CUDA Assignment2-Cylindrical Radiator Finite Differences model

File Organization

All source code and related files are under the top-level `CUDA/` directory. A parallel copy under `CUDA/double/` holds the double-precision build, structured as follows:

```
.
└── CUDA/
    ├── Makefile                    # builds single-precision binaries
    ├── precision.h                 # FLOAT = float
    ├── c_debug.h
    ├── cuda_debug.h
    ├── main.c                      # CLI parsing & orchestration
    ├── radiator.h                  # CPU interface
    ├── radiator.c                  # CPU propagation + average
    ├── radiator_cuda.h             # GPU interface
    └── radiator_cuda.cu            # CUDA kernels + host GPU code
    └── double/                     # double-precision variant
        ├── Makefile                # builds DP binaries (radiator_cuda_double)
        ├── precision.h             # FLOAT = double
        ├── c_debug.h
        ├── cuda_debug.h
        ├── main.c
        ├── radiator.h
        ├── radiator.c
        ├── radiator_cuda.h
        └── radiator_cuda.cu
```

Task 1: CPU Implementation of Cylindrical Radiator Heat Diffusion

1. Introduction

The goal of Task 1 is to implement, in single-precision on the CPU, a finite-difference model for heat propagation in a horizontally cylindrical radiator. The requirements are:

1. **Grid size** $n \times m$ (default 32×32, overridable via `-n` / `-m`).
2. **Iteration count** p (default 10, via `-p`).

3. **Inlet boundary** fixed in column 0:

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

4. **Initial profile** for columns 1…*m*–1:

$$T[i][j] = T[i][0] \times \frac{(m-j)^2}{m^2}. \tag{2}$$

5. **Row-only 5-point stencil** with wrap-around, weights (1.6, 1.55, 1.0, 0.6, 0.25), divided by 5.
6. **Optional row averaging** (`-a`), computing $\frac{1}{m-2}\sum_{j=1}^{m-2} T[i][j]$.

---

2. Implementation Details

2.1 Command-Line & Memory Allocation

- **Defaults**: `n = 32, m = 32, iterations = 10, average = 0`.
- **Flags**:
  - `-n <rows>`, `-m <cols>`, `-p <steps>`,
  - `-a` to enable row averaging,
  - `-t` to print timing.
- **Buffers**:

```
FLOAT *matrix_a   = malloc(n*m*sizeof(*matrix_a));
FLOAT *matrix_cpu = skip_cpu ? NULL : malloc(n*m*sizeof(*matrix_cpu));
```

`matrix_cpu` is a copy of `matrix_a` used for the pure-CPU propagation.

2.2 Matrix Initialization

```
for (int i = 0; i < n; ++i) {
  // inlet boundary
  matrix_a[i*m] = 0.98f * (float)(i+1)*(i+1) / (float)(n*n);
  // quadratic decay
  for (int j = 1; j < m; ++j) {
    matrix_a[i*m + j] =
      matrix_a[i*m] * ((float)(m-j)*(m-j)/(float)(m*m));
  }
}
```

- Ensures column 0 matches the required inlet profile
- Enforces parabolic decay away from the inlet

2.3 Ghost-Column Padding

```
static void initialize_matrix_with_ghost_columns(FLOAT *mat, int n, int m) {
    int pm = m + 2;
    for (int i = 0; i < n; ++i) {
        FLOAT base = 0.98f * (float)((i+1)*(i+1)) / (float)(n*n);
        mat[i*pm + 0] = base;
        for (int j = 1; j < m; ++j)
            mat[i*pm + j] = base * ((float)(m-j)*(m-j)/(float)(m*m));
        mat[i*pm + m  ] = mat[i*pm + 0];   // wrap from -1 → m-1
        mat[i*pm + m+1] = mat[i*pm + 1];   // wrap from -2 → m-2
    }
}
```

- Eliminates `% m` in the hot loop (eliminate costly % m in the hot loop).

