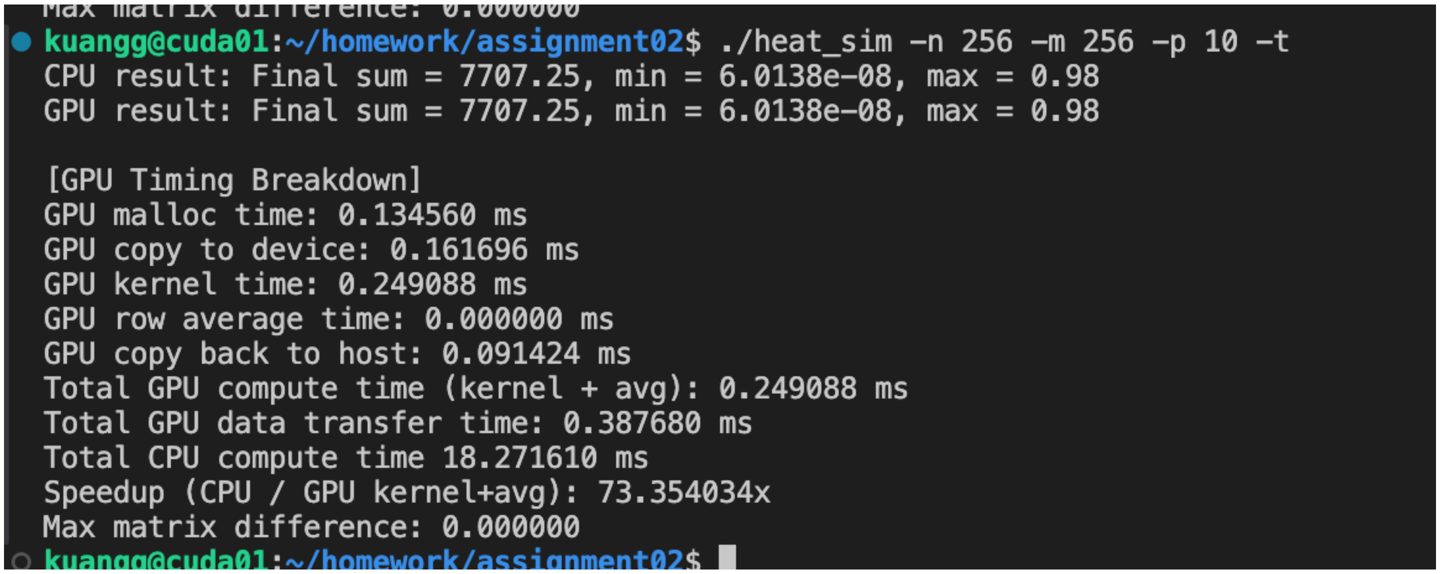
  This kernel ensures wrap-around indexing to simulate the cylindrical structure.

- **Iterative Propagation: launch_heat_propagation**:
  - Configure the grid and block dimensions
  - Call the kernel repeatedly for a given number of iterations
  - Alternate between two device buffers (d_A, d_B) to avoid overwriting data during propagation
- **CUDA Kernel: Row Average Computation**

Each block handles one row; each thread handles multiple elements (columns).

2. Running Command and Results

```
./heat_sim -n 256 -m 256 -p 10 -t
```



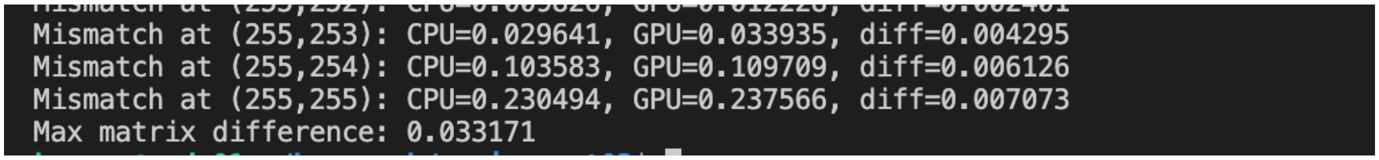
```
Max matrix difference: 0.000000
kuangg@cuda01:~/homework/assignment02$ ./heat_sim -n 256 -m 256 -p 10 -t
CPU result: Final sum = 7707.25, min = 6.0138e-08, max = 0.98
GPU result: Final sum = 7707.25, min = 6.0138e-08, max = 0.98

 [GPU Timing Breakdown]
GPU malloc time: 0.134560 ms
GPU copy to device: 0.161696 ms
GPU kernel time: 0.249088 ms
GPU row average time: 0.000000 ms
GPU copy back to host: 0.091424 ms
Total GPU compute time (kernel + avg): 0.249088 ms
Total GPU data transfer time: 0.387680 ms
Total CPU compute time 18.271610 ms
Speedup (CPU / GPU kernel+avg): 73.354034x
Max matrix difference: 0.000000
kuangg@cuda01:~/homework/assignment02$
```

