CSC4005 FA22 HW03

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

1 Introduction

A typical n-body simulation problem would usually involve a calculation of $n \times n$ interactions. Therefore, the complexities would be $O(n^2)$. For example, a gravitational n-body simulation is the computer simulation of particles under the influence of gravity. Also, (bio) molecular dynamics simulation, which simulates the dynamics of chemical molecules under different conditions is also a typical n-body problem. Usually, the computation of the interactions could be split into mutually independent part–which indicates that n-body problem can be massively parallelized.

In this project, a 2-D gravitational n-body simulation 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 systems contains n particles with random generated position $\mathbf{x}_i$ on a 2-D plane and their mass m_i ($\mathbf{x}_i \in \mathbb{R}^2, i = 1, \ldots, n$). The force that the particle j exerted on the particle j is

$$\mathbf{F}_{ij} = (\mathbf{x}_j - \mathbf{x}_i)\frac{Gm_i m_j}{r_{ij}^3}$$

Hence the acceleration of the ith particle is

$$\mathbf{a}_i = \sum_j \frac{\mathbf{F}_{ij}}{m_i} = \sum_j (\mathbf{x}_j - \mathbf{x}_i)\frac{Gm_j}{r_{ij}^3}$$

To update the system, the Verlet algorithm is implemented to calculate the position of particles during the time evolution.

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) - \mathbf{x}(t - \Delta t) + \mathbf{a}(t)\Delta t^2$$

The reason to choose the Verlet algorithm rather than the Euler's method which mentioned in the homework instructions is that the Verlet algorithm follows the conservation law of energy but the Euler's method doesn't.

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
```

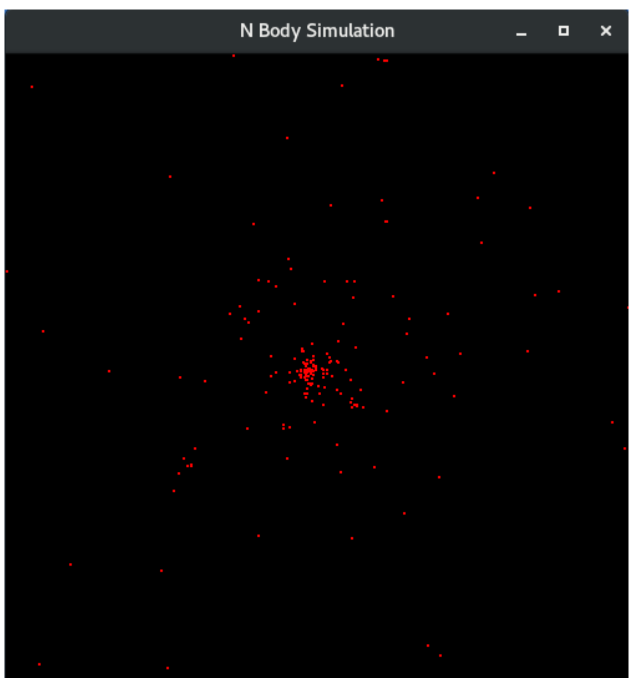

Figure 1: Sample GUI window

One can customize the running parameters such as the number of particles n and simulation steps according to the following lines. `-nt` is for number of threads, `--Tx` and `--Ty` is to set CUDA 1-D grid size and block size.

```
./build/bin/main.seq               -n 100 --nsteps 10000 --record 1
./build/bin/main.omp        -nt 10 -n 100 --nsteps 10000 --record 1
./build/bin/main.omp        -nt 10 -n 100 --nsteps 10000 --record 1
./build/bin/main.cu   --Tx 16 --Ty -n 100 --nsteps 10000 --record 1
mpirun -np 10 ./build/bin/main.mpi -n 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 20 different CPU core numbers (from 1 to 20 with increment 1, $p = 1, 2, \ldots, 20$) and 20 different n (from 50 to 1000 with increment 50), 400 cases in total were sampled for sequential, MPI, OpenMP, and Pthread programs. Test for CUDA program is implemented separately since GPU is much faster than all CPU programs and only large-scale performance will be discussed on CUDA program. 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`.

3 Result and discussion

Remark. Again, since GPU is much faster than CPU, I would discuss their performances separately. Also, the discussion will focus on large-scale cases.