2.4 Single-Point Stencil Update

```
static void process_row_iteration(FLOAT *dst, const FLOAT *src, int m) {
    // j=1 uses ghost column at src[m-1]
    dst[1] = (1.6f*src[m-1] + 1.55f*src[0] + src[1]
              + 0.6f*src[2]   + 0.25f*src[3]) * 0.2f;
    // j=2…m-2 uniform stencil
    for (int j = 2; j < m-1; ++j) {
        dst[j] = (1.6f*src[j-2] + 1.55f*src[j-1]
                  + src[j]     + 0.6f*src[j+1]
                  + 0.25f*src[j+2]) * 0.2f;
    }
    // preserve fixed boundaries
    dst[0]   = src[0];
    dst[m-1] = src[m-1];
    // update ghosts for next iter
    dst[m]   = dst[0];
    dst[m+1] = dst[1];
}
```

- Weights and divide-by-5 via `*0.2f`.
- Boundaries remain unchanged .

2.5 Multi-Iteration & Buffer Swapping

```
static void process_row_iterations(FLOAT *A, FLOAT *B,
                                   int m, int p) {
    if (p & 1) {
        process_row_iteration(B, A, m);
        swap(A, B);
        --p;
    }
    for (int k = 0; k < p/2; ++k) {
        process_row_iteration(B, A, m);
        process_row_iteration(A, B, m);
    }
}
```

- Handles odd/even iteration counts efficiently.

2.6 Row-Average Thermostat

```
static void average_rows(const FLOAT *mat, int n, int m, FLOAT *avgs) {
    for (int i = 0; i < n; ++i) {
        FLOAT sum = 0.0f;
        for (int j = 1; j < m-1; ++j)
            sum += mat[i*m + j];
        avgs[i] = sum / (m-2);
    }
}
```

- Executed only when `-a` flag is passed.

2.7 Timing in `cpu_propagate_heat`

```
void cpu_propagate_heat(..., float *timings, ...) {
    clock_t t0 = clock();
    propagate_heat(...);
    clock_t t1 = clock();
    timings[0] = (t1-t0)/(double)CLOCKS_PER_SEC;
    if (average) {
        clock_t t2 = clock();
        average_rows(...);
        clock_t t3 = clock();
        timings[1] = (t3-t2)/(double)CLOCKS_PER_SEC;
    }
}
```

- Separates propagation and averaging timings for reporting.

---

3. Compilation Example

```
yuhan@cuda01:~/CUDA$ make
gcc -O3 -w   -c -o main.o main.c
nvcc radiator_cuda.cu -c -O3 -arch=sm_60 -w -I.
gcc -O3 -w   -c -o radiator.o radiator.c
nvcc main.o radiator_cuda.o radiator.o -o radiator_cuda -I.
yuhan@cuda01:~/CUDA$ ./radiator_cuda -n  256 -m  256 -p 100 -a -t
```

Sample output:

```
yuhan@cuda01:~/CUDA$ ./radiator_cuda -n  256 -m  256 -p 100 -a -t
size(256, 256)

CPU:
Propagation: 0.004291s
Averaging: 0.000049s
Total: 0.004340s
```

Task 2: GPU Parallel Implementation

1. Overview

The goal of Task 2 is to extend the CPU-only heat-diffusion model to the GPU using CUDA, achieving:

- Skip the CPU run with a `-c` flag, specify matrix size (`-n`/`-m`) and iteration count (`-p`), and enable timing output with `-t`.
- Performance measurement: Time each GPU phase—device allocation, host→device copy, propagation kernel, averaging kernel, device→host copy, and device deallocation—using CUDA events.

---

2. Implementation Details

2.1 Host–Device Integration (`main.c`)

1. **Command-line parsing** adds a `-c` flag to set `skip_cpu`. All other flags (`-n`, `-m`, `-p`, `-a`, `-t`) mirror Task 1 .
2. **Host buffers**:

```
FLOAT *matrix_cuda    = malloc(n*m*sizeof(*matrix_cuda));
FLOAT *cuda_averages = malloc(n   *sizeof(*cuda_averages));
memcpy(matrix_cuda, matrix_a, n*m*sizeof(*matrix_a));
```

3. **Conditional CPU run**:

```
if (!skip_cpu)
    cpu_propagate_heat(matrix_a, matrix_cpu, …);
```

4. **GPU invocation**:

```
cuda_propagate_heat(matrix_cuda, n, m, iterations,
                    gpu_times, cuda_averages, average);
```

