Gell User Manual

Jiayi Du

dujy@g.ucla.edu

August 1st, 2022

Contents

About Gell

Gell is an open-source, fast cell-based simulation software for ultra-large-scale multicellular simulations. It runs entirely on GPU and offers ~400X speedup compared to single-thread CPU-based simulators. With Gell, you can easily handle large-scale simulations with millions of voxels and cells on personal computers with an Nvidia GPU. Gell is developed with C++ and CUDA for computational speed and memory efficiency considerations, but you do not have to be an expert in C++ or CUDA before starting. As a super lightweight and extendable codebase, it will not cost you too much effort to learn to use or extend Gell for your own problems. This document will first show you how to set up the development environment to run and visualize your first HDS simulation. Then we will comprehensively explain all critical simulation modules and provide some small tips for Gell extension.

The source code is available at https://github.com/PhantomOtter/Gell .

The preprint of this work is available at https://www.biorxiv.org/content/10.1101/2022.09.01.506296 .

Feel free to contact me (dujy@g.ucla.edu) if you had any question.

Development environments

This section will show you how to prepare the development environment and run Gell with your own device. Please be aware that Gell is programmed using C++ and CUDA, and CUDA programming language only works for Nvidia GPUs. So make sure you have an Nvidia GPU before starting.

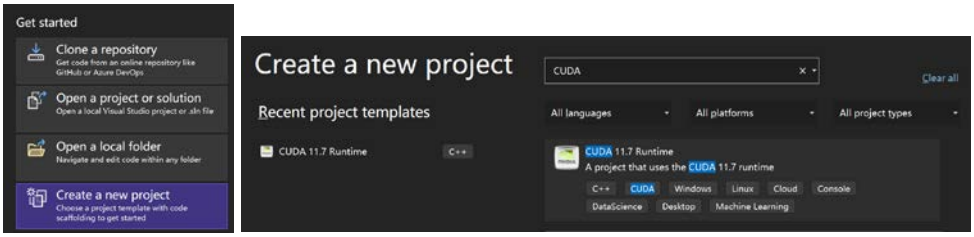
For Windows users

Visual Studio is recommended for development on windows systems.

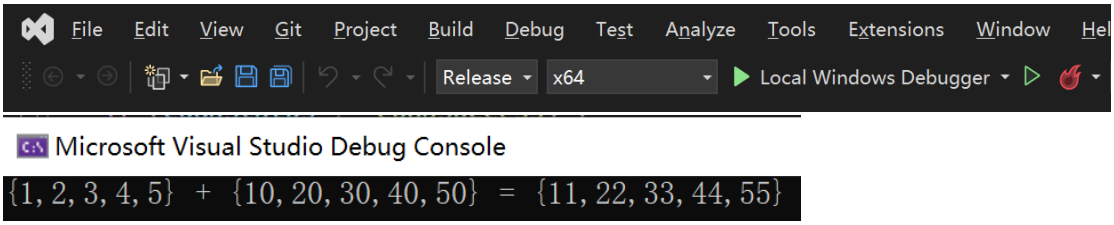
1. Install **Visual Studio** Community.
   Simply download from the following website and install the software.
   https://visualstudio.microsoft.com/vs/community/

2. Install **CUDA Toolkit**.
   Simply download from the following website and install the software.
   https://developer.nvidia.com/cuda-downloads

3. Then you should be able to create a new project using the **CUDA runtime template** in visual Studio.



4. Try to compile and execute the template project with release mode



If successful, Visual Studio is now ready to work with CUDA.

5. **Delete** the template kernel file from the project, then Right click the project and choose "**add existing item**" to include the Gell source file in the project.

