CSC4005 FA22 HW04

Haoran Sun (haoransun@link.cuhk.edu.cn)

1 Introduction

In this project, a heat distribution model using Jacob iteration (same as finite difference) is implemented. Despite a sequential version, the program is also accelerated by common parallelization libraries: MPICH, OpenMP, Pthread, and CUDA. The performance of each method is evaluated.

2 Method

2.1 System setup

The system contains a $n \times n$ square 2D mesh with temperature assigned where x ranging in $[-5, 5]$ and y ranging in $[-5, 5]$. Initially, fire regions Ω is defined as $\{(x, y)|x^2 + y^2 \leq 1\}$, and temperature will be assigned to 100 °C. Other region will be assigned to 20 °C. Note that all fire regions will keep 100 °C and all boundary (edge) region $\{(x, y)|x = --5, 5; y = -5, 5\}$ will keep 100 °C.

2.2 Program design and implementation

The programs are written in the C++ programming language. MPICH, Pthread, OpenMP, and CUDA libraries were used for parallelization. Besides, OpenGL is used for visualization purposes. Also, to improve the performance, the MPI version is further accelerated using OpenMP.

Despite MPI version written separately in `src/main.mpi.cpp`, the main program of other version are all wrapped in `src/main.cpp`. Particularly, CUDA functions are compiled in a separated library `build/lib/libcudalib.a`.

One can refer to A.1 to understand the program design.

2.3 Usage

Remark. For convenience, one can directly build the program by `scripts/build.sh` to compile all targets.

To simplify the compiling process, the CMake build system is used to compile programs and link libraries. One can execute the following lines to build executables.

```
cmake -B build -DCMAKE_BUILD_TYPE=Release -DGUI=ON
cmake --build build
```

To disable the GUI feature, one can set `-DGUI=OFF` in the first line. The compiled programs and libraries are shown in the `build/bin` and `build/lib`. One can directly execute `build/bin/main*.gui` for a visualized demonstration.

```
./build/bin/main.seq.gui
./build/bin/main.omp.gui
./build/bin/main.pth.gui
./build/bin/main.mpi.gui
./build/bin/main.cu.gui
```

One can customize the running parameters such as the number of particles n and simulation steps according to the following lines.

```
./build/bin/main.seq              --dim 100 --nsteps 10000 --record 1
./build/bin/main.omp        -nt 10 --dim 100 --nsteps 10000 --record 1
./build/bin/main.omp        -nt 10 --dim 100 --nsteps 10000 --record 1
./build/bin/main.cu               --dim 100 --nsteps 10000 --record 1
mpirun -np 10 ./build/bin/main.mpi --dim 100 --nsteps 10000 --record 1
```

Remark. To execute **MPI + OpenMP** hybrid program, one can just append `-nt [n]` parameters when executing the MPI program. For example, the following line initializes a program with 10 MPI process, and each process has 2 OpenMP threads, which have $10 \times 2 = 20$ threads in total.

```
mpirun -np 10 ./build/bin/main.mpi -nt 2
```

2.4 Performance evaluation

The program was executed under different configurations to evaluate performance. With 40 different CPU core numbers (from 1 to 40 with increment 1, $p = 1, 2, \ldots, 40$) and 40 different n (from 50 to 2000 with increment 50), 1600 cases in total were sampled for sequential, MPI, OpenMP, and Pthread programs. Test for CUDA program is implemented separately. Recorded runtime is analyzed through the Numpy package in Python. Figures were plotted through the Matplotlib and the Seaborn packages in Python. Analysis codes were written in `analysis/main.ipynb`.

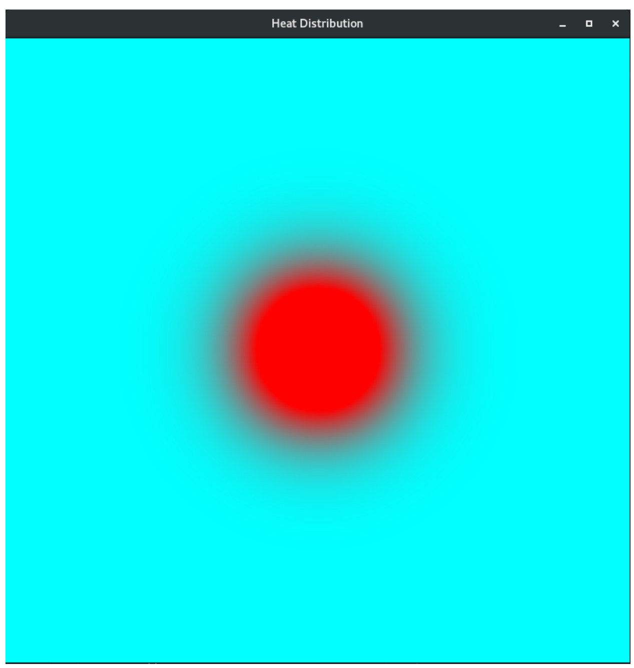

Figure 1: Sample GUI window

3 Result and discussion

3.1 CPU parallelization

From Figure A.2, we can know that when n ranging from 1000 to 2000, MPI, OpenMP, and Pthread have very different performance. For MPI program, the fps almost monotonically decreases with the increase of overall processes. The reason may caused from the communication time. It seems that the time cost of communication increase faster than the computation time when when n increase. For OpenMP program, the fps increases linearly. This proves that multi-thread indeed improve the performance. For Pthread program, the fps increases with the number of threads until it reaches it maximum when the number of threads are around 15.

The heatmap which indicate the rate of acceleration plotted in the Figure A.4 provides some direct visualization of the performances of parallel variants. From this figure we can clearly know that MPI scheme has worst performance–it is even slower when overall threads is greater than 1. For Pthread, we can see it reaches its maximum speed-up rate when the number of thread is around 15. For OpenMP program, when n is large (≥ 1400), speed-up rate will be higher when the number of threads is greater.

3.2 GPU parallelization

GPU parallelization is much more massive than CPU parallelization. This allows one to implemented $n > 10^4$ with high fps, as Figure A.5 shows. Notably, the gpu shared memory is used to accelerate the read operations. (please refer to `__shared__` type and `__syncthreads` function in `cudalib.cu`). According to NVIDIA, the memory access on shared memory is approximately 100× faster than the GPU memory(`__device__`) access.

For example, a naive vector addition in CUDA could be written as

```
__global__ void VecAdd(int *a, int *b, int *c, long int dim){
    // thread partition
    int start_idx = dim / (blockDim.x * gridDim.x) * threadIdx.x;
    int end_idx   = dim / (blockDim.x * gridDim.x) * (threadIdx.x+1);
    if (threadIdx.x+1==blockDim.x) end_idx = dim;
    // vector add
    for (int i = 0; i < dim; i++){
        c[i] = a[i] + b[i];
    }
}
```