5. **Timing & speedup printout** under `-t`, mirroring the CPU section but with six phases for the GPU.
6. **Validation** only when both runs occur:

```
for (i,j)
  assert(fabs(matrix_a[i*m+j] - matrix_cuda[i*m+j]) <= 1e-4);
if (average)
  for (i)
    assert(fabs(cpu_avg[i] - cuda_averages[i]) <= 1e-4);
```

2.2 Memory Management & Timing (`radiator_cuda.cu`)

- **CUDA events** `start` / `finish` surround each phase.
- **Phase 0**: `cudaMalloc(devA, devB[, devAvg])` → `timings[0]`.
- **Phase 1**: `cudaMemcpy(devA, host)`, `cudaMemcpy(devB, host)` → `timings[1]`.
- **Phase 2**: Propagation kernel launch & `cudaDeviceSynchronize()` → `timings[2]`.
- **Phase 3**: Averaging kernel (if `-a`) → `timings[3]`.
- **Phase 4**: `cudaMemcpy(host, devA[, devAvg])` → `timings[4]`.
- **Phase 5**: `cudaFree(devA, devB[, devAvg])` → `timings[5]`.

2.3 Kernel Design

2.3.1 Propagation Kernel

- **Global-memory version** (`propagate_row_heat_per_block`) simply reads/writes `A[i*m + j]` and `B[i*m + j]`, using

```
row_iteration_cuda(A, i, j, m);
```

which applies the five-point stencil with wrap-around via `(j±k + m)%m`.

- **Shared-memory variant** (`propagate_row_heat_per_block_with_shared`) allocates per-block tiles in `__shared__` arrays and updates them with halo regions before applying the same stencil, reducing global load traffic.

2.3.2 Averaging Kernel

```
__global__ void average_row_heat(FLOAT *M, int n, int m, FLOAT *out) {
  extern __shared__ FLOAT sum_row[];
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  sum_row[threadIdx.y] = 0.0f;
  __syncthreads();
  for (int x = threadIdx.x+1; x < m-1; x += blockDim.x)
    atomicAdd(&sum_row[threadIdx.y], M[row*m + x]);
  __syncthreads();
  if (threadIdx.x == 0)
    out[row] = sum_row[threadIdx.y] / (m-2);
}
```

- Uses one shared-array entry per row and `atomicAdd` for partial sums.

2.4 Grid & Block Configuration

```
dim3 blockP(BLOCK_SIZE_X, BLOCK_SIZE_Y);
dim3 gridP((m+BLOCK_SIZE_X-1)/BLOCK_SIZE_X,
           (n+BLOCK_SIZE_Y-1)/BLOCK_SIZE_Y);

#ifdef SHARED_MEM
  size_t shmem = ...;
  propagate_shared<<<gridP, blockP, shmem>>>(...);
#else
  propagate_global<<<gridP, blockP>>>(...);
#endif

dim3 blockA(BLOCK_SIZE_X_AVG, BLOCK_SIZE_Y_AVG);
dim3 gridA(1, (n+BLOCK_SIZE_Y_AVG-1)/BLOCK_SIZE_Y_AVG);
average_row_heat<<<gridA, blockA,
                   BLOCK_SIZE_Y_AVG*sizeof(FLOAT)>>>(...);
```

- **Propagation**: 1D blocks over columns, multiple blocks per row if needed.
- **Averaging**: one block per row group in `y`.

2.5 Error Handling and Robustness

To ensure the reliability of our CUDA implementation, we implemented comprehensive error checking throughout the code:

2.5.1 CUDA Error Checking Macro

```
#define CUDA_CHECK(call, flag)                                              \
    do                                                                      \
    {                                                                       \
        cudaError_t err = call;                                             \
        if (err != cudaSuccess && flag <= DEBUG_THREAD)                     \
        {                                                                   \
            printf("Error: %s:%d, ", __FILE__, __LINE__);                  \
            printf("code:%d, reason: %s\n", err, cudaGetErrorString(err)); \
            exit(EXIT_FAILURE);                                             \
        }                                                                   \
    }
```

This macro wraps all CUDA API calls, providing:

- File and line number information for error localization
- Error code and human-readable error description
- Immediate program termination on failure
- Configurable debug levels via the `DEBUG_THREAD` variable

2.5.2 Memory Management Considerations

For large matrix sizes, our implementation:

1. **Checks allocation results**: All `cudaMalloc` calls are verified using the `CUDA_CHECK` macro to detect out-of-memory conditions.
2. **Handles device memory limitations**: When facing GPU memory constraints (particularly relevant for large matrices like 15360×15360), the program provides a clear error message identifying the specific allocation that failed.
3. **Resource cleanup**: Even in error conditions, we ensure proper cleanup of previously allocated resources to prevent memory leaks.

Example error output for memory allocation failure:

```
Error: radiator_cuda.cu:118, code:2, reason: out of memory
```

This error handling approach balances robustness with performance by:

- Providing immediate feedback when resources are insufficient

- Maintaining clean shutdown procedures
- Including diagnostic information for debugging
- Avoiding excessive error-checking overhead in release builds via debug levels

For production deployments, the implementation could be extended with graceful fallbacks for large problem sizes, such as tiled execution or dynamic reduction of problem size when memory constraints are detected.

---

3. Compilation Example

```
yuhan@cuda01:~/CUDA$ make
gcc -O3 -w   -c -o main.o main.c
nvcc radiator_cuda.cu -c -O3 -arch=sm_60 -w -I.
gcc -O3 -w   -c -o radiator.o radiator.c
nvcc main.o radiator_cuda.o radiator.o -o radiator_cuda -I.
yuhan@cuda01:~/CUDA$ ./radiator_cuda -n 1024 -m 1024 -p 200 -a -t -c
```

Sample output(1024*1024):

```
yuhan@cuda01:~/CUDA$ ./radiator_cuda -n 1024 -m 1024 -p 200 -a -t -c
size(1024, 1024)

--------timing info--------
GPU:
Malloc: 0.202112ms
Copy to Device: 1.737728ms
Propagation: 4.516032ms
Averaging: 0.227264ms
Copy to Host: 0.844992ms
Free: 0.393920ms
Total: 7.922048ms
```

Non-square grids(1024, 2048):

```
yuhan@cuda01:~/CUDA$ ./radiator_cuda -n 1024 -m 2048 -p 100 -t -a -c
size(1024, 2048)

--------timing info--------
GPU:
Malloc: 0.211488ms
Copy to Device: 3.142880ms
Propagation: 4.059360ms
Averaging: 0.429728ms
Copy to Host: 1.623552ms
Free: 0.589536ms
Total: 10.056543ms
```

---

4. Conclusion

- **Shared-memory stencil** may further speed propagation by reducing global loads.
- **Tree-based or warp-reduction averaging** can eliminate costly atomics.
- **2D-block partitioning** could improve memory coalescing and occupancy.

Task 3 – Performance Improvement

1. Experimental Setup

- **Matrix sizes & iterations:**
  - Small: 256^2, 512^2 (p=100)
  - Medium: 1024^2, 2048^2 (p=200)
  - Large: 4096^2 (p=200), 15360^2 (p=1000)
- **Build & Run Flags**
  - `-a` to compute per-row averages
  - `-t` to print detailed timings
- **Default block sizes:**
  - Propagation kernel: 256×1 threads per block
  - Averaging kernel: 64×1 threads per block

---

2. Measured Results

Table 1. Raw Timing Metrics

n=m	CPU Propagation (ms)	CPU Averaging (ms)	GPU Propagation (ms)	GPU Averaging (ms)	H→D Transfer (ms)	D→H Transfer (ms)
256	4.291	0.049	0.4374	0.0342	0.1654	0.0997
512	14.925	0.223	0.9094	0.0612	0.6808	0.3190
1024	371.119	0.827	4.5193	0.2367	1.6915	0.8522
2048	453.579	3.468	15.0881	0.7173	6.3618	3.3595
4096	1876.774	13.939	54.2144	2.5474	24.3568	12.7787
15360	119739.594	191.387	4735.9429	27.1244	306.6423	158.3948