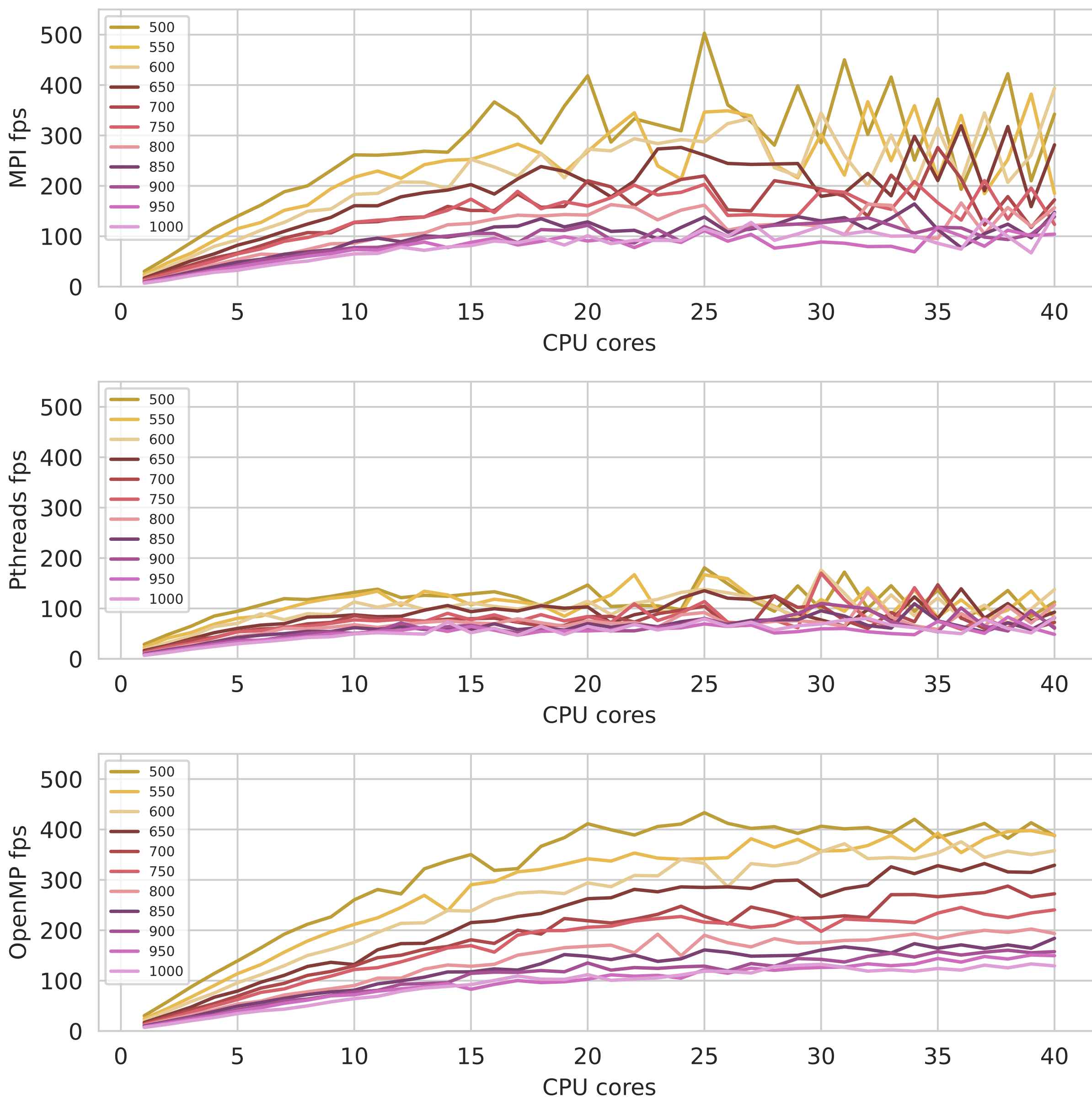

Figure 2: fps vs the number of threads/processes plot.

3.1 CPU parallelization

From Figure 2, we can know that when n ranging from 500 to 1000, MPI and OpenMP programs have similar performance when the number of processes/threads is under 20: fps steadily increases

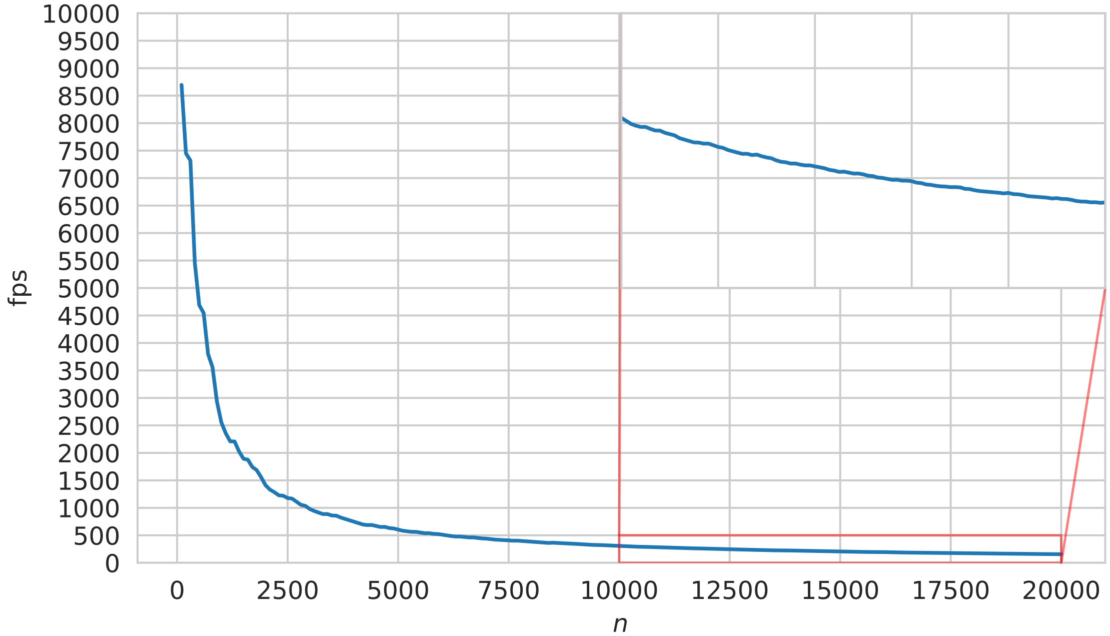

Figure 3: CUDA fps vs n plot.

with threads/processes number while decreases with n. The MPI program becomes quite unstable when the number of cores exceeds 20: the reason might be the unstable communication traffic and CPU resources. Meanwhile, the Pthread program has low and relatively constant performance. That may result from the Pthread function `compute_pth` in `src/utils.h`. In each iteration (each frame), the program will initialize `nt` threads, perform the computation parallelly and then merge these threads. Different from OpenMP which is fully optimized, the initialization and joining of threads in each iteration could be much more time-costly. To fix this issue, one may initialize threads at the start of the program, and join all threads after finishing all calculations.

The heatmap which indicate the rate of acceleration plotted in the Figure A.2 provides some direct visualization of the performances of parallel variants.

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 3 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 global (`__device__`) memory 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