During the calculation, each thread in GPU will require to access the memory independently. When the overall thread number is large, the memory miss could cost a huge amount of time. However, in CUDA, we can split those threads into different blocks: for example, if one call a kernel function `kernel` by `kernel<<<16,64>>>()`, then he is asking CUDA to generate 16 blocks where each block has 64 threads, overall $16 \times 64 = 1024$ threads. Similarly, `kernel<<<1,1024>>>()` also calls the function with 1024 threads. In principle, `VecAdd<<<16,64>>>(a, b, c, dim)` and `VecAdd<<<1,1024>>>(a, b, c, dim)` has no difference. Now consider, if we can let threads in each block, share a part of memory, then can it reduce the time cost by memory miss? Have a look at the following function

```
#define BLOCKSIZE 64
__global__ void sharedMemVecAdd(int *a, int *b, int *c, long int dim){
    // block partition
    int block_start_idx  = dim / gridDim.x * blockIdx.x;
```

```
    int block_end_idx    = dim / gridDim.x * (blockIdx.x + 1);
    if (blockIdx.x+1==gridDim.x) block_end_idx = dim;
    int total_task       = block_end_idx - block_start_idx;
    // shared memory partition
    int num_iter = (total_task + BLOCK_SIZE - 1) / BLOCK_SIZE;
    // block-wise shared memory
    __shared__ int a_t[BLOCK_SIZE*2];
    __shared__ int b_t[BLOCK_SIZE];
    __shared__ int c_t[BLOCK_SIZE];
    __syncthreads();

    // main program
    for (int i = 0; i < num_iter; i++){
    if (threadIdx.x+i*BLOCK_SIZE < block_end_idx){
        // thread
        // copy data
        a_t[threadIdx.x] = a[block_start_idx + threadIdx.x + BLOCK_SIZE*i];
        b_t[threadIdx.x] = b[block_start_idx + threadIdx.x + BLOCK_SIZE*i];
        __syncthreads();

        // vector add
        c_t[threadIdx.x] = a_t[threadIdx.x] + b_t[threadIdx.x];


        // copy data back
        c[block_start_idx + threadIdx.x + BLOCK_SIZE*i] = c_t[threadIdx.x];
        __syncthreads();
    }}
}
```

One should convince himself that `sharedMemVecAdd<<<16,BLOCKSIZE>>>(a, b, c, dim)` do the exact same work as `VecAdd`. So what is the difference here? In each block, CUDA will create a shared memory, that is a fast memory accessible by ALL threads within this block. During the computation, the block will first read a memory block, then perform computation; after all threads finish the computation, the threads will write data back to the global memory.

4 Conclusion

In conclusion, four parallel computing schemes for n-body simulation are implemented and their performances are evaluated. For large, ignoring the precision, one may use GPU to accelerate the calculation.