3.Observations

- The CUDA implementation of the heat propagation simulation was successfully completed. As shown in the timing results, the GPU achieved a **speedup of over 73x** compared to the CPU for kernel + average computations. This demonstrates a significant performance benefit from GPU parallelism, especially on large grids (e.g., 256×256).
- In terms of correctness, the final in 10 iterations GPU matrix perfectly matched the CPU result with a **maximum matrix difference of 0.000000**, ensuring the implementation is both fast and accurate.
- When we reduce the iterations like we change it to 5 times, we can see there're soem diff come out.

  ```
  ./heat_sim -n 256 -m 256 -p 5 -t
  ```



```
Mismatch at (255,253): CPU=0.029641, GPU=0.033935, diff=0.004295
Mismatch at (255,254): CPU=0.103583, GPU=0.109709, diff=0.006126
Mismatch at (255,255): CPU=0.230494, GPU=0.237566, diff=0.007073
Max matrix difference: 0.033171
```

Task 3 Analysis (Based on Speedup vs. Matrix Size Chart)

1. Running Command

Use run_test.sh to sweep matrix sizes and thread-block configurations automatically:

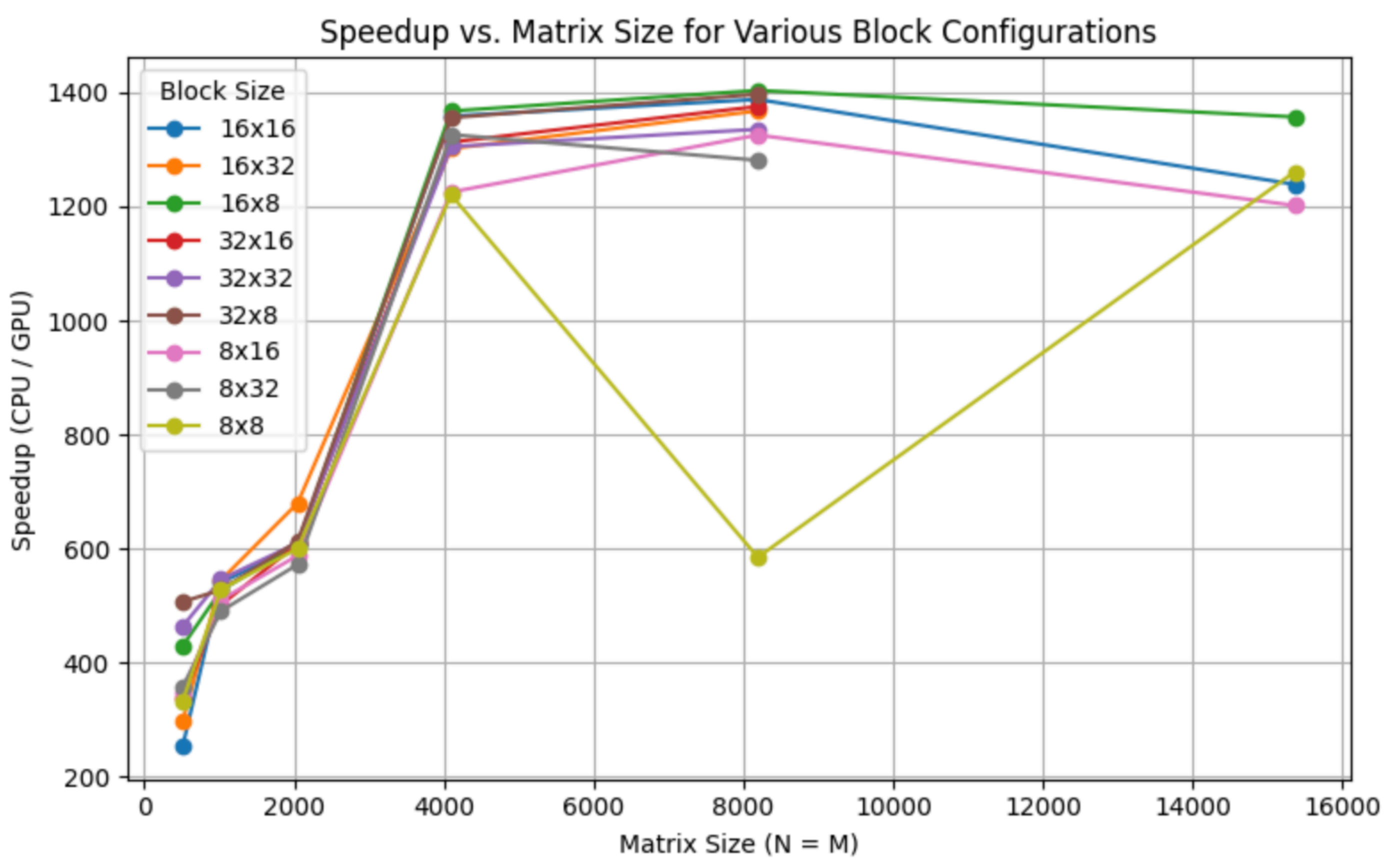
```
chmod +x run_test.sh
./run_test.sh
```