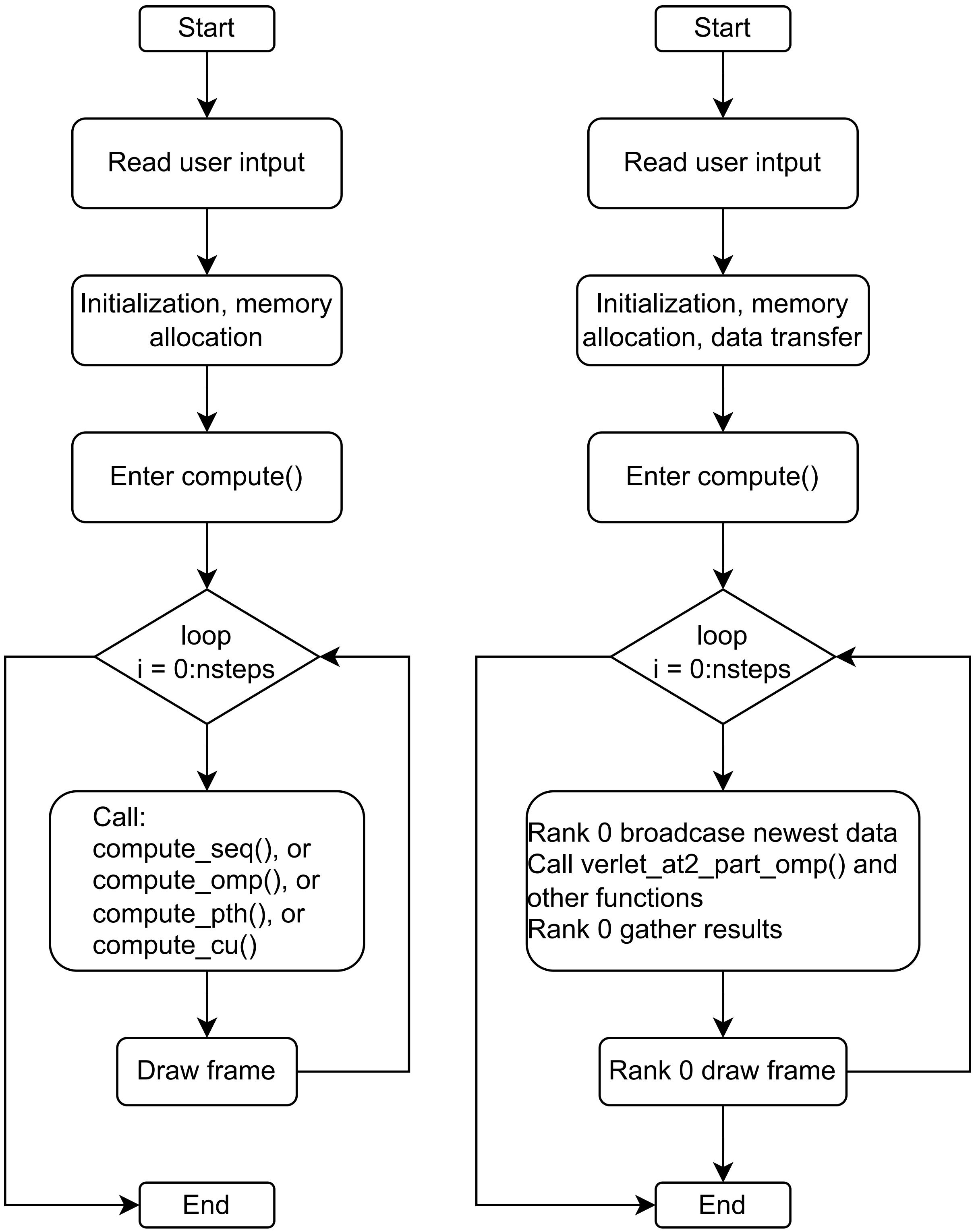

Figure A.1: Program flowchart

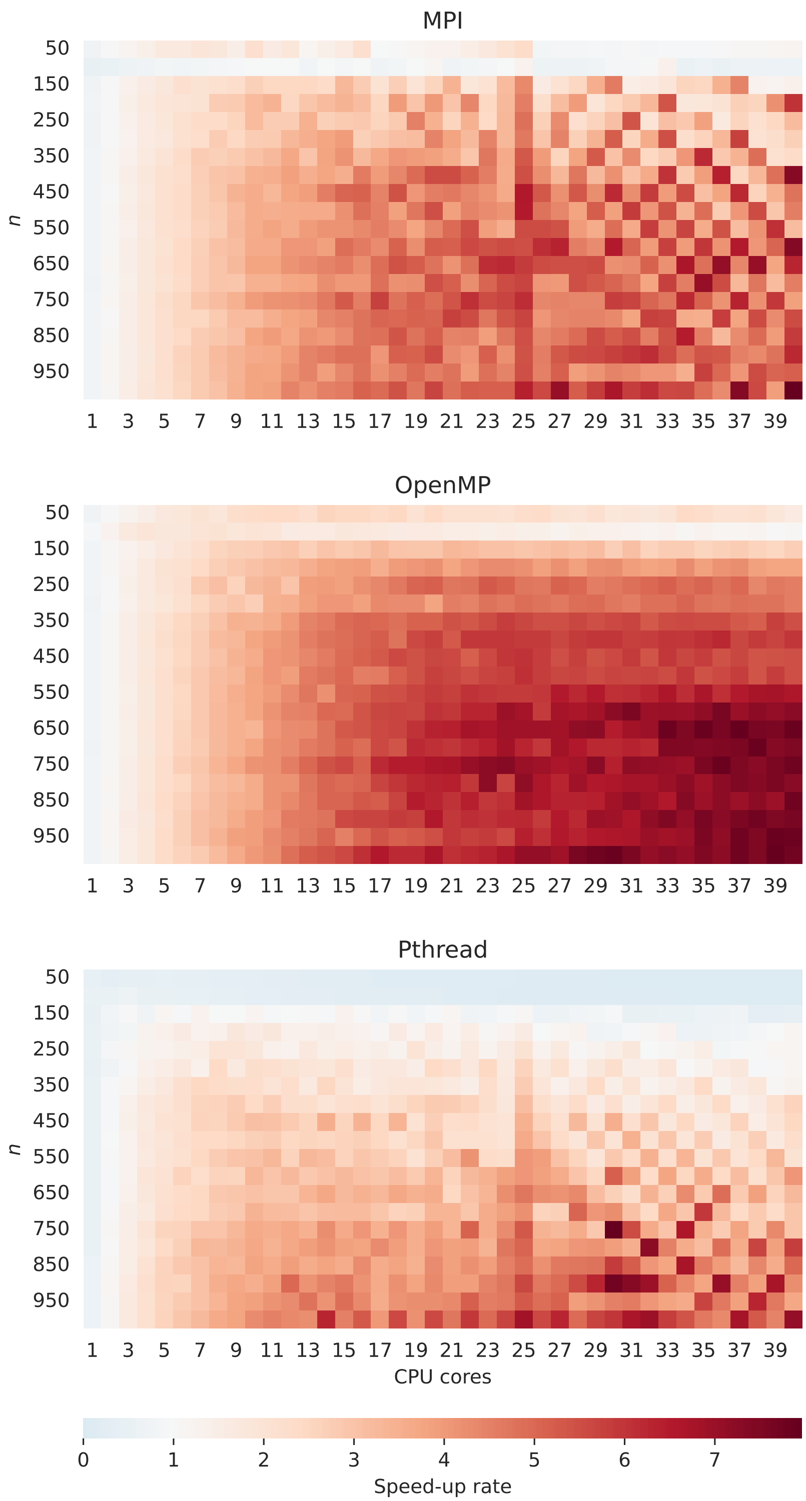

Figure A.2: CUDA fps vs n plot.