A Supplementary figures

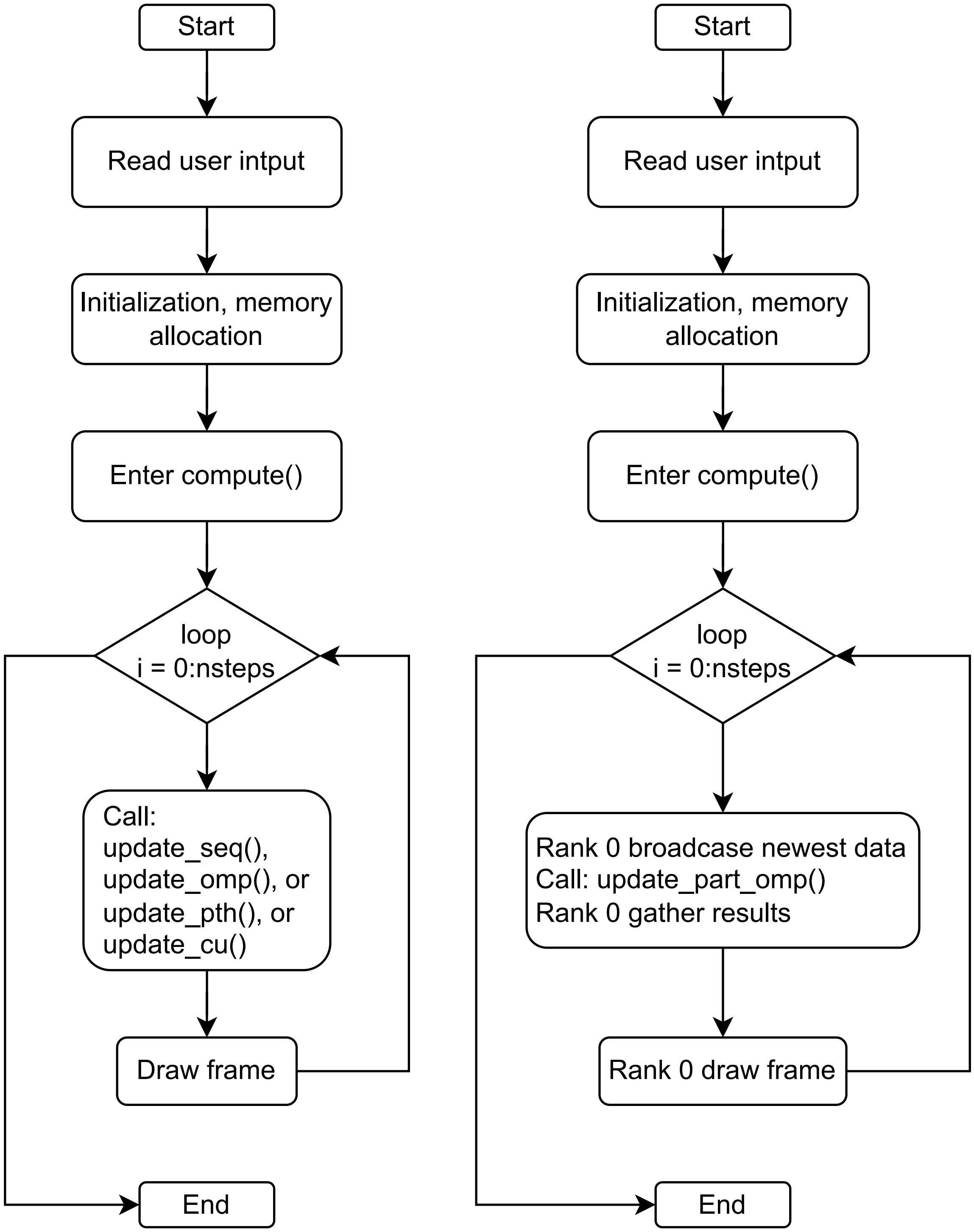

Figure A.1: Program flowchart

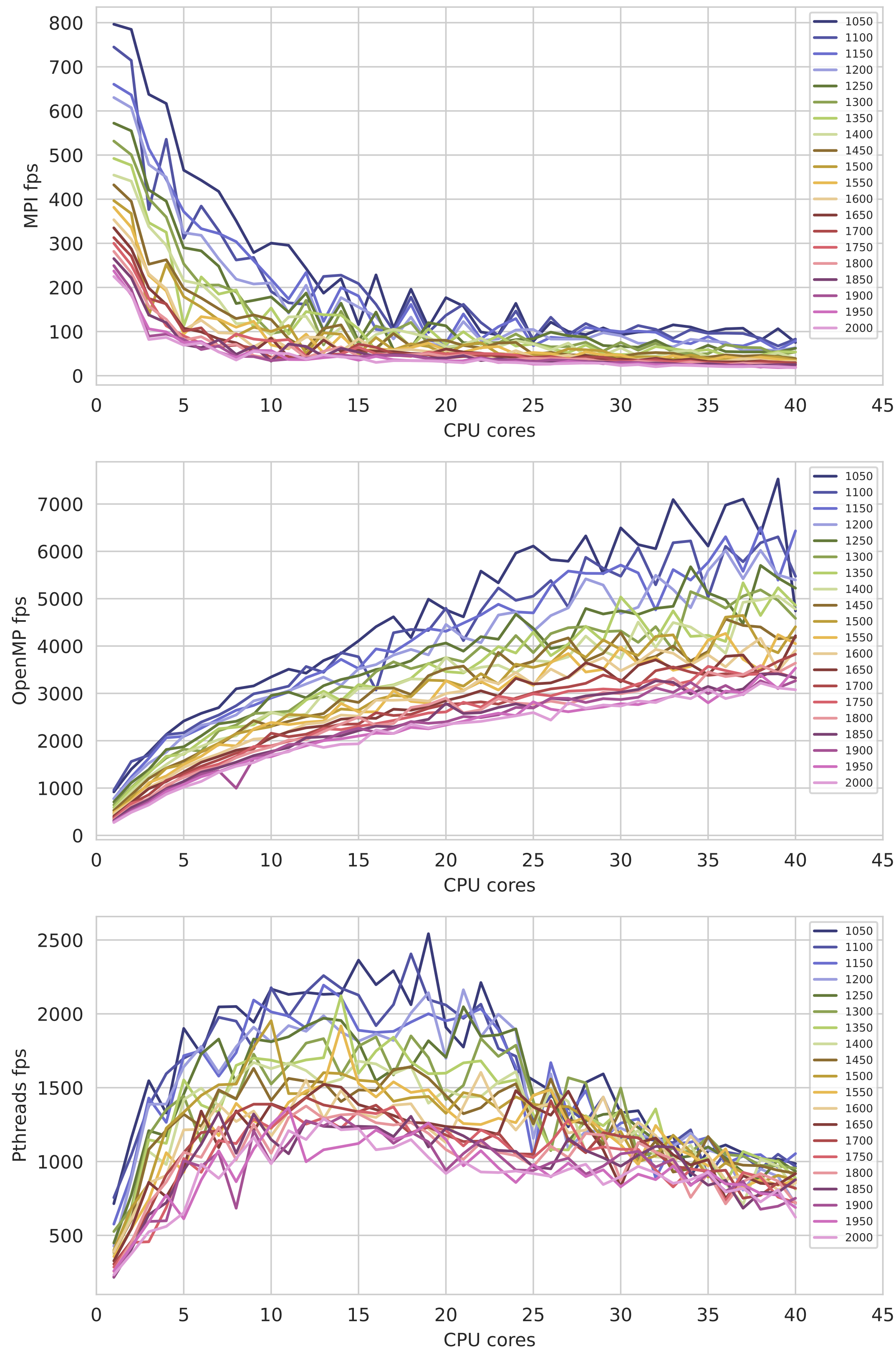

Figure A.2: fps vs the number of threads/processes plot.

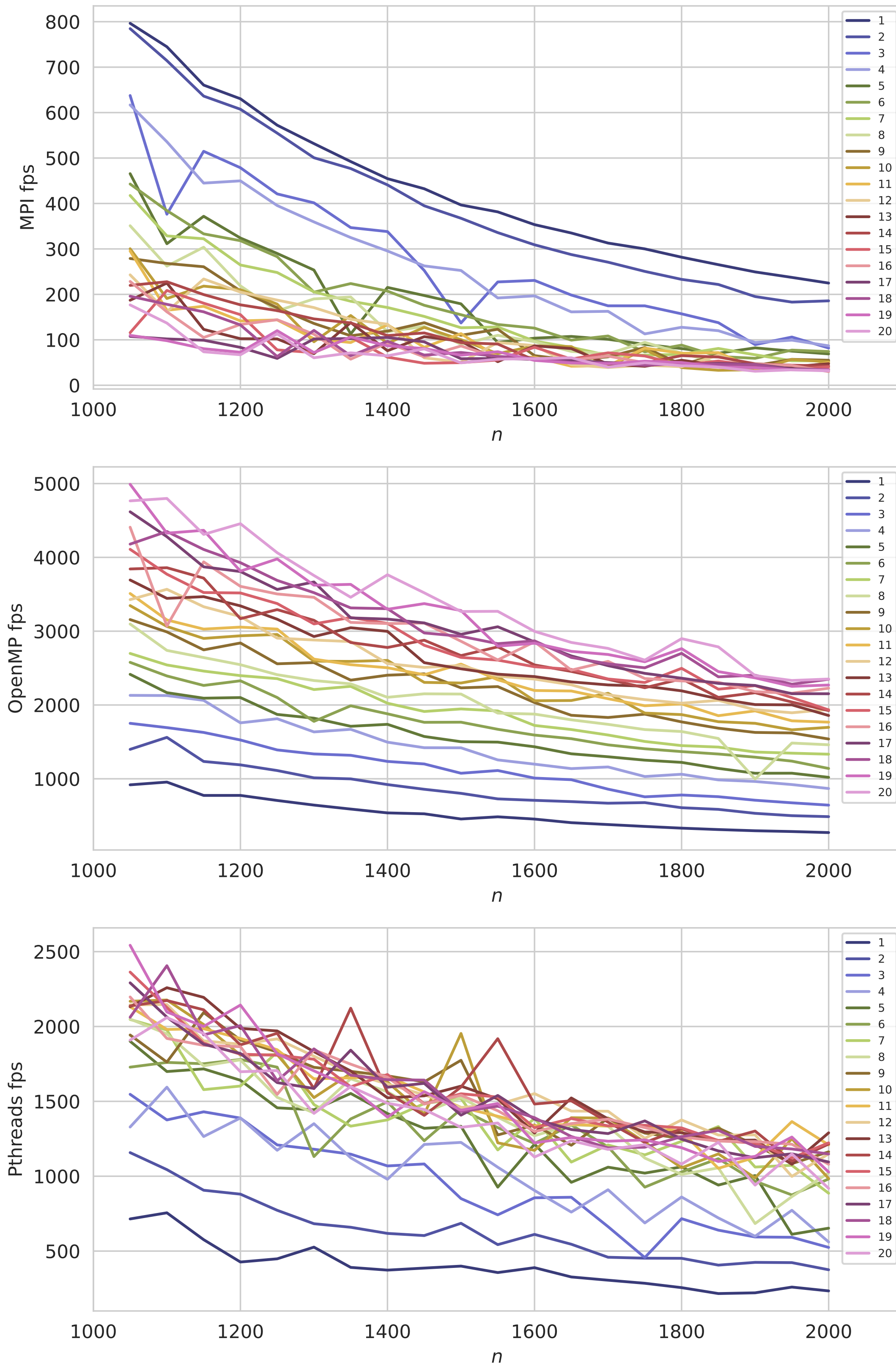

Figure A.3: fps vs the number of threads/processes plot.