Table 2. Speedup & Total Metrics

n=m	CPU Total (ms)	GPU Total incl. transfers (ms)	Propagation Speedup ×	Averaging Speedup ×	Overall (excl. transfers) ×	Overall (incl. transfers) ×
256	4.340	0.9164	9.81	1.43	9.20	4.74
512	15.148	2.3355	16.41	3.64	15.61	6.49
1024	371.946	7.8948	82.12	3.49	78.21	47.11
2048	457.047	26.7496	30.06	4.83	28.92	17.09
4096	1890.713	96.9882	34.62	5.47	33.31	19.49
15360	119930.981	5233.3203	25.28	7.06	25.18	22.92

- Propagation speedup peaks at 82× for 1024^2, then levels off around 25–35× for larger matrices.
- Overall (including transfers) speedup is highest (~47×) at 1024^2, then settles to 19–23× for the biggest sizes.
- Data-transfer overhead (H→D + D→H) grows from ~20% of the GPU total at 4096^2 down to ~9% at 15360^2.
- Max |CPU–GPU|" (all were <1e–4)

3.Compared to the first assignment.

In this assignment, I rewrote the "average per row" reduction as a true parallel kernel:

- Per-row shared buffer + atomicAdd: each block holds a small `row_sum[]` in shared memory, threads atomically accumulate their slice of the row, then thread 0 writes out the average.
- Timing (1024×1024 case):
  - GPU averaging: 0.2367 ms
  - CPU averaging (sequential loop over 1022 elements): 0.827 ms
    → 3.5× faster on GPU .
- Scaling to large rows (15360 columns):
  - GPU: 27.12 ms

- CPU: 191.39 ms
  → 7.1× speedup .

Vector Length	A1 GPU “reduce” (1000*1000)	A2 GPU “average” (1024*1024)	Speedup
≈1000	2.345 ms	0.237 ms	9.9×

Compared to the first-assignment GPU reduction (2.345 ms for a 1000-element vector), the new shared-memory reduction is nearly 10× faster even on similar-sized rows (0.237 ms vs 2.345 ms) . And against the CPU version it scales up to 7× faster on long rows.

In summary:

- Old GPU reduce (Assignment 1): global-memory, single thread → 2.345 ms @1000
- New GPU average (this work): shared-mem + atomic → 0.237 ms @1024, 27.1 ms @15360
- Speedup over old GPU reduce: ~10× for 1000-length rows
- Speedup over CPU average: up to ~7× for large rows
- The new implementation demonstrates significant performance gains through:
  - More efficient memory access patterns
  - Leveraging shared memory
  - Atomic operations for thread-safe accumulation

---

4. Compilation Example

Run Instructions：

```
./radiator_cuda -n  256 -m  256 -p 100 -a -t
./radiator_cuda -n  512 -m  512 -p 100 -a -t

./radiator_cuda -n 1024 -m 1024 -p 200 -a -t
./radiator_cuda -n 2048 -m 2048 -p 200 -a -t

./radiator_cuda -n 4096 -m 4096 -p 200 -a -t
./radiator_cuda -n 15360 -m 15360 -p 1000 -a -t
```

PS：All running results are displayed at the end of the document.

Task 4 – Double Precision Version

1. Porting to Double Precision

To support double precision, we simply:

1. **Enable** `DOUBLE_PRECISION` in `precision.h` (or pass `-DDOUBLE_PRECISION` to the compiler).
2. Rebuild both CPU and CUDA code (`radiator.c`, `radiator_cuda.cu`) under that switch.
3. Add a make-target in the Makefile that compiles with `-DDOUBLE_PRECISION`.

No other code changes were required—the arithmetic in both host and device kernels automatically switches to `double`.