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.omp.gui main.cpp)
    add_executable(main.cu.gui main.cpp)
```

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

src/main.cpp

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

void compute(){
    // running type buffer
    char type[1000];
    // 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;
    // main program
    for (int s = 0; s < nsteps; s++){
        // verlet omp
        #ifdef OMP
        if (s==0) {
            printf("Start OpenMP version.\n");
            strcpy(type, "omp");
        }
        compute_omp(&xarr, &xarr0, dxarr, marr, N, dim, G, dt, radius);
        #elif CUDA
        if (s==0) {
            printf("Start CUDA version.\n");
            strcpy(type, "cuda");
        }
        compute_cu(xarr, nsteps, N, dim, G, dt, radius);
        #elif PTH
        if (s==0) {
            printf("Start Pthread version.\n");
            strcpy(type, "pth");
        }
        compute_pth(&xarr, &xarr0, dxarr, marr, N, dim, G, dt, radius, nt);
        #else
        if (s==0) {
            printf("Start sequential version.\n");
            strcpy(type, "seq");
        }
        compute_seq(&xarr, &xarr0, dxarr, marr, N, dim, G, dt, radius);
        #endif


        // calculating fps
        int step = 200;
        #ifdef GUI
        step = 30;
        #endif
```

```
        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);
        }

        // opengl
        #ifdef GUI
        glClear(GL_COLOR_BUFFER_BIT);
        glColor3f(1.0f, 0.0f, 0.0f);
        glPointSize(2.0f);

        // gl points
        glBegin(GL_POINTS);
        float xi;
        float yi;
        float xmin, xmax, ymin, ymax;
        for (int i = 0; i < N; i++){
            xi = xarr[i*dim+0];
            yi = xarr[i*dim+1];
            glVertex2f(xi, yi);
        }
        glEnd();

        glFlush();
        glutSwapBuffers();
        #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, N, nt, 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, ”-n”)==0){
            std::string num(argv[i+1]);
            N = std::stoi(num);
        }
        if (strcmp(buff, ”-nt”)==0){
            std::string num(argv[i+1]);
            nt = std::stoi(num);
        }
        if (strcmp(buff, ”--xmin”)==0){
            std::string num(argv[i+1]);
            xmin = std::stof(num);
        }
        if (strcmp(buff, ”--xmax”)==0){
            std::string num(argv[i+1]);
            xmax = std::stof(num);
        }
        if (strcmp(buff, ”--ymin”)==0){
            std::string num(argv[i+1]);
            ymin = std::stof(num);
        }
        if (strcmp(buff, ”--ymax”)==0){
            std::string num(argv[i+1]);
```

```
            ymax = std::stof(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 options
    #ifdef OMP
    omp_set_dynamic(0);
    omp_set_num_threads(nt);
    #endif

    // print info
    print_info(N, nsteps);

    // array allocation
    marr  = (float *)malloc(sizeof(float) * N);
    xarr  = (float *)malloc(sizeof(float) * N * dim);
    xarr0 = (float *)malloc(sizeof(float) * N * dim);
    dxarr = (float *)malloc(sizeof(float) * N * dim);

    // random generate initial condition
    random_generate(xarr, marr, N, dim);
    print_arr(xarr, 8);

    // initialization
    vec_add(xarr0, xarr0, xarr, 0, 1, N*dim);

    // cuda initialize
    #ifdef CUDA
    Tx = 16;
    Ty = 16;
    initialize_cu(marr, xarr, N, dim, Tx, Ty, xmin, xmax, ymin, ymax);
    #endif

    // start timing
    auto t1 = std::chrono::high_resolution_clock::now();
    // main program
    #ifdef GUI
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
    glutInitWindowPosition(0, 0);
    glutInitWindowSize(500, 500);
    glutCreateWindow("N Body Simulation");
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glutDisplayFunc(&compute);
    glutKeyboardFunc(&guiExit);
    gluOrtho2D(xmin, xmax, ymin, ymax);
    glutSetOption( GLUT_ACTION_ON_WINDOW_CLOSE, GLUT_ACTION_GLUTMAINLOOP_RETURNS);
    glutMainLoop();
    #else
    compute();
    // cudaDeviceSynchronize();
    #endif
```

```

    // end timing
    auto t2 = std::chrono::high_resolution_clock::now();
    double t = std::chrono::duration_cast<std::chrono::duration<double>>(t2-t1).count();

    printf("Duration: %fs\n", t);

    // free
    free(marr);
    free(xarr);
    free(xarr0);
    free(dxarr);

    #ifdef CUDA
    // cudafree
    finalize_cu();
    cudaDeviceSynchronize();
    #endif

    return 0;
}
```

src/main.mpi.cpp

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

void compute(){
    // main program
    char type[] = "mpi";
    int start_idx, end_idx;
    int jobsize = N / size;
    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;
    partition(N, size, rank, &start_idx, &end_idx);
    if (rank == 0) printf("Start MPI version.\n");
    for (int s = 0; s < nsteps; s++){
        // transfer data
        MPI_Bcast(xarr, N*dim, MPI_FLOAT, 0, MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);

        // calculate dx
        vec_assign_const(dxarr, 0, N*dim);
        verlet_at2_part_omp(dim, marr, xarr, xarr0, dxarr, dt, G, N, radius, start_idx,
            end_idx);
        // verlet_at2_part(dim, marr, xarr, xarr0, dxarr, dt, G, N, radius, start_idx,
            end_idx);
        vec_add_part(dxarr, dxarr, xarr, 1.0, 1.0, N*dim, start_idx*dim, end_idx*dim);
        vec_add_part(dxarr, dxarr, xarr0, 1.0, -1.0, N*dim, start_idx*dim, end_idx*dim);
        float *tmp = xarr;
        xarr = xarr0;
        xarr0 = tmp;
        MPI_Barrier(MPI_COMM_WORLD);
        verlet_add_part_omp(xarr, xarr0, dxarr, N, dim, xmin, xmax, ymin, ymax,
            start_idx, end_idx);
        // verlet_add_part(xarr, xarr0, dxarr, N, dim, xmin, xmax, ymin, ymax, start_idx
            , end_idx);
```

```

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

        // opengl
        if (rank==0){
            #ifdef GUI
            // calculating fps
            int step = 200;
            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);
            }
            glClear(GL_COLOR_BUFFER_BIT);
            glColor3f(1.0f, 0.0f, 0.0f);
            glPointSize(2.0f);

            // gl points
            glBegin(GL_POINTS);
            float xi;
            float yi;
            float xmin, xmax, ymin, ymax;
            for (int i = 0; i < N; i++){
                xi = xarr[i*dim+0];
                yi = xarr[i*dim+1];
                glVertex2f(xi, yi);
            }
            glEnd();

            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, N, size, fps);
    }
}

int main(int argc, char* argv[]){
    // mpi initializatio
    MPI_Init(NULL, NULL);
```