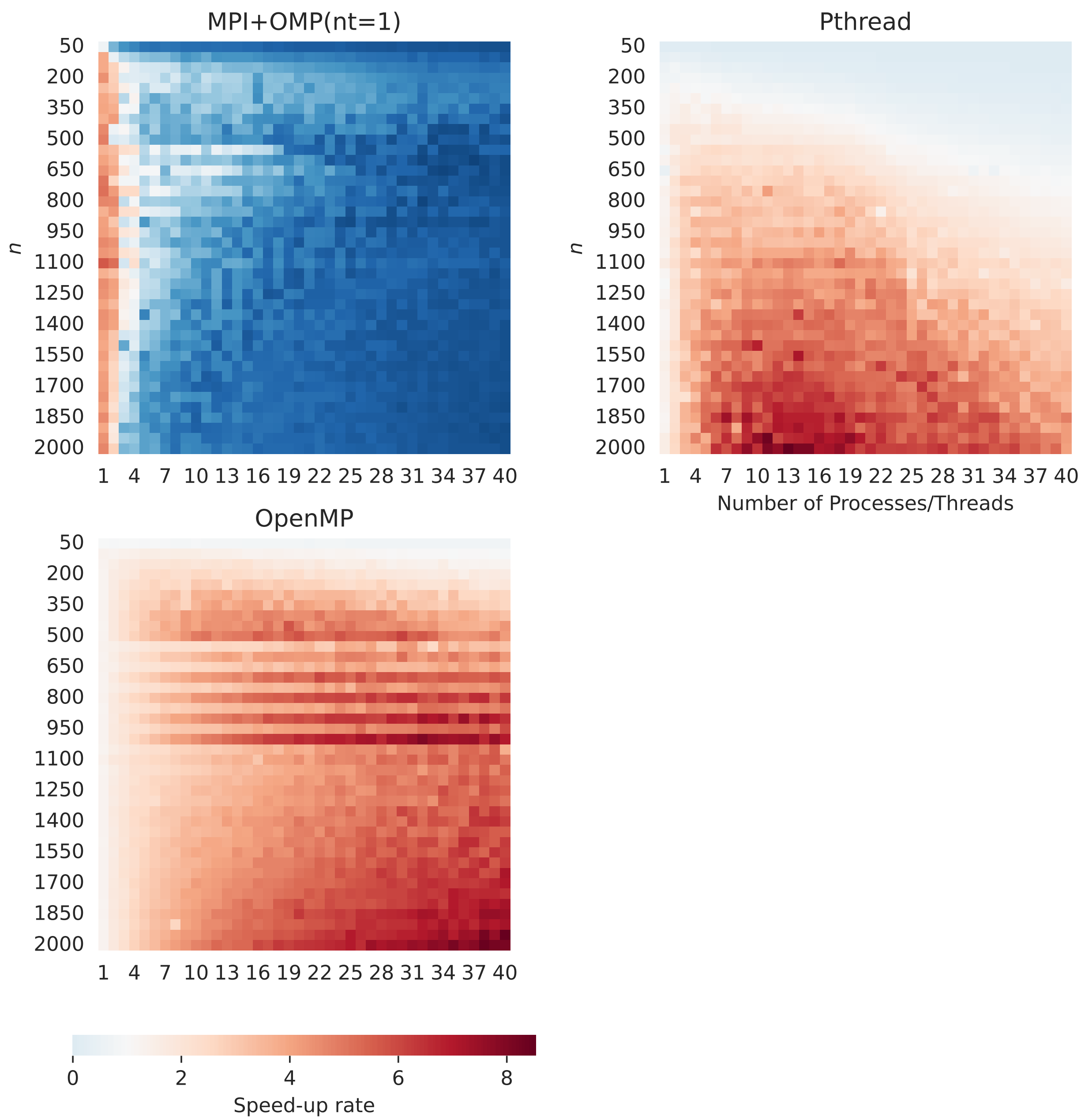

Figure A.4: Speed-up rate for multi-process/thread schemes.

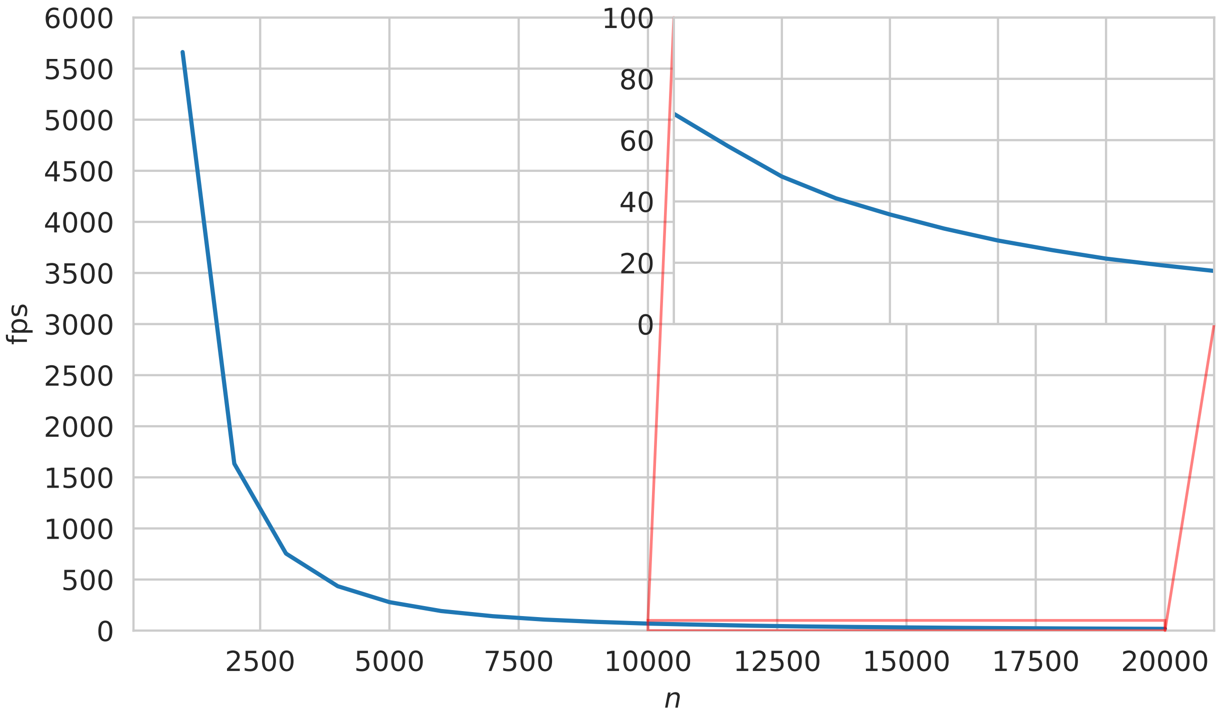

Figure A.5: CUDA fps vs n plot.

B Source code

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.20)
project(hw03 LANGUAGES CXX CUDA)

# set output path
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/lib)
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/lib)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)

# set include libraires
include_directories(src)

set(CMAKE_CXX_STANDARD 11)

# add src folder
add_subdirectory(src)
```

src/CMakeLists.txt

```
find_package(MPI REQUIRED)
find_package(CUDA REQUIRED)
find_package(Threads REQUIRED)
find_package(OpenMP REQUIRED)

# options
# gui option
option(GUI ”OPENGL Rendering” OFF)

# omp flags
set(CMAKE_CXX_FLAGS ”${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}”)

# libraries
add_library(cudalib cudalib.cu)
set(THREADS_PREFER_PTHREAD_FLAG ON)
include_directories(
    ${MPI_INCLUDE_PATH}
    ${CUDA_INCLUDE_DIRS}
)
link_libraries(
    ${MPI_LIBRARIES}
    ${CUDA_LIBRAIRES}
    cudalib
)


# targets & libs
add_executable(main.seq main.cpp)
add_executable(main.omp main.cpp)
add_executable(main.pth main.cpp)
add_executable(main.cu main.cpp)
add_executable(main.mpi main.mpi.cpp)
target_compile_definitions(main.omp PUBLIC OMP)
target_compile_definitions(main.pth PUBLIC PTH)
target_compile_definitions(main.cu PUBLIC CUDA)