---

2. Measured Results (Double Precision)

Table 1. Raw Timing Metrics

n=m	CPU Propagation (ms)	CPU Averaging (ms)	GPU Propagation (ms)	GPU Averaging (ms)	H → D Transfer (ms)	D → H Transfer (ms)
256	7.070	0.047	0.7667	0.0558	0.3718	0.1783
512	28.467	0.246	1.9155	0.1500	0.9900	0.5056
1024	217.104	0.849	12.6562	0.5495	3.1361	1.5932
2048	858.785	3.514	44.3803	1.9060	11.3987	5.8077
4096	4064.694	13.741	166.0098	6.6907	44.1661	22.8350
15360	227590.546	194.024	10048.519	69.1955	614.046	316.506

Table 2. Speedup & Total Metrics

n=m	CPU Total (ms)	GPU Total incl. transfers (ms)	Propagation Speedup ×	Averaging Speedup ×	Overall (excl. transfers) ×	Overall (incl. transfers) ×
256	7.117	1.5864	9.22	0.84	8.65	4.49
512	28.713	4.0401	14.86	1.64	13.90	7.11
1024	217.953	18.7320	17.15	1.55	16.50	11.64
2048	862.299	65.4335	19.35	1.84	18.63	13.18
4096	4078.435	243.4098	24.48	2.05	23.62	16.76
15360	227784.570	11054.055	22.65	2.80	22.51	20.61

- **Propagation speedup** climbs from ~9× at 256² up to ~24.5× at 4096², then about ~22.6× at 15360².
- **Averaging speedup** starts below 1× for tiny sizes but grows to ~2.8× at 15360².
- **Overall (including transfers)** peaks at ~16.8× for 4096² and remains ~20.6× for the largest size.

- **Max |CPU–GPU|** differences stayed below 1×10^{-6} for all tests.

---

3. Conclusions

1. **Throughput**
   - Double-precision FLOP throughput is only ~1/32 that of single precision, so the GPU propagation time grows from ~4.5 ms (SP) to ~12.7 ms (DP) at 1024^2 — a ~3× slowdown vs. the raw 32× theoretical, thanks to memory-bound behavior.
   - CPU DP loops run ~2× slower than CPU SP (217 ms vs 371 ms for 1024^2).
2. **Net Speedups**
   - At 1024^2, DP GPU still beats CPU by 17× in propagation and 11.6× end-to-end.
   - At 15360^2, DP GPU maintains a 22.5× propagation speedup and 20.6× overall.
   - These sustained gains show that even with FP64's lower throughput, the GPU remains advantageous for large problems.
3. **Averaging Kernel**
   - DP averaging uses the same shared-memory + `atomicAdd` strategy.
   - Speedups range from **<1×** at tiny sizes (0.84× at 256) to **2.8×** at 15360^2 — slower than SP's 7× at that size but still beneficial beyond ~512 columns.
4. **Data-Transfer Overhead**
   - H⇄D overhead is ~40–60 MB/sec for doubles, representing ~5–6% of GPU time at 4096^2 and ~8% at 15360^2 — similar fraction to single precision.
5. Performance degrades compared to single precision but remains GPU-favored for all but the smallest sizes.

---

4. Compilation Example

Run Instructions：

```
./radiator_cuda_double -n  256 -m  256 -p 100 -a -t
./radiator_cuda_double -n  512 -m  512 -p 100 -a -t

./radiator_cuda_double -n 1024 -m 1024 -p 200 -a -t
./radiator_cuda_double -n 2048 -m 2048 -p 200 -a -t

./radiator_cuda_double -n 4096 -m 4096 -p 200 -a -t
./radiator_cuda_double -n 15360 -m 15360 -p 1000 -a -t
```

PS：All running results are displayed at the end of the document.

FLOAT version Results：

Result(256, 256)：

```
yuhan@cuda01:~/CUDA$ ./radiator_cuda -n  256 -m  256 -p 100 -a -t
size(256, 256)

--------timing info--------
GPU:
Malloc: 0.088704ms
Copy to Device: 0.165440ms
Propagation: 0.437440ms
Averaging: 0.034208ms
Copy to Host: 0.099712ms
Free: 0.090944ms
Total: 0.916448ms


CPU:
Propagation: 0.004291s
Averaging: 0.000049s
Total: 0.004340s

Speedups:
Propataion: 9.809345
Averaging: 1.432413
Overall (Excluding transfers): 9.201777
Overall (Including transfers): 4.735675
```