```
    // fetch size and rank
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // parse arguments
    char buff[200];
    for (int i = 0; i < argc; i++){
        strcpy(buff, argv[i]);
        if (strcmp(buff, "-n")==0){
            std::string num(argv[i+1]);
            N = std::stoi(num);
        }
        if (strcmp(buff, "-nt")==0){
            std::string num(argv[i+1]);
            nt = std::stoi(num);
        }
        if (strcmp(buff, "--xmin")==0){
            std::string num(argv[i+1]);
            xmin = std::stof(num);
        }
        if (strcmp(buff, "--xmax")==0){
            std::string num(argv[i+1]);
            xmax = std::stof(num);
        }
        if (strcmp(buff, "--ymin")==0){
            std::string num(argv[i+1]);
            ymin = std::stof(num);
        }
        if (strcmp(buff, "--ymax")==0){
            std::string num(argv[i+1]);
            ymax = std::stof(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);
        }
    }

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

    // initialization
    // array allocation
    marr      = (float *)malloc(sizeof(float) * N);
    xarr      = (float *)malloc(sizeof(float) * N * dim);
    xarr0     = (float *)malloc(sizeof(float) * N * dim);
    dxarr     = (float *)malloc(sizeof(float) * N * dim);
    // random generate initial condition
    if (rank == 0){
        random_generate(xarr, marr, N, dim);
        // initialize xarr0
        vec_add(xarr0, xarr0, xarr, 0, 1, N*dim);
    }
    // transfer data
    MPI_Bcast(marr, N, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Bcast(xarr, N*dim, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Bcast(xarr0, N*dim, MPI_FLOAT, 0, MPI_COMM_WORLD);

    // omp options
    omp_set_dynamic(0);
    omp_set_num_threads(nt);

    // main computing program
```

```
    if (rank==0){
        #ifdef GUI
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
        glutInitWindowPosition(0, 0);
        glutInitWindowSize(500, 500);
        glutCreateWindow("N Body Simulation");
        glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        glutDisplayFunc(&compute);
        glutKeyboardFunc(&guiExit);
        gluOrtho2D(xmin, xmax, ymin, ymax);
        glutSetOption( GLUT_ACTION_ON_WINDOW_CLOSE, GLUT_ACTION_GLUTMAINLOOP_RETURNS);
        glutMainLoop();
        #else
        compute();
        #endif
    }
    else {
        compute();
    }

    // mpi finalization
    MPI_Finalize();
}
```

src/cudalib.cu

```
#include "utils.cuh"
#include "const.cuh"
#define BLOCK_SIZE 1024

#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 get_xij_d(int i, int j, int dim, float *xarr, float *xij, int N){
    for (int k = 0; k < dim; k++){
        xij[k] = xarr[j*dim+k] - xarr[i*dim+k];
    }
}

__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;
}

__device__ float norm_d(float *x, int dim){
    float r = 0;
    for (int i = 0; i < dim; i++){
        r += x[i]*x[i];
    }
    r = sqrt(r);
    return r;
}

__device__ void vec_add_d(float *a, float *b, float *c,
    float fac1, float fac2, int dim){
    for (int i = 0; i < dim; i++){
        a[i] = fac1*b[i] + fac2*c[i];
```

```
    }
}


__global__ void vec_add_cu(float *a, float *b, float *c, int dim){
    int size = blockDim.x * gridDim.x;
    int idx = blockDim.x*blockIdx.x + threadIdx.x;
    int start_idx, end_idx;
    partition_d(dim, size, idx, &start_idx, &end_idx);
    for (int i = start_idx; i < end_idx; i++){
        a[i] = b[i] + c[i];
    }
}

__global__ void verlet_add_cu(float *a, float *b, float *c, int N, int dim,
    int xmin, int xmax, int ymin, int ymax){
    int size = blockDim.x * gridDim.x;
    int idx = blockDim.x*blockIdx.x + threadIdx.x;
    int start_idx, end_idx;
    partition_d(N, size, idx, &start_idx, &end_idx);
    for (int i = start_idx; i < end_idx; i++){
        float x = b[i*dim+0] + c[i*dim+0];
        float y = b[i*dim+1] + c[i*dim+1];
        if (x < xmin) x += 2 * (xmin - x);
        else if (x > xmax) x += 2 * (xmax - x);
        if (y < ymin) y += 2 * (ymin - y);
        else if (y > ymax) y += 2 * (ymax - y);
        a[i*dim+0] = x;
        a[i*dim+1] = y;
    }
}

__global__ void gather_dx_cu(float *a, float *b, float *c, int dim){
    int size = blockDim.x * gridDim.x;
    int idx = blockDim.x*blockIdx.x + threadIdx.x;
    int start_idx, end_idx;
    partition_d(dim, size, idx, &start_idx, &end_idx);
    for (int i = start_idx; i < end_idx; i++){
        a[i] += b[i] - c[i];
    }
}

__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");
}

__global__ void verlet_at2_cu(const int dim, float *marr, float *xarr, float *xarr0,
    float *dxarr, float dt, float G, int N, float cut){
    // partition
    int size = gridDim.x;
    int idx = blockIdx.x;
    int block_start_idx, block_end_idx;
    partition_d(N, size, idx, &block_start_idx, &block_end_idx);
    // if (threadIdx.x==0) printf("%d %d\n", block_start_idx, block_end_idx);
    // shared memory
    __shared__ float     marr_t[BLOCK_SIZE];
    __shared__ float xarr_l_t[BLOCK_SIZE*2];
```

```
    __shared__ float xarr_g_t[BLOCK_SIZE*2];
    __shared__ float  dxarr_t[BLOCK_SIZE*2];
    // G*dt*dt factor
    float fac = G*dt*dt;
    for (int i = block_start_idx; i < block_end_idx; i+=BLOCK_SIZE){
        // tmp variables
        float tmpx = 0.0;
        float tmpy = 0.0;
        if (i + threadIdx.x < block_end_idx){
            // get local coords
            xarr_l_t[threadIdx.x*dim+0] = xarr[i*dim+threadIdx.x*dim+0];
            xarr_l_t[threadIdx.x*dim+1] = xarr[i*dim+threadIdx.x*dim+1];
        }
        __syncthreads();
        // N loop
        for (int j = 0; j < N; j+=BLOCK_SIZE){
            if (threadIdx.x + j < N){
                marr_t[threadIdx.x] = marr[threadIdx.x+j];
                xarr_g_t[threadIdx.x*dim+0] = xarr[threadIdx.x*dim+j*dim+0];
                xarr_g_t[threadIdx.x*dim+1] = xarr[threadIdx.x*dim+j*dim+1];
            }
            __syncthreads();
            // if (blockIdx.x==0 && threadIdx.x==0 && j==0) print_arr_d(xarr_g_t, 8);
            for (int k = 0; k < BLOCK_SIZE; k++){
            if (k + j < N && threadIdx.x + j < N){
                // compute xij
                float xij0 = xarr_g_t[k*dim+0] - xarr_l_t[threadIdx.x*dim+0];
                float xij1 = xarr_g_t[k*dim+1] - xarr_l_t[threadIdx.x*dim+1];
                float rij = sqrt(xij0*xij0 + xij1*xij1);
                if (rij < cut) rij = cut;
                tmpx += xij0/(rij*rij*rij) * marr_t[k]*fac;
                tmpy += xij1/(rij*rij*rij) * marr_t[k]*fac;
            }}
            // assign value to shared memory
        }
        if (i + threadIdx.x < block_end_idx){
            // assign value back to global memory
            dxarr_t[threadIdx.x*dim+0] = tmpx;
            dxarr_t[threadIdx.x*dim+1] = tmpy;
        }
        __syncthreads();
        if (i + threadIdx.x < block_end_idx){
            dxarr[threadIdx.x*dim+i*dim+0] = dxarr_t[threadIdx.x*dim+0];
            dxarr[threadIdx.x*dim+i*dim+1] = dxarr_t[threadIdx.x*dim+1];
        }
        __syncthreads();
    }
}