6. Compile and run the project with release setting.

For Linux users

1. Install g++ and CUDA Toolkit
   Reference: https://linuxhint.com/install-cuda-ubuntu/
   a. Update the package repository cache
      `sudo apt update`
   b. Install gcc and other building tools
      `sudo apt install build-essential`
      check installation
      `gcc --version`
   c. Install CUDA
      `sudo apt install nvidia-cuda-toolkit`
      check installation
      `nvcc –version`

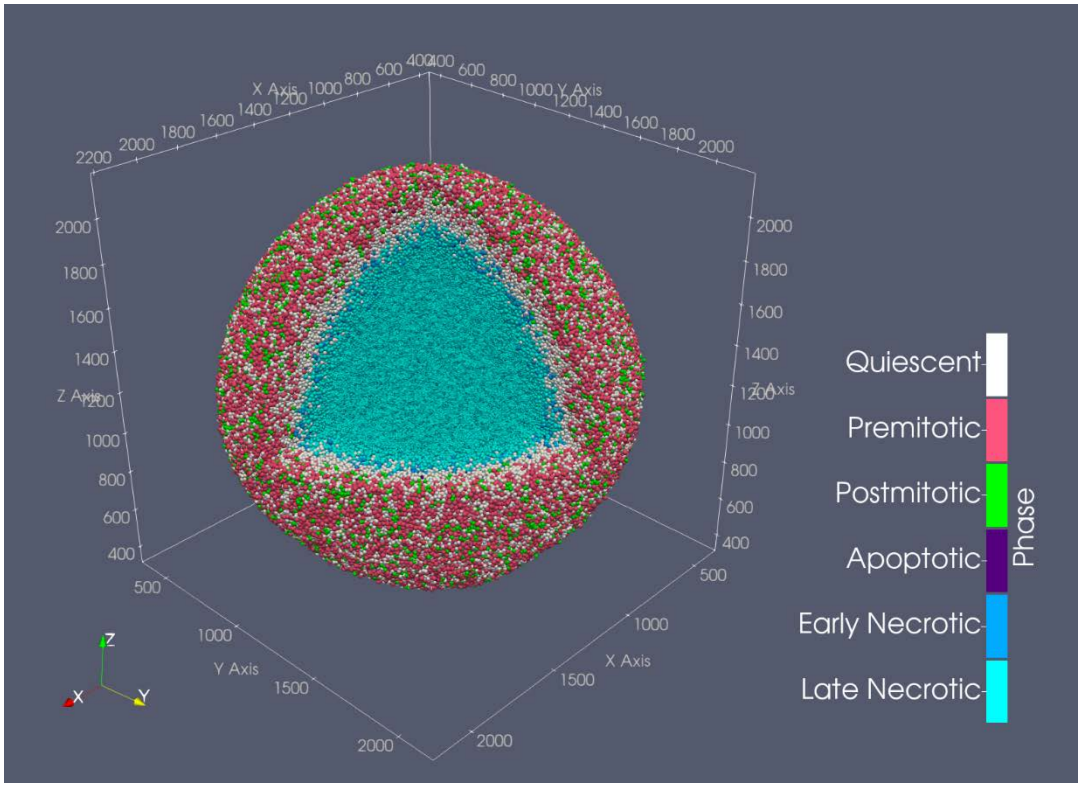
2. Download Gell source code and compile Gell using the following command
   `make`
   After the compilation, execute the executable with
   `./HDS`

Fast start —— HDS example, output format, and visualization

HDS example

The provided source codes are for HDS simulation. In the HDS benchmark, a suspended multicellular aggregate is cultured in the middle of a growth medium with oxygen supplied through diffusion from the domain boundary. The simulation started with 2347 cells and evolved to nearly one million cells after 450 hours of cultivation. The simulation domain contains one million isotropic voxels with a side length of 25 um. Cell mechanics and phase update is calculated every 0.1 minutes, and the diffusion-reaction of oxygen in the extracellular fluid is solved every 0.01 minutes.

In our test platform with an Nvidia RTX 2080Ti graphics card, Gell completed the 19-day simulation within 50 minutes. If you don't want to go over the entire cultivation, you can adjust the `Max_Simulation_Time` in file `./include/Definitions.h` .



Output format

The output file is in CVS format.
You can change the output folder in **Main.cu** line 40

```
std::string datapath = "./data/";
```

And you can decide whether you want to save the intermediate data or not in **Main.cu** line 30-31

```
#define save_intermediate_Cell true
#define save_intermediate_Mesh false
```

If you choose not to save intermediate data, Gell will only record the cell and mesh data at the start and the end of the entire simulation.