Result(512, 512)：

```
yuhan@cuda01:~/CUDA$ ./radiator_cuda -n  512 -m  512 -p 100 -a -t
size(512, 512)

--------timing info--------
GPU:
Malloc: 0.161792ms
Copy to Device: 0.680832ms
Propagation: 0.909376ms
Averaging: 0.061248ms
Copy to Host: 0.318976ms
Free: 0.203264ms
Total: 2.335488ms
```

```
CPU:
Propagation: 0.014925s
Averaging: 0.000223s
Total: 0.015148s

Speedups:
Propataion: 16.412353
Averaging: 3.640935
Overall (Excluding transfers): 15.606455
Overall (Including transfers): 6.486010
```

Result(1024, 1024):

```
yuhan@cuda01:~/CUDA$ ./radiator_cuda -n 1024 -m 1024 -p 200 -a -t
size(1024, 1024)

--------timing info--------
GPU:
Malloc: 0.204416ms
Copy to Device: 1.691456ms
Propagation: 4.519264ms
Averaging: 0.236672ms
Copy to Host: 0.852224ms
Free: 0.390720ms
Total: 7.894752ms


CPU:
Propagation: 0.371119s
Averaging: 0.000827s
Total: 0.371946s

Speedups:
Propataion: 82.119340
Averaging: 3.494287
Overall (Excluding transfers): 78.206684
Overall (Including transfers): 47.113071
```

Result(2048, 2048):

```
yuhan@cuda01:~/CUDA$ ./radiator_cuda -n 2048 -m 2048 -p 200 -a -t
size(2048, 2048)

--------timing info--------
GPU:
Malloc: 0.233216ms
```

```
Copy to Device: 6.361792ms
Propagation: 15.088096ms
Averaging: 0.717312ms
Copy to Host: 3.359488ms
Free: 0.989728ms
Total: 26.749630ms


CPU:
Propagation: 0.453579s
Averaging: 0.003468s
Total: 0.457047s

Speedups:
Propataion: 30.062046
Averaging: 4.834717
Overall (Excluding transfers): 28.917127
Overall (Including transfers): 17.086106
```

Result(4096, 4096):

```
yuhan@cuda01:~/CUDA$ ./radiator_cuda -n 4096 -m 4096 -p 200 -a -t
size(4096, 4096)

--------timing info--------
GPU:
Malloc: 0.261600ms
Copy to Device: 24.356833ms
Propagation: 54.214432ms
Averaging: 2.547360ms
Copy to Host: 12.778688ms
Free: 2.829312ms
Total: 96.988228ms


CPU:
Propagation: 1.876774s
Averaging: 0.013939s
Total: 1.890713s

Speedups:
Propataion: 34.617607
Averaging: 5.471939
Overall (Excluding transfers): 33.309608
Overall (Including transfers): 19.494251
```

Result(15360, 15360):

```
yuhan@cuda01:~/CUDA$ ./radiator_cuda -n 15360 -m 15360 -p 1000 -a -t
size(15360, 15360)

CPU processing row: 0/15360
CPU processing row: 1536/15360
CPU processing row: 3072/15360
CPU processing row: 4608/15360
CPU processing row: 6144/15360
CPU processing row: 7680/15360
CPU processing row: 9216/15360
CPU processing row: 10752/15360
CPU processing row: 12288/15360
CPU processing row: 13824/15360
--------timing info--------
GPU:
Malloc: 0.318464ms
Copy to Device: 306.642273ms
Propagation: 4735.942871ms
Averaging: 27.124416ms
Copy to Host: 158.394821ms
Free: 4.897024ms
Total: 5233.320312ms


CPU:
Propagation: 119.739594s
Averaging: 0.191387s
Total: 119.930981s

Speedups:
Propataion: 25.283159
Averaging: 7.055894
Overall (Excluding transfers): 25.179359
Overall (Including transfers): 22.916805
```

Double version Results:

Result(256, 256):

```
yuhan@cuda01:~/CUDA/double$ ./radiator_cuda_double -n  256 -m  256 -p 100 -a -t
size(256, 256)

--------timing info--------
```

```
GPU:
Malloc: 0.115008ms
Copy to Device: 0.371840ms
Propagation: 0.766688ms
Averaging: 0.055808ms
Copy to Host: 0.178272ms
Free: 0.098816ms
Total: 1.586432ms


CPU:
Propagation: 0.007070s
Averaging: 0.000047s
Total: 0.007117s

Speedups:
Propataion: 9.221483
Averaging: 0.842173
Overall (Excluding transfers): 8.652929
Overall (Including transfers): 4.486168
```