// cuda initialize program
void initialize_cu(float *marr, float *xarr, int N, int dim, int Tx, int Ty,
    float xmin, float xmax, float ymin, float ymax){
    printf(”cuda initialize\n”);
    // cuda parameters
    Tx_cu = Tx;
    Ty_cu = Ty;
    xmin_d = xmin;
    xmax_d = xmax;
    ymin_d = ymin;
    ymax_d = ymax;
    // cuda memory allocation
    gpuErrchk( cudaMalloc((void **) &marr_d, sizeof(float)*N));
    gpuErrchk( cudaMalloc((void **) &xarr_d, sizeof(float)*N*dim));
    gpuErrchk( cudaMalloc((void **) &xarr0_d, sizeof(float)*N*dim));
    gpuErrchk( cudaMalloc((void **) &dxarr_d, sizeof(float)*N*dim));
    // copy
    gpuErrchk( cudaMemcpy(marr_d, marr, sizeof(float)*N, cudaMemcpyHostToDevice) );
```

```
    gpuErrchk( cudaMemcpy(xarr_d, xarr, sizeof(float)*N*dim, cudaMemcpyHostToDevice) );
    gpuErrchk( cudaMemcpy(xarr0_d, xarr, sizeof(float)*N*dim, cudaMemcpyHostToDevice) );

    // print check: passed
    // print_arr_cu<<<1,1>>>(marr_d, N);
    cudaDeviceSynchronize();
}

// switch pointers
__global__ void swap(float * &a, float * &b){
    float *tmp = a;
    a = b;
    b = tmp;
}

// verlet cuda callee
void compute_cu(float *xarr, int nsteps, int N, int dim, float G, float dt, float cut){
    // verlet cuda main program
    float *tmp;
    cudaMemset(dxarr_d, 0x00, sizeof(float)*N*dim);
    cudaDeviceSynchronize();
    verlet_at2_cu<<<32,BLOCK_SIZE>>>(dim, marr_d, xarr_d, xarr0_d, dxarr_d, dt, G, N,
        cut); // dx: acc
    cudaDeviceSynchronize();
    gather_dx_cu<<<Tx_cu,Ty_cu>>>(dxarr_d, xarr_d, xarr0_d, N*dim);
    cudaDeviceSynchronize();
    tmp = xarr_d;
    xarr_d = xarr0_d;
    xarr0_d = tmp;
    verlet_add_cu<<<Tx_cu,Ty_cu>>>(xarr_d, xarr0_d, dxarr_d, N, dim, xmin_d, xmax_d,
        ymin_d, ymax_d);

    cudaDeviceSynchronize();
    cudaMemcpy(xarr, xarr_d, sizeof(float)*N*dim, cudaMemcpyDeviceToHost);

    #ifdef GUI
    // copy x to host
    cudaMemcpy(xarr, xarr_d, sizeof(float)*N*dim, cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();
    #endif
}

// cuda finalize program
void finalize_cu(){
    // free
    printf("cuda finalize\n");
    gpuErrchk( cudaFree(marr_d) );
    gpuErrchk( cudaFree(xarr_d) );
    gpuErrchk( cudaFree(xarr0_d) );
    gpuErrchk( cudaFree(dxarr_d) );
}
```

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 N, int nsteps){
    printf("Name: Haoran Sun\n");
```

```
    printf("ID:   119010271\n");
    printf("HW:   N-Body Simulation\n");
    printf("Set N to %d, nsteps to %d\n", N, 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 map_idx_to_pair(int N, int idx, int *i_ptr, int *j_ptr){
    int work = N*(N-1) / 2;
    int tmp = (-1 + sqrt(8*idx+9)) / 2;
    int idx_ = tmp * (tmp+1) / 2 - 1;
    if (idx_ < idx) tmp += 1;
    idx_ = tmp * (tmp+1) / 2 - 1;
    *i_ptr = tmp;
    *j_ptr = tmp - 1 + idx - idx_;
    // printf("mmm %d %d\n", *i_ptr, *j_ptr);
}

float norm(float *x, int dim){
    float r = 0;
    for (int i = 0; i < dim; i++){
        r += pow(x[i], 2);
    }
    r = sqrt(r);
    return r;
}

void get_xij(int i, int j, int dim, float *xarr, float *xij, int N){
    for (int k = 0; k < dim; k++){
        xij[k] = xarr[j*dim+k] - xarr[i*dim+k];
    }
}

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

void vec_add(float *a, float *b, float *c,
             float fac1, float fac2, int dim){
    for (int i = 0; i < dim; i++){
        a[i] = fac1*b[i] + fac2*c[i];
    }
}

void vec_add_omp(float *a, float *b, float *c,
             float fac1, float fac2, int dim){
    #pragma omp parallel for
    for (int i = 0; i < dim; i++){
        a[i] = fac1*b[i] + fac2*c[i];
    }
}

void vec_add_part(float *a, float *b, float *c,
    float fac1, float fac2, int dim,
    int start_idx, int end_idx){
    for (int i = start_idx; i < end_idx; i++){
        a[i] = fac1*b[i] + fac2*c[i];
    }
}

