CSC4005 Project 3 Template

Physics

We have some physics variable declared in `headers/physics.h`:

```
#define bound_x 4000
#define bound_y 4000
#define max_mass 400
#define err 1e-5f
#define dt 0.0001f
#define gravity_const 1000000.0f
#define radius2 0.01f
```

`gravity_const` is the gravity constant when you compute $F=G m*{i} m*{j} / d^2$.

`dt` is the time span between two iterations, it can be used when you compute $\Delta v=F\Delta t$ and $\Delta x=v\Delta t$.

`err` is a small number used to avoid `DivisionByZero` error and can accelerate computation. It can be used like $F=G m*{i} m*{j} / (d^2 + error)$.

`radius2` is the squared radius of particles. It can be used when you determine whether two particles have collision.

`bound_x` is the upper bound of X axis. x is between $[0,bound_x]$.

`bound_y` is the upper bound of Y axis. y is between $[0,bound_y]$.

`max_mass` is the maximum mass of a particle. You can use it to generate particles.

You may need to modify them to better visualize your result.

Workflow & Logger & Reproduce

Workflow

Simulation with a large amount of bodies is impossible on your own VM, so it is necessay to simulate on cluster and then transfer the data back to your VM for visualization. We have implemented 2 utilities to help you do that.

Also, for CUDA implementation, there is no CUDA on our VM, so you can use cluster to generate simulation results first, then you can transfer the results back to your VM. You can visualize the results on your VM.

Logger

We provide a utility called `Logger` at `headers/logger.h`, it can save `x,y` coordinates of multiple frames. Result will be stored in `./checkpoints/xxxxxxx/`.

Here we give a sample use:

```
Logger l = Logger("cuda", 10000, 4000, 4000);
for (int i = 0; i < n_iterations; i++){
    // compute x, y
    l.save_frame(x, y);
}
```

Copy Results Back

You can use `scp` command.

Execute the following command on your VM.

```
scp -r $STUDENT_ID@10.26.200.21:/nfsmnt/$STUDENT_ID/xxxxx .
```

here `-r` represent recursively copying all files within a given directory.

You will find the target directory on cluster appears in your local working directory.

Produce a Video

We have implemented `video.cpp` to help you visualize simulation results generated by cluster.

Use

```
g++ ./src/video.cpp -o video -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL
-lm -DGUI -O2 -std=c++11
```

to compile `video` GUI application on your VM.

Once you get a checkpoint directory like `./checkpoints/sequential_1000_20221107025155` (from cluster or somewhere else), you can use

```
./video ./checkpoints/sequential_1000_20221107025155
```

to reproduce result with GUI on your VM.

So, you can visualize the output of your CUDA program!

Compile & Run

Compile

Sequential (command line application):

```
g++ ./src/sequential.cpp -o seq -O2 -std=c++11
```

Sequential (GUI application):

```
g++ ./src/sequential.cpp -o seqg -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm -DGUI -O2 -std=c++11
```

MPI (command line application):

```
mpic++ ./src/mpi.cpp -o mpi -std=c++11
```

MPI (GUI application):

```
mpic++ ./src/mpi.cpp -o mpig -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm -DGUI -std=c++11
```

Pthread (command line application):

```
g++ ./src/pthread.cpp -o pthread -lpthread -O2 -std=c++11
```

Pthread (GUI application):

```
g++ ./src/pthread.cpp -o pthreadg -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm -lpthread -DGUI -O2 -std=c++11
```

CUDA (command line application): notice that `nvcc` is not available on VM, please use cluster.

```
nvcc ./src/cuda.cu -o cuda -O2 --std=c++11
```

CUDA (GUI application): notice that `nvcc` is not available on VM, please use cluster.

```
nvcc ./src/cuda.cu -o cudag -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm -O2 -DGUI --std=c++11
```

OpenMP (command line application):

```
g++ ./src/openmp.cpp -o openmp -fopenmp -O2 -std=c++11
```