The **Gell_#.csv** file contains cell-related information.
Columns in the output file represent the following values

1	2	3	4	5	6	7	8	9	10
Index	Sign	Phase	X (um)	Y (um)	Z (um)	X (um)	V (um^3)	Vns (um^3)	Vcs (um^3)

The cell index counts from 0.
The **Mesh_#.csv** file records the oxygen concentration field data.

1	2	3	4
x	y	z	Concentration (mmHg)

x,y,z are indexes of the voxel, counts from 0.

To save time and storage space, the default mesh saves only record one slice of the multi-layer 3D domain: **Mesh_slice_#.csv**.

1	2	3
x	y	Concentration (mmHg)

x, y are indexes of the voxel, counts from 0.
The cell number and simulation time cost of every simulation step (cell phase update step: default 6 min) is recorded in **Gell_Time_record.csv**

0	1	2
Index	Cell Num	Time (s)

The record index counts from 0.
The cell number and simulation time cost of every recording gap (default 6 h) is recorded in **Gell_GapTime_record.csv**

0	1	2
Index	Cell Num	Time (s)

The record index counts from 0.

Code related to data saving can be found in `./include/FileSave.cuh`.
The function to save cell data:

```
void savecsv_cell(thrust::host_vector<Cell>& C, int currentnum, const std::string f)
```

The function to save whole 3D mesh:

```
void savecsv_mesh(Mesh_struct* mesh, const std::string f)
```

The function to save a mesh slice:

```
void savecsv_meshslice(Mesh_struct* mesh, const std::string f)
```

The functions to record cell number and simulation time cost change over time:

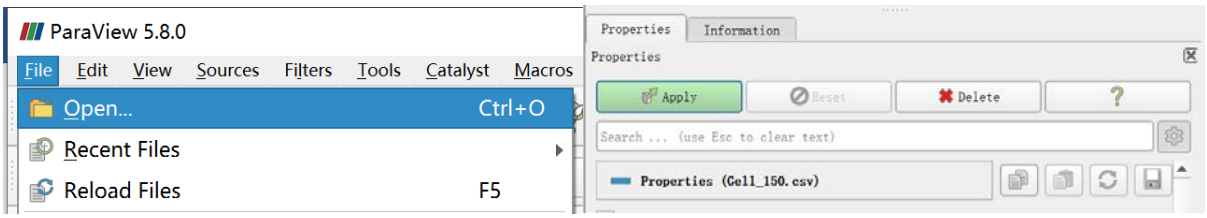
```
void savecsv_array(int* cnum, double* time, const std::string f)
void savecsv_gaparray(int* cnum, double* time, const std::string f)
```

Visualization

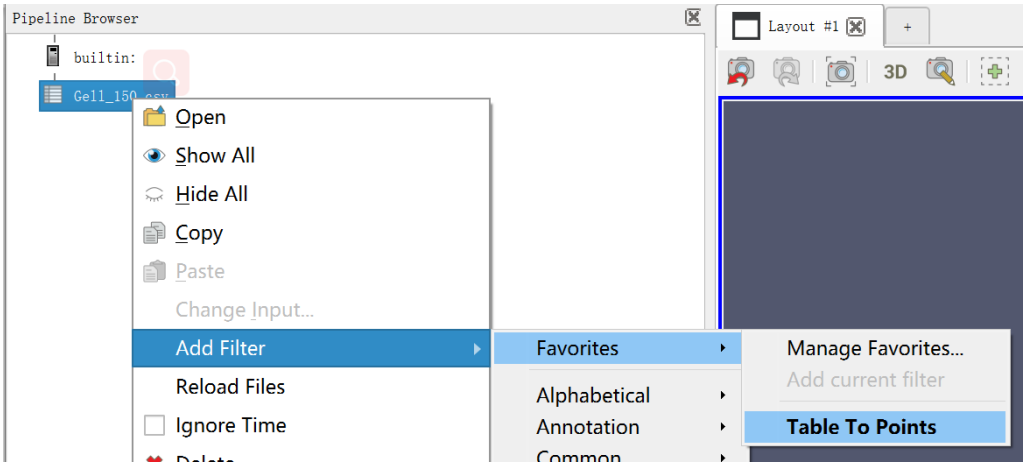
We recommend using **ParaView** to visualize the simulation result. ParaView is an open-source, multi-platform data analysis and visualization application.
You can freely download the software at https://www.paraview.org/download/.
You can either configure you visualization from scratch or use our provided visualization script.