# opengl & glut
if(GUI)
    find_package(OpenGL REQUIRED)
    find_package(GLUT REQUIRED)
    include_directories(${OPENGL_INCLUDE_DIRS} ${GLUT_DINCLUDE_DIRS})
    link_libraries(${OPENGL_LIBRARIES} ${GLUT_LIBRARIES})
    add_executable(main.seq.gui main.cpp)
    add_executable(main.omp.gui main.cpp)
```

```
    add_executable(main.pth.gui main.cpp)
    add_executable(main.cu.gui main.cpp)
    add_executable(main.mpi.gui main.mpi.cpp)
    target_compile_definitions(main.seq.gui PUBLIC GUI)
    target_compile_definitions(main.omp.gui PUBLIC GUI OMP)
    target_compile_definitions(main.pth.gui PUBLIC GUI PTH)
    target_compile_definitions(main.cu.gui PUBLIC GUI CUDA)
    target_compile_definitions(main.mpi.gui PUBLIC GUI)
endif()
```

src/main.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <memory.h>
#include <chrono>
#include "utils.h"
#include "utils.cuh"
#ifdef GUI
#include "gui.h"
#endif
#include "const.h"
#include <thread>

void compute(){
    // start timing
    auto t0 = std::chrono::high_resolution_clock::now();
    auto t1 = std::chrono::high_resolution_clock::now();
    auto t2 = std::chrono::high_resolution_clock::now();
    double t;
    for (int s = 0; s < nsteps; s++){
        // main compute program
        #ifdef OMP
        update_omp(&temp_arr, &temp_arr0, fire_arr, x_arr, y_arr, DIM, T_fire);
        #elif PTH
        update_pth(&temp_arr, &temp_arr0, fire_arr, x_arr, y_arr, DIM, T_fire,
            thread_arr, args_arr, nt);
        #elif CUDA
        update_cu(temp_arr0);
        #ifdef GUI
        copy_cu(temp_arr0);
        #endif
        #else
        update_seq(&temp_arr, &temp_arr0, fire_arr, x_arr, y_arr, DIM, T_fire);
        #endif

        // calculating fps
        int step = 60;
        if (s%step==0 && s%(step*2)!=0) t1 = std::chrono::high_resolution_clock::now();
        else if (s%(step*2)==0 && s!=0) {
            t2 = std::chrono::high_resolution_clock::now();
            t = std::chrono::duration_cast<std::chrono::duration<double>>(t2-t1).count()
                ;
            printf("fps: %f frame/s\n", step/t);
        }

        #ifdef GUI
        #ifdef OMP
        data2pix_omp(temp_arr0, pix, DIM, RES, T_bdy, T_fire);
        #else
        data2pix(temp_arr0, pix, DIM, RES, T_bdy, T_fire);
        #endif
        glClear(GL_COLOR_BUFFER_BIT);
        glDrawPixels(RES, RES, GL_RGB, GL_UNSIGNED_BYTE, pix);
        glFlush();
        glutSwapBuffers();
```

```
        // glFinish();
        // std::this_thread::sleep_for(std::chrono::milliseconds(100));
        #endif
    }

    // record data
    if (record==1){
        t2 = std::chrono::high_resolution_clock::now();
        t = std::chrono::duration_cast<std::chrono::duration<double>>(t2-t0).count();
        double fps = nsteps / t;
        runtime_record(type, DIM, size, fps);
    }
}

int main(int argc, char *argv[]){
    // parse argument
    char buff[200];
    for (int i = 0; i < argc; i++){
        strcpy(buff, argv[i]);
        if (strcmp(buff, ”--dim”)==0){
            std::string num(argv[i+1]);
            DIM = std::stoi(num);
        }
        if (strcmp(buff, ”-nt”)==0){
            std::string num(argv[i+1]);
            nt = std::stoi(num);
        }
        if (strcmp(buff, ”--nsteps”)==0){
            std::string num(argv[i+1]);
            nsteps = std::stof(num);
        }
        if (strcmp(buff, ”--record”)==0){
            std::string num(argv[i+1]);
            record = std::stoi(num);
        }
        if (strcmp(buff, ”--Tx”)==0){
            std::string num(argv[i+1]);
            Tx = std::stoi(num);
        }
        if (strcmp(buff, ”--Ty”)==0){
            std::string num(argv[i+1]);
            Ty = std::stoi(num);
        }
    }

    // print info
    print_info(DIM, nsteps);

    // initialization
    temp_arr = (float *)malloc(sizeof(float)*DIM*DIM);
    temp_arr0 = (float *)malloc(sizeof(float)*DIM*DIM);
    fire_arr = (bool *)malloc(sizeof(bool)*DIM*DIM);
    x_arr   = (float *)malloc(sizeof(float)*DIM);
    y_arr   = (float *)malloc(sizeof(float)*DIM);
    #ifdef GUI
    pix = (GLubyte *)malloc(sizeof(GLubyte)*RES*RES*3);
    #endif

    // assign mesh
    for (int i = 0; i < DIM; i++){
        x_arr[i] = (xmax-xmin) * i/DIM + xmin;
        y_arr[i] = (ymax-ymin) * i/DIM + ymin;
    }
    // assign temperature
    for (int i = 0; i < DIM; i++){
    for (int j = 0; j < DIM; j++){
        float x = x_arr[i];
```

```
            float y = y_arr[j];
            temp_arr[i*DIM+j] = T_bdy;
            fire_arr[i*DIM+j] = false;
            if (is_fire(x, y)){
                temp_arr[i*DIM+j] = T_fire;
                fire_arr[i*DIM+j] = true;
            }
        }}
        memcpy(temp_arr0, temp_arr, sizeof(float)*DIM*DIM);

        #ifdef OMP
        strcpy(type, "omp");
        omp_set_num_threads(nt);
        size = nt;
        #elif PTH
        strcpy(type, "pth");
        thread_arr = (pthread_t *)malloc(sizeof(pthread_t)*nt);
        args_arr = (PthArgs *)malloc(sizeof(PthArgs)*nt);
        size = nt;
        #elif CUDA
        strcpy(type, "cuda");
        initialize_cu(temp_arr, temp_arr0, fire_arr, x_arr, y_arr, DIM, T_fire,
            Tx, Ty);
        #else
        strcpy(type, "seq");
        size = 1;
        #endif

        // main program
        #ifdef GUI
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
        glutInitWindowPosition(0, 0);
        glutInitWindowSize(RES, RES);
        glutCreateWindow("Heat Distribution");
        glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        // glutDisplayFunc(&compute);
        gluOrtho2D(xmin, xmax, ymin, ymax);
        glutSetOption( GLUT_ACTION_ON_WINDOW_CLOSE, GLUT_ACTION_GLUTMAINLOOP_RETURNS);
        // glutMainLoop();
        #endif

        compute();

        // finalization
        free(temp_arr);
        free(temp_arr0);
        free(fire_arr);
        free(x_arr);
        free(y_arr);

        #ifdef PTH
        free(args_arr);
        free(thread_arr);
        #elif CUDA
        finalize_cu();
        #else
        #endif

        return 0;
}
```

src/main.mpi.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
```

```
#include <memory.h>
#include <chrono>
#include ”utils.h”
#ifdef GUI
#include ”gui.h”
#endif
#include ”const.h”

void compute(){
    // running type buffer
    strcpy(type, ”mpi”);
    // start timing
    auto t0 = std::chrono::high_resolution_clock::now();
    auto t1 = std::chrono::high_resolution_clock::now();
    auto t2 = std::chrono::high_resolution_clock::now();
    double t;
    // mpi computing parameters
    int start_idx, end_idx;
    int jobsize = DIM / size;
    partition(DIM, size, rank, &start_idx, &end_idx);
    for (int s = 0; s < nsteps; s++){
        // transfer data
        MPI_Bcast(temp_arr0, DIM*DIM, MPI_FLOAT, 0, MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);