Result(512, 512):

```
yuhan@cuda01:~/CUDA/double$ ./radiator_cuda_double -n  512 -m  512 -p 100 -a -t
size(512, 512)

--------timing info--------
GPU:
Malloc: 0.210496ms
Copy to Device: 0.989984ms
Propagation: 1.915520ms
Averaging: 0.150016ms
Copy to Host: 0.505632ms
Free: 0.268416ms
Total: 4.040064ms


CPU:
Propagation: 0.028467s
Averaging: 0.000246s
Total: 0.028713s

Speedups:
Propataion: 14.861239
Averaging: 1.639825
Overall (Excluding transfers): 13.900991
```

```
Overall (Including transfers): 7.107066
```

Result(1024, 1024)：

```
yuhan@cuda01:~/CUDA/double$ ./radiator_cuda_double -n 1024 -m 1024 -p 200 -a -t
size(1024, 1024)

--------timing info--------
GPU:
Malloc: 0.205056ms
Copy to Device: 3.136064ms
Propagation: 12.656224ms
Averaging: 0.549472ms
Copy to Host: 1.593248ms
Free: 0.591936ms
Total: 18.732000ms


CPU:
Propagation: 0.217104s
Averaging: 0.000849s
Total: 0.217953s

Speedups:
Propataion: 17.153932
Averaging: 1.545120
Overall (Excluding transfers): 16.504469
Overall (Including transfers): 11.635330
```

Result(2048, 2048)：

```
yuhan@cuda01:~/CUDA/double$ ./radiator_cuda_double -n 2048 -m 2048 -p 200 -a -t
size(2048, 2048)

--------timing info--------
GPU:
Malloc: 0.230944ms
Copy to Device: 11.398720ms
Propagation: 44.380322ms
Averaging: 1.905984ms
Copy to Host: 5.807712ms
Free: 1.709824ms
Total: 65.433510ms


CPU:
```

```
Propagation: 0.858785s
Averaging: 0.003514s
Total: 0.862299s

Speedups:
Propataion: 19.350580
Averaging: 1.843667
Overall (Excluding transfers): 18.629679
Overall (Including transfers): 13.178247
```

Result(4096, 4096):

```
yuhan@cuda01:~/CUDA/double$ ./radiator_cuda_double -n 4096 -m 4096 -p 200 -a -t
size(4096, 4096)

--------timing info--------
GPU:
Malloc: 0.245344ms
Copy to Device: 44.166080ms
Propagation: 166.009827ms
Averaging: 6.690656ms
Copy to Host: 22.835039ms
Free: 3.462816ms
Total: 243.409760ms


CPU:
Propagation: 4.064694s
Averaging: 0.013741s
Total: 4.078435s

Speedups:
Propataion: 24.484658
Averaging: 2.053760
Overall (Excluding transfers): 23.615655
Overall (Including transfers): 16.755429
```

Result(15360, 15360):

```
yuhan@cuda01:~/CUDA/double$ ./radiator_cuda_double -n 15360 -m 15360 -p 1000 -a -t
size(15360, 15360)

CPU processing row: 0/15360
CPU processing row: 1536/15360
CPU processing row: 3072/15360
CPU processing row: 4608/15360
```

```
CPU processing row: 6144/15360
CPU processing row: 7680/15360
CPU processing row: 9216/15360
CPU processing row: 10752/15360
CPU processing row: 12288/15360
CPU processing row: 13824/15360
--------timing info--------
GPU:
Malloc: 0.398272ms
Copy to Device: 614.045959ms
Propagation: 10048.518555ms
Averaging: 69.195518ms
Copy to Host: 316.506226ms
Free: 5.390912ms
Total: 11054.054688ms


CPU:
Propagation: 227.590546s
Averaging: 0.194024s
Total: 227.784570s

Speedups:
Propataion: 22.649165
Averaging: 2.803997
Overall (Excluding transfers): 22.513442
Overall (Including transfers): 20.606428
```