OpenMP (GUI application):

```
g++ ./src/openmp.cpp -o openmpg -fopenmp -I/usr/include -L/usr/local/lib -L/usr/lib -lglut
-lGLU -lGL -lm -O2 -DGUI -std=c++11
```

Run

Sequential (command line mode):

```
./seq $n_body $n_iterations
```

Sequential (GUI mode): please run this on VM (with GUI desktop).

```
./seqg $n_body $n_iterations
```

MPI (command line mode):

```
mpirun -np $n_processes ./mpi $n_body $n_iterations
```

MPI (GUI mode): please run this on VM (with GUI desktop).

```
mpirun -np $n_processes ./mpig $n_body $n_iterations
```

Pthread (command line mode):

```
./pthread $n_body $n_iterations $n_threads
```

Pthread (GUI mode): please run this on VM (with GUI desktop).

```
./pthreadg $n_body $n_iterations $n_threads
```

CUDA (command line mode): for VM users, please run this on cluster.

```
./cuda $n_body $n_iterations
```

CUDA (GUI mode): if you have both nvcc and GUI desktop, you can try this.

```
./cuda $n_body $n_iterations
```

OpenMP (command line mode):

```
openmp $n_body $n_iterations $n_omp_threads
```

OpenMP (GUI mode):

```
openmpg $n_body $n_iterations $n_omp_threads
```

Makefile

Makefile helps you simplify compilation command.

```
make $command
```

where `command` is one of `seq, seqg, mpi, mpig, pthread, pthreadg, cuda, cudag, openmp, openmpg`.

When you need to recompile, please first run `make clean`!

Advertisement: Valgrind

Valgrind is a memory mismanagement detector. It shows you memory leaks, deallocation errors, etc. Actually, Valgrind is a wrapper around a collection of tools that do many other things (e.g., cache profiling); however, here we focus on the default tool, memcheck. Memcheck can detect:

- Use of uninitialised memory
- Reading/writing memory after it has been free'd
- Reading/writing off the end of malloc'd blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks -- where pointers to malloc'd blocks are lost forever
- Mismatched use of malloc/new/new [] vs free/delete/delete []
- Overlapping src and dst pointers in memcpy() and related functions
- Some misuses of the POSIX pthreads API

To use this on our example program, test.c, try

```
gcc -o test -g test.c
```

This creates an executable named test. To check for memory leaks during the execution of test, try

```
valgrind --tool=memcheck --leak-check=yes --show-reachable=yes --num-callers=20 --track-fds=yes ./test
```

This outputs a report to the terminal like

```
=9704== Memcheck, a memory error detector for x86-linux.
==9704== Copyright (C) 2002-2004, and GNU GPL'd, by Julian Seward et al.
==9704== Using valgrind-2.2.0, a program supervision framework for x86-linux.
==9704== Copyright (C) 2000-2004, and GNU GPL'd, by Julian Seward et al.
==9704== For more details, rerun with: -v
==9704==
==9704==
==9704== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 11 from 1)
==9704== malloc/free: in use at exit: 35 bytes in 2 blocks.
==9704== malloc/free: 3 allocs, 1 frees, 47 bytes allocated.
==9704== For counts of detected errors, rerun with: -v
==9704== searching for pointers to 2 not-freed blocks.
==9704== checked 1420940 bytes.
==9704==
==9704== 16 bytes in 1 blocks are definitely lost in loss record 1 of 2
==9704==    at 0x1B903D38: malloc (vg_replace_malloc.c:131)
==9704==    by 0x80483BF: main (test.c:15)
==9704==
==9704==
==9704== 19 bytes in 1 blocks are definitely lost in loss record 2 of 2
==9704==    at 0x1B903D38: malloc (vg_replace_malloc.c:131)
==9704==    by 0x8048391: main (test.c:8)
==9704==
==9704== LEAK SUMMARY:
==9704==    definitely lost: 35 bytes in 2 blocks.
==9704==    possibly lost:   0 bytes in 0 blocks.
==9704==    still reachable: 0 bytes in 0 blocks.
==9704==         suppressed: 0 bytes in 0 blocks.
```

Reference:

[1] http://cs.ecs.baylor.edu/~donahoo/tools/valgrind/

[2] http://senlinzhan.github.io/2017/12/31/valgrind/

Sbatch script

For example, we want to use 20 cores for experiment.