        // main compute program
        update_omp_part(&temp_arr, &temp_arr0, fire_arr, x_arr, y_arr, DIM, T_fire,
            start_idx, end_idx);
        MPI_Barrier(MPI_COMM_WORLD);

        // transfer data
        if (rank==0) MPI_Gather(MPI_IN_PLACE, jobsize*DIM, MPI_FLOAT, temp_arr0+
            start_idx*DIM,
            jobsize*DIM, MPI_FLOAT, 0, MPI_COMM_WORLD);
        else MPI_Gather(temp_arr0+start_idx*DIM, jobsize*DIM, MPI_FLOAT, temp_arr0,
            jobsize*DIM,
            MPI_FLOAT, 0, MPI_COMM_WORLD);
        // solve tail case
        if (DIM%jobsize!=0) {
            if (rank==0){
                MPI_Recv(temp_arr0+(DIM/size*size)*DIM, (DIM%jobsize)*DIM, MPI_FLOAT,
                    size-1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
            else if (rank+1==size){
                MPI_Send(temp_arr0+(DIM/size*size)*DIM, (DIM%jobsize)*DIM, MPI_FLOAT, 0,
                     1, MPI_COMM_WORLD);
            }
        }
        MPI_Barrier(MPI_COMM_WORLD);

        // calculate fps
        int step = 60;
        if (s%step==0 && s%(step*2)!=0) t1 = std::chrono::high_resolution_clock::now();
        else if (s%(step*2)==0 && s!=0) {
            t2 = std::chrono::high_resolution_clock::now();
            t = std::chrono::duration_cast<std::chrono::duration<double>>(t2-t1).count()
                ;
            if (rank==0) printf(”fps: %f frame/s\n”, step/t);
        }

        #ifdef GUI
        if (rank==0){
            data2pix_omp(temp_arr0, pix, DIM, RES, T_bdy, T_fire);
            glClear(GL_COLOR_BUFFER_BIT);
            glDrawPixels(RES, RES, GL_RGB, GL_UNSIGNED_BYTE, pix);
            glFlush();
            glutSwapBuffers();
```

```
        }
        #endif
    }

    // record data
    if (rank==0 && record==1){
        t2 = std::chrono::high_resolution_clock::now();
        t = std::chrono::duration_cast<std::chrono::duration<double>>(t2-t0).count();
        double fps = nsteps / t;
        runtime_record(type, DIM, size, fps);
    }
}

int main(int argc, char *argv[]){
    // mpi initialize
    MPI_Init(NULL, NULL);
    // fetch size and rank
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // parse argument
    char buff[200];
    for (int i = 0; i < argc; i++){
        strcpy(buff, argv[i]);
        if (strcmp(buff, "--dim")==0){
            std::string num(argv[i+1]);
            DIM = std::stoi(num);
        }
        if (strcmp(buff, "-nt")==0){
            std::string num(argv[i+1]);
            nt = std::stoi(num);
        }
        if (strcmp(buff, "--nsteps")==0){
            std::string num(argv[i+1]);
            nsteps = std::stof(num);
        }
        if (strcmp(buff, "--record")==0){
            std::string num(argv[i+1]);
            record = std::stoi(num);
        }
        if (strcmp(buff, "--Tx")==0){
            std::string num(argv[i+1]);
            Tx = std::stoi(num);
        }
        if (strcmp(buff, "--Ty")==0){
            std::string num(argv[i+1]);
            Ty = std::stoi(num);
        }
    }

    // omp initialize
    omp_set_num_threads(nt);


    // print info
    if (rank==0) print_info(DIM, nsteps);

    // initialization
    temp_arr = (float *)malloc(sizeof(float)*DIM*DIM);
    temp_arr0 = (float *)malloc(sizeof(float)*DIM*DIM);
    fire_arr = (bool *)malloc(sizeof(bool)*DIM*DIM);
    x_arr   = (float *)malloc(sizeof(float)*DIM);
    y_arr   = (float *)malloc(sizeof(float)*DIM);
    #ifdef GUI
    if (rank==0) pix = (GLubyte *)malloc(sizeof(GLubyte)*RES*RES*3);
    #endif

```

```
    // assign mesh
    for (int i = 0; i < DIM; i++){
        x_arr[i] = (xmax-xmin) * i/DIM + xmin;
        y_arr[i] = (ymax-ymin) * i/DIM + ymin;
    }
    // assign temperature
    for (int i = 0; i < DIM; i++){
    for (int j = 0; j < DIM; j++){
        float x = x_arr[i];
        float y = y_arr[j];
        temp_arr[i*DIM+j] = T_bdy;
        fire_arr[i*DIM+j] = false;
        if (is_fire(x, y)){
            temp_arr[i*DIM+j] = T_fire;
            fire_arr[i*DIM+j] = true;
        }
    }}
    memcpy(temp_arr0, temp_arr, sizeof(float)*DIM*DIM);

    // main program
    #ifdef GUI
    if (rank==0){
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
        glutInitWindowPosition(0, 0);
        glutInitWindowSize(RES, RES);
        glutCreateWindow(”Heat Distribution”);
        glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        gluOrtho2D(xmin, xmax, ymin, ymax);
        glutSetOption( GLUT_ACTION_ON_WINDOW_CLOSE, GLUT_ACTION_GLUTMAINLOOP_RETURNS);
    }
    #endif

    compute();

    // finalization
    free(temp_arr);
    free(temp_arr0);
    free(fire_arr);
    free(x_arr);
    free(y_arr);

    #ifdef GUI
    if (rank==0) free(pix);
    #endif

    return 0;
}
```

src/cudalib.cu

```
#include ”utils.cuh”
#include ”const.cuh”
#define BLOCK_SIZE 256

#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort=true)
{
   if (code != cudaSuccess)
   {
      fprintf(stderr,”GPUassert: %s %s %d\n”, cudaGetErrorString(code), file, line);
      if (abort) exit(code);
   }
}

__device__ void partition_d(int nsteps, int size, int idx, int *start_ptr, int *end_ptr)
    {
```

```
    *start_ptr = nsteps / size * idx;
    *end_ptr = nsteps / size * (idx+1);
    if (idx+1==size) *end_ptr = nsteps;
}

__global__ void print_arr_cu(float *arr, int dim){
    for (int i = 0; i < dim; i++){
        printf(”%f ”, arr[i]);
    }
    printf(”\n”);
}