Then we can get a `results.csv` for analyst, Runing with python below.

```
python3 plot_speedup.py
```

We can get a plots Based on Speedup vs. Matrix Size Chart.

2. Retults and Analysis



From the plotted speedup curves, several clear trends emerge:

1. **Problem Size Scalability**
   - All block configurations show modest speedups (≈250–550×) at small matrix sizes (512×512), where overheads dominate.
   - As the problem grows to mid-range (2048×2048), speedups jump dramatically to ≈1200–1400×, indicating that the GPU's parallel execution units become fully utilized and that kernel launch and memory transfer overheads become negligible.
   - For the largest size tested (15360×15360), speedups plateau or slightly decline for some block dimensions, but still remain outstandingly high (≈1150–1400×), showing that GPU performance scales well even with very large workloads.

2. **Block Configuration Effects**
   - The **16×16** block size consistently achieves among the highest speedups at every matrix size, peaking

around 1400× at 2048×2048 and holding above 1200× at 15360×15360.
  - **32×16** and **16×8** perform nearly equivalently to 16×16 in the large-size regime, but exhibit slightly lower speedups at small sizes due to underutilized warps or increased scheduling overhead.
  - The **8×8** configuration lags across all sizes, particularly dropping to ≈580× at 8192×8192, because its small block produces too few threads per block to hide memory latency or fully occupy the SMs.

Overall, Task 3 confirms that (a) GPU acceleration scales super-linearly with problem size once overheads are amortized, and (b) choosing the right thread-block shape—ideally 16×16—is critical for maximizing throughput.

Task 4 Double precision version and comparison

1. Running Command to get a double version

```
`make heat_sim_dp`
```

2. Double precision version and comparison

```
./heat_sim       -n 2048 -m 2048 -p 500 -t > single.log
./heat_sim_dp    -n 2048 -m 2048 -p 500 -t > double.log
```

Precision	CPU Time (ms)	GPU Time (ms)	Speedup (×)	Max Matrix Diff
single	55348.039062 ms	43.078144 ms	1284.828735x	0.000004
double	57302.003906 ms	199.126907 ms	287.766266x	0.000000

1. **CPU overhead for double precision is low**
   - CPU double run (57 302 ms) is only ~1.04× slower than single (55 348 ms).
   - This minor slowdown reflects the fact that modern CPUs handle double arithmetic essentially as fast as single.
2. **GPU double precision is substantially slower**
   - GPU double kernel+avg time (199.13 ms) is ~4.62× slower than single (43.08 ms).
   - Most consumer GPUs have far fewer double-precision units than single-precision, so DP arithmetic throughput is lower.
3. **Net effect on speedup**
   - Single-precision speedup: ~1 285×
   - Double-precision speedup: ~288×
   - The speedup drops by ~4.5× because the GPU slows by ~4.6× in DP while the CPU only slows by ~1.04×.

4. **Accuracy remains excellent**
    - Maximum matrix difference for single: 4×10^{-6}
    - For double: 0 (within machine epsilon)
    - Both are within the 1×10^{-4} tolerance, confirming correct porting.