```

```
void verlet_at2(int dim, float *marr, float *xarr, float *xarr0,
               float *dxarr, float dt, float G, int N, float cut){
    for (int idx = 0; idx < N*(N-1)/2; idx++) {
        int i, j;
        map_idx_to_pair(N, idx, &i, &j);
        // printf("%d %d\n", i, j);
        float xij[dim];
        float tmp[dim];
        float mi = marr[i];
        float mj = marr[j];
        // get xij
        get_xij(i, j, dim, xarr, xij, N);
        // compute rij
        float rij = norm(xij, dim);
        float fac = 1.0;
        if (rij < cut) {
            rij = cut;
        }
        // compute intermediate variable
        for (int k = 0; k < dim; k++){
            tmp[k] = xij[k]*G/pow(rij, 3);
        }
        // add to dx
        vec_add(dxarr+i*dim, dxarr+i*dim, tmp, 1.0, mj*dt*dt, dim);
        vec_add(dxarr+j*dim, dxarr+j*dim, tmp, 1.0, -mi*dt*dt, dim);
    }
}


void verlet_at2_omp(int dim, float *marr, float *xarr, float *xarr0,
               float *dxarr, float dt, float G, int N, float cut){
    #pragma omp parallel for
    for (int i = 0; i < N; i++){
        float tmp[dim];
        for (int j = 0; j < dim; j++) tmp[j] = 0;
        for (int j = 0; j < N; j++){
            if (j!=i){
            float xij[dim];
            float mi = marr[i];
            float mj = marr[j];
            // get xij
            get_xij(i, j, dim, xarr, xij, N);
            // compute rij
            float rij = norm(xij, dim);
            float fac = 1.0;
            if (rij < cut) {
                rij = cut;
            }
            // compute intermediate variable
            for (int k = 0; k < dim; k++){
                tmp[k] += xij[k]*G/pow(rij, 3) *mj*dt*dt;
            }
            }
        }
        vec_add(dxarr+i*dim, dxarr+i*dim, tmp, 1.0, 1.0, dim);
    }
}

void verlet_at2_part(int dim, float *marr, float *xarr, float *xarr0,
    float *dxarr, float dt, float G, int N, float cut,
    int start_idx, int end_idx){
    for (int i = start_idx; i < end_idx; i++){
        float tmp[dim];
        for (int j = 0; j < dim; j++) tmp[j] = 0;
        for (int j = 0; j < N; j++){
            if (j!=i){
            float xij[dim];
```

```
            float mi = marr[i];
            float mj = marr[j];
            // get xij
            get_xij(i, j, dim, xarr, xij, N);
            // compute rij
            float rij = norm(xij, dim);
            float fac = 1.0;
            if (rij < cut) {
                rij = cut;
            }
            // compute intermediate variable
            for (int k = 0; k < dim; k++){
                tmp[k] += xij[k]*G/pow(rij, 3)*mj*dt*dt;
            }
            }
        }
        vec_add(dxarr+i*dim, dxarr+i*dim, tmp, 1.0, 1.0, dim);
    }
}

void verlet_at2_part_omp(int dim, float *marr, float *xarr, float *xarr0,
    float *dxarr, float dt, float G, int N, float cut,
    int start_idx, int end_idx){
    #pragma omp parallel
    {
        int omp_start_idx, omp_end_idx;
        partition(end_idx-start_idx, omp_get_num_threads(), omp_get_thread_num(),
            &omp_start_idx, &omp_end_idx);
        for (int i = start_idx+omp_start_idx; i < start_idx+omp_end_idx; i++){
            float tmp[dim];
            for (int j = 0; j < dim; j++) tmp[j] = 0;
            for (int j = 0; j < N; j++){
                if (j!=i){
                float xij[dim];
                float mi = marr[i];
                float mj = marr[j];
                // get xij
                get_xij(i, j, dim, xarr, xij, N);
                // compute rij
                float rij = norm(xij, dim);
                float fac = 1.0;
                if (rij < cut) {
                    rij = cut;
                }
                // compute intermediate variable
                for (int k = 0; k < dim; k++){
                    tmp[k] += xij[k]*G/pow(rij, 3)*mj*dt*dt;
                }
                }
            }
            vec_add(dxarr+i*dim, dxarr+i*dim, tmp, 1.0, 1.0, dim);
        }
    }
}

void verlet_add(float *a, float *b, float *c, int N, int dim,
    int xmin, int xmax, int ymin, int ymax){
    for (int i = 0; i < N; i++){
        float x = b[i*dim+0] + c[i*dim+0];
        float y = b[i*dim+1] + c[i*dim+1];
        if (x < xmin) x += 2 * (xmin - x);
        else if (x > xmax) x += 2 * (xmax - x);
        if (y < ymin) y += 2 * (ymin - y);
        else if (y > ymax) y += 2 * (ymax - y);
        a[i*dim+0] = x;
        a[i*dim+1] = y;
    }
```

```
}

void verlet_add_omp(float *a, float *b, float *c, int N, int dim,
    int xmin, int xmax, int ymin, int ymax){
    #pragma omp parallel for
    for (int i = 0; i < N; i++){
        float x = b[i*dim+0] + c[i*dim+0];
        float y = b[i*dim+1] + c[i*dim+1];
        if (x < xmin) x += 2 * (xmin - x);
        else if (x > xmax) x += 2 * (xmax - x);
        if (y < ymin) y += 2 * (ymin - y);
        else if (y > ymax) y += 2 * (ymax - y);
        a[i*dim+0] = x;
        a[i*dim+1] = y;
    }
}

void verlet_add_part(float *a, float *b, float *c, int N, int dim,
    int xmin, int xmax, int ymin, int ymax,
    int start_idx, int end_idx){
    for (int i = start_idx; i < end_idx; i++){
        float x = b[i*dim+0] + c[i*dim+0];
        float y = b[i*dim+1] + c[i*dim+1];
        if (x < xmin) x += 2 * (xmin - x);
        else if (x > xmax) x += 2 * (xmax - x);
        if (y < ymin) y += 2 * (ymin - y);
        else if (y > ymax) y += 2 * (ymax - y);
        a[i*dim+0] = x;
        a[i*dim+1] = y;
    }
}

void verlet_add_part_omp(float *a, float *b, float *c, int N, int dim,
    int xmin, int xmax, int ymin, int ymax, int start_idx, int end_idx){
    #pragma omp parallel
    {
        int omp_start_idx, omp_end_idx;