MPI

For MPI program, you can use

```
#!/bin/bash
#SBATCH --job-name=your_job_name # Job name
#SBATCH --nodes=1                    # Run all processes on a single node
#SBATCH --ntasks=20                   # number of processes = 20
#SBATCH --cpus-per-task=1      # Number of CPU cores allocated to each process (please use
1 here, in comparison with pthread)
#SBATCH --partition=Project            # Partition name: Project or Debug (Debug is
default)

cd /nfsmnt/119010355/CSC4005_2022Fall_Demo/project3_template/
mpirun -np 4 ./mpi 1000 100
mpirun -np 20 ./mpi 1000 100
mpirun -np 40 ./mpi 1000 100
```

Pthread

For pthread program, you can use

```
#!/bin/bash
#SBATCH --job-name=your_job_name # Job name
#SBATCH --nodes=1                    # Run all processes on a single node
#SBATCH --ntasks=1                   # number of processes = 1
#SBATCH --cpus-per-task=20      # Number of CPU cores allocated to each process
#SBATCH --partition=Project            # Partition name: Project or Debug (Debug is
default)

cd /nfsmnt/119010355/CSC4005_2022Fall_Demo/project3_template/
./pthread 1000 100 4
./pthread 1000 100 20
./pthread 1000 100 40
./pthread 1000 100 80
./pthread 1000 100 120
./pthread 1000 100 200
...
```

here you can create as many threads as you want while the number of cpu cores are fixed.

For a pthread program, we notice that sbatch script contains

```
#SBATCH --ntasks=1                   # number of processes = 1
#SBATCH --cpus-per-task=20      # Number of CPU cores allocated to each process
```

the meaning of these two lines are: only one process is started, it can create many threads, where threads are distributed to all available 20 cpu cores by OS.

CUDA

For CUDA program, you can use

```
#!/bin/bash

#SBATCH --job-name CSC3150CUDADemo  ## Job name
#SBATCH --gres=gpu:1                ## Number of GPUs required for job execution.
#SBATCH --output result.out         ## filename of the output
#SBATCH --partition=Project           ## the partitions to run in (Debug or Project)
#SBATCH --ntasks=1                  ## number of tasks (analyses) to run
#SBATCH --gpus-per-task=1           ## number of gpus per task
#SBATCH --time=0-00:02:00           ## time for analysis (day-hour:min:sec)

## Compile the cuda script using the nvcc compiler
## You can compile your codes out of the script and simply srun the executable file.
cd /nfsmnt/119010355/CSC4005_2022Fall_Demo/project3_template/
## Run the script
srun ./cuda 10000 100
```

OpenMP

For OpenMP program, you can use

```
#!/bin/bash

#SBATCH --job-name job_name  ## Job name
#SBATCH --output result.out         ## filename of the output
#SBATCH --partition=Project           ## the partitions to run in (Debug or Project)
#SBATCH --ntasks=1                  ## number of tasks (analyses) to run
#SBATCH --gpus-per-task=1           ## number of gpus per task
#SBATCH --time=0-00:02:00           ## time for analysis (day-hour:min:sec)

## Compile the cuda script using the nvcc compiler
## You can compile your codes out of the script and simply srun the executable file.
cd /nfsmnt/119010355/CSC4005_2022Fall_Demo/project3_template/
## Run the script
./openmp 10000 100 20
```

To submit your job, use

```
sbatch xxx.sh
```

Salloc

If you want to run your program using interactive mode, use

MPI

For MPI porgram, we have learned before:

```
salloc -n20 -c1 # -c1 can be omitted.
mpirun -np 20 ./mpi 1000 1000 100
```