__device__ void print_arr_d(float *arr, int dim){
    for (int i = 0; i < dim; i++){
        printf(”%f ”, arr[i]);
    }
    printf(”\n”);
}

void initialize_cu(float *temp_arr, float *temp_arr0, bool *fire_arr,
    float *x_arr, float *y_arr, int DIM, float T_fire, int Tx, int Ty){
    printf(”CUDA initialization\n”);
    // cuda parameters
    DIM_d = DIM;
    T_fire_d = T_fire;
    Tx_d = Tx;
    Ty_d = Ty;
    // cuda memory allocation
    gpuErrchk( cudaMalloc((void **)&temp_arr_d, sizeof(float)*DIM*DIM) );
    gpuErrchk( cudaMalloc((void **)&temp_arr0_d, sizeof(float)*DIM*DIM) );
    gpuErrchk( cudaMalloc((void **)&fire_arr_d, sizeof(bool)*DIM*DIM) );
    // cuda memory copy
    gpuErrchk( cudaMemcpy(temp_arr_d, temp_arr, sizeof(float)*DIM*DIM,
        cudaMemcpyHostToDevice) );
    gpuErrchk( cudaMemcpy(temp_arr0_d, temp_arr0, sizeof(float)*DIM*DIM,
        cudaMemcpyHostToDevice) );
    gpuErrchk( cudaMemcpy(fire_arr_d, fire_arr, sizeof(bool)*DIM*DIM,
        cudaMemcpyHostToDevice) );
    // synchronize
    cudaDeviceSynchronize();
}

void finalize_cu(){
    printf(”CUDA finalization\n”);
    // cuda free
    gpuErrchk( cudaFree(temp_arr_d) );
    gpuErrchk( cudaFree(temp_arr0_d) );
    gpuErrchk( cudaFree(fire_arr_d) );
    // synchronize
    cudaDeviceSynchronize();
}

__global__ void update_cu_callee(float *temp_arr, float *temp_arr0, bool *fire_arr,
    float *x_arr, float *y_arr, int DIM, float T_fire){
    int start_idx, end_idx;
    int size = blockDim.x * gridDim.x;
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    partition_d(DIM-2, size, idx, &start_idx, &end_idx);
    for (int i = start_idx+1; i < end_idx+1; i++){
    for (int j = 1; j < DIM-1; j++){
        float xw, xa, xs, xd; // w: up; a: left; s: down; d: right
        xw = temp_arr0[i*DIM+j+1];
        xa = temp_arr0[(i-1)*DIM+j];
        xs = temp_arr0[i*DIM+j-1];
        xd = temp_arr0[(i+1)*DIM+j];
        temp_arr[i*DIM+j] = (xw + xa + xs + xd) / 4;
        if (fire_arr[i*DIM+j])
```

```
            temp_arr[i*DIM+j] = T_fire;
    }}
}

__global__ void update_cu_callee_shared(float *temp_arr, float *temp_arr0, bool *fire_arr,
    float *x_arr, float *y_arr, int DIM, float T_fire){
    // block partition
    int block_start_idx, block_end_idx;
    int size = gridDim.x;
    int idx = blockIdx.x;
    partition_d(DIM, size, idx, &block_start_idx, &block_end_idx);
    // block-size shared memory
    __shared__ float temp_u[BLOCK_SIZE]; // upper
    __shared__ float temp_c[BLOCK_SIZE]; // current
    __shared__ float temp_l[BLOCK_SIZE]; // lower
    __shared__ float temp_r[BLOCK_SIZE]; // record
    __shared__ bool  fire_c[BLOCK_SIZE]; // current fire
    // tail case
    float t_l, t_r;
    // pre-initialize data
    // main loop
    for (int i = 1; i < DIM-1; i += BLOCK_SIZE){
    for (int j = block_start_idx; j < block_end_idx; j++){
    if (j!=0 && j!=DIM-1){
        // load data
        if (i+threadIdx.x < DIM){
            temp_u[threadIdx.x] = temp_arr0[(j+1)*DIM+i+threadIdx.x];
            temp_c[threadIdx.x] = temp_arr0[(j+0)*DIM+i+threadIdx.x];
            temp_l[threadIdx.x] = temp_arr0[(j-1)*DIM+i+threadIdx.x];
            fire_c[threadIdx.x] = fire_arr[(j+0)*DIM+i+threadIdx.x];
        }
        if (i+threadIdx.x<DIM-1 && i+threadIdx.x>0){
            if (threadIdx.x==0) t_l = temp_arr0[(j+0)*DIM+i+threadIdx.x-1];
            else if (threadIdx.x==BLOCK_SIZE-1) t_r = temp_arr0[(j+0)*DIM+i+threadIdx.x+1];
        }
        __syncthreads();

        // main compute program
        float xw, xa, xs, xd; // w: up; a: left; s: down; d: right
        if (i+threadIdx.x<DIM-1 && i+threadIdx.x>0){
            xw = temp_u[threadIdx.x];
            xs = temp_l[threadIdx.x];
            if (threadIdx.x==0){
                xa = t_l;
            } else xa = temp_c[threadIdx.x-1];
            if (threadIdx.x==BLOCK_SIZE-1){
                xd = t_r;
            } else xd = temp_c[threadIdx.x+1];
            temp_r[threadIdx.x] = (xw + xa + xs + xd) / 4;
            // temp_r[threadIdx.x] = temp_c[threadIdx.x];
            if (fire_c[threadIdx.x]) temp_r[threadIdx.x] = T_fire;
        }
        __syncthreads();

        // copy data back
        if (i+threadIdx.x<DIM-1 && i+threadIdx.x>0)
            temp_arr[(j+0)*DIM+i+threadIdx.x] = temp_r[threadIdx.x];
        __syncthreads();
    }}}
}

__global__ void foo(float *arr, int DIM){
    for (int i = 0; i < DIM; i++)
        arr[i] = 0;
}
```

```

void update_cu(float *temp_arr){
    // update_cu_callee<<<4,BLOCK_SIZE>>>(temp_arr_d, temp_arr0_d, fire_arr_d,
    update_cu_callee_shared<<<16,BLOCK_SIZE>>>(temp_arr_d, temp_arr0_d, fire_arr_d,
        NULL, NULL, DIM_d, T_fire_d);
    cudaDeviceSynchronize();

    // switch pointers
    float *tmp = temp_arr_d;
    temp_arr_d = temp_arr0_d;
    temp_arr0_d = tmp;

    // synchronize
    cudaDeviceSynchronize();
}

void copy_cu(float *temp_arr){
    // copy data to host
    gpuErrchk( cudaMemcpy(temp_arr, temp_arr0_d, sizeof(float)*DIM_d*DIM_d,
        cudaMemcpyDeviceToHost) );
    cudaDeviceSynchronize();
}
```

src/utils.h

```
#pragma once
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <math.h>
#include <mpi.h>
#include <omp.h>
#include <pthread.h>
#include <sys/stat.h>
#include <sys/types.h>

void print_info(int DIM, int nsteps){
    printf("Name: Haoran Sun\n");
    printf("ID:   119010271\n");
    printf("HW:   Heat Distribution\n");
    printf("Set DIM to %d, nsteps to %d\n", DIM, nsteps);
}

void partition(int nsteps, int size, int idx, int *start_ptr, int *end_ptr){
    *start_ptr = nsteps / size * idx;
    *end_ptr = nsteps / size * (idx+1);
    if (idx+1==size) *end_ptr = nsteps;
}

void print_arr(float *arr, int n){
    for (int i = 0; i < n; i++){
        printf("%10.2f  ", arr[i]);
    }
    printf("\n");
}

bool is_fire(float x, float y){
    return (x*x + y*y <= 1);
}

void update_seq(float **temp_arr_ptr, float **temp_arr0_ptr, bool *fire_arr, float *
    x_arr, float *y_arr, int DIM,
    float T_fire){
    float *temp_arr = *temp_arr_ptr;
    float *temp_arr0 = *temp_arr0_ptr;
    for (int i = 1; i < DIM-1; i++){
    for (int j = 1; j < DIM-1; j++){
```