        partition(end_idx-start_idx, omp_get_num_threads(), omp_get_thread_num(),
            &omp_start_idx, &omp_end_idx);
        for (int i = start_idx+omp_start_idx; i < start_idx+omp_end_idx; i++){
            float x = b[i*dim+0] + c[i*dim+0];
            float y = b[i*dim+1] + c[i*dim+1];
            if (x < xmin) x += 2 * (xmin - x);
            else if (x > xmax) x += 2 * (xmax - x);
            if (y < ymin) y += 2 * (ymin - y);
            else if (y > ymax) y += 2 * (ymax - y);
            a[i*dim+0] = x;
            a[i*dim+1] = y;
        }
    }
}

void vec_assign_const(float *a, float c, int dim){
    for (int i = 0; i < dim; i++){
        a[i] = c;
    }
}

void random_generate(float *xarr, float *marr, int N, int dim){
    for (int i = 0; i < N; i++){
        for (int j = 0; j < dim; j++){
            float x = (float) rand() / RAND_MAX * 4 - 2;
            xarr[i*dim+j] = x;
        }
        float m = (float) rand() / RAND_MAX + 1;
        marr[i] = m;
    }
```

```c
}


void compute_seq(float **xarr_ptr, float **xarr0_ptr, float *dxarr, float *marr, int N, int dim,
    float G, float dt, float radius){
    float *tmp;
    float *xarr = *xarr_ptr;
    float *xarr0 = *xarr0_ptr;
    vec_assign_const(dxarr, 0, N*dim);
    verlet_at2(dim, marr, xarr, xarr0, dxarr, dt, G, N, radius); // dx: acc
    vec_add(dxarr, dxarr, xarr, 1.0, 1.0, N*dim);         // dx: x(t)
    vec_add(dxarr, dxarr, xarr0, 1.0, -1.0, N*dim);       // dx: x(t-dt)
    *xarr0_ptr = xarr;
    *xarr_ptr = xarr0;  // switch pointers
    xarr = *xarr_ptr;
    xarr0 = *xarr0_ptr;
    verlet_add(xarr, xarr0, dxarr, N, dim, xmin, xmax, ymin, ymax);    // xarr = xarr(0) + dxarr
}

void compute_omp(float **xarr_ptr, float **xarr0_ptr, float *dxarr, float *marr,
    int N, int dim, float G, float dt, float radius){
    float *xarr = *xarr_ptr;
    float *xarr0 = *xarr0_ptr;
    float *tmp;
    vec_assign_const(dxarr, 0, N*dim);
    verlet_at2_omp(dim, marr, xarr, xarr0, dxarr, dt, G, N, radius); // dx: acc
    vec_add_omp(dxarr, dxarr, xarr, 1.0, 1.0, N*dim);         // dx: x(t)
    vec_add_omp(dxarr, dxarr, xarr0, 1.0, -1.0, N*dim);       // dx: x(t-dt)
    *xarr0_ptr = xarr;
    *xarr_ptr = xarr0;  // switch pointers
    xarr0 = *xarr0_ptr;
    xarr = *xarr_ptr;
    verlet_add_omp(xarr, xarr0, dxarr, N, dim, xmin, xmax, ymin, ymax);    // xarr = xarr(0) + dxarr
}

typedef struct pthArgs{
    int dim;
    float *marr;
    float *xarr;
    float *xarr0;
    float *dxarr;
    float dt;
    float G;
    int N;
    float cut;
    int nt;
    int idx;
    pthread_barrier_t *barr_ptr;
} PthArgs;

void *compute_pth_callee(void *vargs){
    // initialization
    PthArgs args = *(PthArgs *) vargs;
    int dim = args.dim;
    float *marr = args.marr;
    float *xarr = args.xarr;
    float *xarr0 = args.xarr0;
    float *dxarr = args.dxarr;
    float dt = args.dt;
    float G = args.G;
    int N = args.N;
    float radius = args.cut;
    int nt = args.nt;
    int idx = args.idx;
```

```
    pthread_barrier_t *barr_ptr = args.barr_ptr;
    int start_idx, end_idx;

    // verlet algorithm
    partition(N, nt, idx, &start_idx, &end_idx);
    verlet_at2_part(dim, marr, xarr, xarr0, dxarr, dt, G, N, radius, start_idx, end_idx)
        ;
    // vector add
    vec_add_part(dxarr, dxarr, xarr, 1.0, 1.0, N*dim, start_idx*dim, end_idx*dim);
    pthread_barrier_wait(barr_ptr);
    vec_add_part(dxarr, dxarr, xarr0, 1.0, -1.0, N*dim, start_idx*dim, end_idx*dim);
    pthread_barrier_wait(barr_ptr);
    float *tmp = xarr;
    xarr = xarr0;
    xarr0 = tmp;
    pthread_barrier_wait(barr_ptr);
    verlet_add_part(xarr, xarr0, dxarr, N, dim, xmin, xmax, ymin, ymax, start_idx,
        end_idx);

    return NULL;
}

void compute_pth(float **xarr_ptr, float **xarr0_ptr, float *dxarr, float *marr,
    int N, int dim, float G, float dt, float radius, int nt){
    float *tmp;
    float *xarr = *xarr_ptr;
    float *xarr0 = *xarr0_ptr;
    pthread_t threads[nt];
    pthread_barrier_t barr;
    PthArgs args_arr[nt];
    pthread_barrier_init(&barr, NULL, nt);
    // call verlet
    vec_assign_const(dxarr, 0, N*dim);
    for (int i = 0; i < nt; i++){
        args_arr[i] = (PthArgs){.dim=dim, .marr=marr, .xarr=xarr, .xarr0=xarr0,
            .dxarr=dxarr, .dt=dt, .G=G, .N=N, .cut=radius,
            .nt=nt, .idx=i, .barr_ptr=&barr};
        pthread_create(&threads[i], NULL, compute_pth_callee, (void *)(&args_arr[i]));
    }
    // join threads
    for (int i = 0; i < nt; i++)
        pthread_join(threads[i], NULL);
    // switch pointers
    *xarr_ptr = xarr0;
    *xarr0_ptr = xarr;
}

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 *marr, float *xarr, int N, int dim, int Tx, int Ty,
    float xmin, float xmax, float ymin, float ymax);
void compute_cu(float *xarr, int nsteps, int N, int dim, float G, float dt, float cut);
void finalize_cu();
```

src/const.h

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

// global variables
// computing-related constants
int        N = 200;   // number of particles
int   nsteps = 1e5;   // number of steps
int        dim = 2;   // dimension
float radius = 0.01;   // gravity cut-off
float        G = 0.1;   // gravity constant
float   dt = 0.001;   // time step
float        *marr;   // mass array
float        *xarr;   // position array at time t
float       *xarr0;   // position array at time t - dt
float        *varr;   // velocity array
float       *dxarr;   // position shift array
float       *dvarr;   // velocity shift array
float xmin = -10;
float xmax =  10;
float ymin = -10;
float ymax =  10;

// IO & runtime options
int record = 0;
int nt = 1;

// mpi parameters
int size, rank;
float *xarr_copy;

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