Pthread

For pthread program,

```
salloc -n1 -c20 -p Project # we have only 1 process, 20 is the number of cores allocated per process.
srun ./pthread 1000 1000 100 20 # 20 is the number of threads.
```

CUDA

For CUDA program,

```
salloc -n1 -c1 --gres=gpu:1 -p Project # require 1 gpu
srun ./cuda 10000 1000
```

OpenMP

For openMP program,

```
salloc -n1 -c20 -p Project # require 1 gpu
srun ./openmp 10000 1000 20
```

Changelog Nov 14, 2022

1. Fixed wrong CUDA api invoke

The original wrong invoke is

```
cudaMemcpy(device_m, m, n_body, cudaMemcpyHostToDevice);
```

The fixed invoke is

```
cudaMemcpy(device_m, m, n_body * sizeof(double), cudaMemcpyHostToDevice);
```

If you have included wrong invoke in your job, you will come across a lot of problems. Sorry for this.

2. Updated physics constant definition

Previously, the physics constant is defined using variable definition. This approach make some trouble when you try to use them in CUDA program. So, we have changed the definition of physics constants back to macro definition, like this:

```
#define bound_x 4000
#define bound_y 4000
#define max_mass 400
#define err 1e-5f
#define dt 0.0001f
#define gravity_const 1000000.0f
#define radius2 0.01f
```

You can directly use them in your cuda program (they are treated as constants on both your host and device).

3. Changed `block_size` for CUDA program to `512`

Some students reported that their kernel function will not run with `block_size` = `1024`. So, we have tested that a smaller `block_size` is available.

4. Changed some physics constants to obtain better visualization

```
#define bound_x 4000
#define bound_y 4000
#define max_mass 400
#define err 1e-5f
#define dt 0.0001f
#define gravity_const 1000000.0f
#define radius2 0.01f
```

We have changed `max_mass` to `400` instead of `40000000`. Because in the latter case extreme velocity will occur very frequently.

We have also changed `err` to `1e-5f` instead of `1e-9f` to avoid large force.

We have changed `gravity_const` larger (original value is `1.0f`) to make sure the motion of bodies visible.

We have also changed `radius2` smaller (original value is `4.0f`) to avoid frequent collision.

5. Changed the data generator

We have updated `generate_data` function.

```
void generate_data(double *m, double *x,double *y,double *vx,double *vy, int n) {
    // TODO: Generate proper initial position and mass for better visualization
    srand((unsigned)time(NULL));
    for (int i = 0; i < n; i++) {
        m[i] = rand() % max_mass + 1.0f;
        x[i] = 2000.0f + rand() % (bound_x / 4);
        y[i] = 2000.0f + rand() % (bound_y / 4);
        vx[i] = 0.0f;
        vy[i] = 0.0f;
    }
}
```

This function can make the initial positions bodies more concentrated, so, you can easily reproduce the results given by Prof.Chung.

6. Wrong order of time and logger

Previously, `l.save_frame(x, y);` was executed earlier than `std::chrono::high_resolution_clock::time_point t2 =std::chrono::high_resolution_clock::now();`, so the time was not accurate.

Now we have changed the order into following order:

```
std::chrono::high_resolution_clock::time_point t1 =
std::chrono::high_resolution_clock::now();

/* computation part */

std::chrono::high_resolution_clock::time_point t2 =
std::chrono::high_resolution_clock::now();// new order
std::chrono::duration<double> time_span = t2 - t1;

printf("Iteration %d, elapsed time: %.3f\n", i, time_span);

l.save_frame(x, y);  // new order
```

If yours are not in such order, you may get wrong running time! Please check all versions.

Fixed unpaired use of new/delete

If you are still using

```
delete m;
delete x;
delete y;
delete vx;
delete vy;
```

It is recommended to switch to

```
delete[] m;
delete[] x;
delete[] y;
delete[] vx;
delete[] vy;
```

Authors

Bokai Xu

Thank @Peilin Li, @Yangyang Peng, and @SydianAndrewChen for giving valuable suggestions to this series of templates.