```
        float xw, xa, xs, xd; // w: up; a: left; s: down; d: right
        xw = temp_arr0[i*DIM+j+1];
        xa = temp_arr0[(i-1)*DIM+j];
        xs = temp_arr0[i*DIM+j-1];
        xd = temp_arr0[(i+1)*DIM+j];
        temp_arr[i*DIM+j] = (xw + xa + xs + xd) / 4;
        if (fire_arr[i*DIM+j])
            temp_arr[i*DIM+j] = T_fire;
    }}
    // switch pointers
    *temp_arr_ptr = temp_arr0;
    *temp_arr0_ptr = temp_arr;
}

void update_omp(float **temp_arr_ptr, float **temp_arr0_ptr, bool *fire_arr,
    float *x_arr, float *y_arr, int DIM, float T_fire){
    float *temp_arr = *temp_arr_ptr;
    float *temp_arr0 = *temp_arr0_ptr;
    #pragma omp parallel for
    for (int i = 1; i < DIM-1; i++){
    for (int j = 1; j < DIM-1; j++){
        float xw, xa, xs, xd; // w: up; a: left; s: down; d: right
        xw = temp_arr0[i*DIM+j+1];
        xa = temp_arr0[(i-1)*DIM+j];
        xs = temp_arr0[i*DIM+j-1];
        xd = temp_arr0[(i+1)*DIM+j];
        temp_arr[i*DIM+j] = (xw + xa + xs + xd) / 4;
        if (fire_arr[i*DIM+j])
            temp_arr[i*DIM+j] = T_fire;
    }}
    // switch pointers
    *temp_arr_ptr = temp_arr0;
    *temp_arr0_ptr = temp_arr;
}

void update_omp_part(float **temp_arr_ptr, float **temp_arr0_ptr, bool *fire_arr,
    float *x_arr, float *y_arr, int DIM, float T_fire, int start_idx, int end_idx){
    float *temp_arr = *temp_arr_ptr;
    float *temp_arr0 = *temp_arr0_ptr;
    #pragma omp parallel for
    for (int i = start_idx; i < end_idx; i++){
    for (int j = 1; j < DIM-1; j++){
        if (i!=0 && i!=DIM-1){
        float xw, xa, xs, xd; // w: up; a: left; s: down; d: right
        xw = temp_arr0[i*DIM+j+1];
        xa = temp_arr0[(i-1)*DIM+j];
        xs = temp_arr0[i*DIM+j-1];
        xd = temp_arr0[(i+1)*DIM+j];
        temp_arr[i*DIM+j] = (xw + xa + xs + xd) / 4;
        if (fire_arr[i*DIM+j])
            temp_arr[i*DIM+j] = T_fire;
        }
    }}
    // switch pointers
    *temp_arr_ptr = temp_arr0;
    *temp_arr0_ptr = temp_arr;
}

typedef struct pthArgs{
    float *temp_arr;
    float *temp_arr0;
    bool *fire_arr;
    float *x_arr;
    float *y_arr;
    int DIM;
    float T_fire;
    int start_idx;
```

```
    int end_idx;
    pthread_barrier_t *barr_ptr;
} PthArgs;

void *update_pth_callee(void *vargs){
    PthArgs args = *(PthArgs *) vargs;
    float *temp_arr = args.temp_arr;
    float *temp_arr0 = args.temp_arr0;
    bool *fire_arr = args.fire_arr;
    float *x_arr = args.x_arr;
    float *y_arr = args.y_arr;
    int DIM = args.DIM;
    float T_fire = args.T_fire;
    int start_idx = args.start_idx;
    int end_idx = args.end_idx;
    for (int i = 1+start_idx; i < 1+end_idx; i++){
    for (int j = 1; j < DIM-1; j++){
        float xw, xa, xs, xd; // w: up; a: left; s: down; d: right
        xw = temp_arr0[i*DIM+j+1];
        xa = temp_arr0[(i-1)*DIM+j];
        xs = temp_arr0[i*DIM+j-1];
        xd = temp_arr0[(i+1)*DIM+j];
        temp_arr[i*DIM+j] = (xw + xa + xs + xd) / 4;
        if (fire_arr[i*DIM+j])
            temp_arr[i*DIM+j] = T_fire;
    }}

    return NULL;
}

void update_pth(float **temp_arr_ptr, float **temp_arr0_ptr, bool *fire_arr, float *x_arr, float *y_arr,
    int DIM, float T_fire, pthread_t *thread_arr, PthArgs *args_arr, int nt){
    float *temp_arr = *temp_arr_ptr;
    float *temp_arr0 = *temp_arr0_ptr;

    for (int i = 0; i < nt; i++){
        int start_idx, end_idx;
        partition(DIM-2, nt, i, &start_idx, &end_idx);
        args_arr[i] = (PthArgs){.temp_arr=temp_arr, .temp_arr0=temp_arr0, .fire_arr=fire_arr,
            .x_arr=x_arr, .y_arr=y_arr, .DIM=DIM, .T_fire=T_fire, .start_idx=start_idx, .end_idx=end_idx};
        pthread_create(&thread_arr[i], NULL, update_pth_callee, (void *)&args_arr[i]);
    }

    for (int i = 0; i < nt; i++) pthread_join(thread_arr[i], NULL);

    // switch array
    *temp_arr_ptr = temp_arr0;
    *temp_arr0_ptr = temp_arr;
}

void arr_check_if_identical(float *a, float *b, int dim){
    for (int i = 0; i < dim; i++){
        if (a[i]!=b[i]){
            printf("fuck\n");
            exit(1);
        }
    }
}

void runtime_record(char *jobtype, int N, int nt, double fps){
    const char *folder = "data";
    mkdir(folder, 0777);
    FILE* outfile;
    char filebuff[200];
```

```
    snprintf(filebuff, sizeof(filebuff), ”./%s/runtime_%s.txt”, folder, jobtype);
    outfile = fopen(filebuff, ”a”);
    fprintf(outfile, ”%10d %5d %10.4f\n”, N, nt, fps);
    fclose(outfile);
    printf(”Runtime added in %s.\n”, filebuff);
}
```

src/utils.cuh

```
#pragma once
#include <cuda.h>
#include <cuda_runtime.h>
#include <cuda_runtime_api.h>
#include <cuda_device_runtime_api.h>
#include <driver_types.h>


void initialize_cu(float *temp_arr, float *temp_arr0, bool *fire_arr,
    float *x_arr, float *y_arr, int DIM, float T_fire, int Tx, int Ty);
void finalize_cu();
void update_cu(float *temp_arr);
void copy_cu(float *temp_arr);
```

src/**const**.h

```
#pragma once
#include <stdio.h>
#include <stdlib.h>
#include <iostream>

// global variables
int DIM = 200;    // overall dimension
float T_bdy  = 20; // boundary temperature
float T_fire = 100; // fire temperature
float xmin = -5;
float xmax = 5;
float ymin = -5;
float ymax = 5;
float *temp_arr = NULL;
float *temp_arr0 = NULL;
bool *fire_arr = NULL;
float *x_arr = NULL;
float *y_arr = NULL;

// computing-related constants
int nsteps = 100;

// IO & runtime options
int record = 0;
int nt = 1;
char type[1000];

// pthread parameters
pthread_t *thread_arr = NULL;
PthArgs *args_arr = NULL;

// mpi parameters
int size, rank;

// cuda parameters
int Tx = 16;
int Ty = 16;
```