Start from scratch

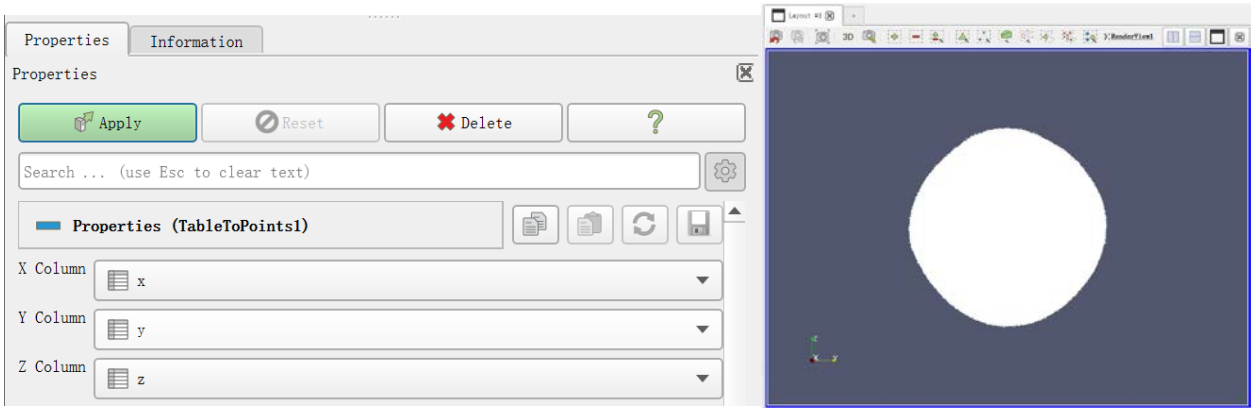
You can directly import csv files into ParaView.
Select **File – Open –** and choose **Gell_#.csv.** Click apply in the **Properties window** to load the selected file.



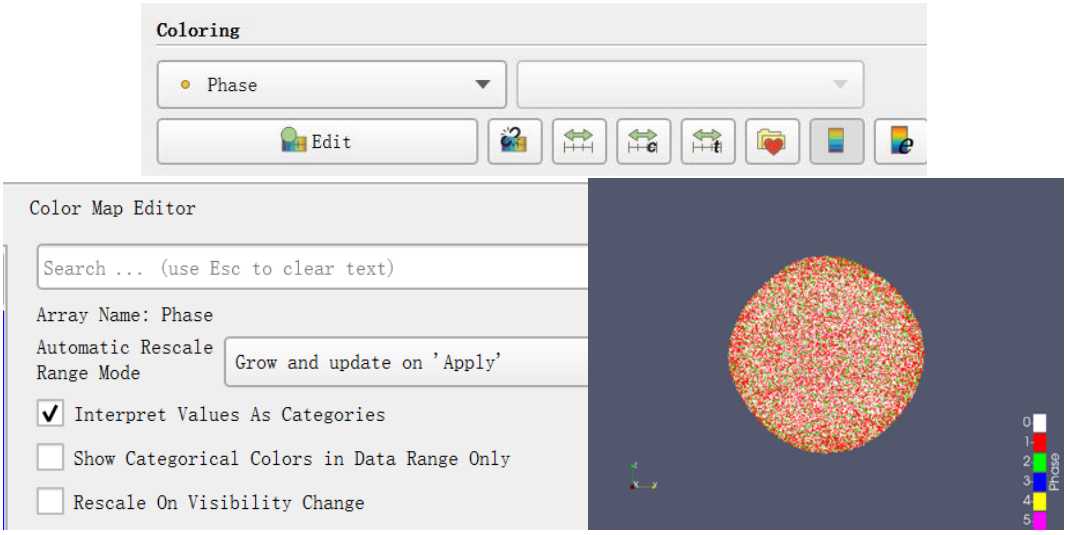
Right click the file and select add filter.
Find and apply the **Table To Points** filter in the Manu.



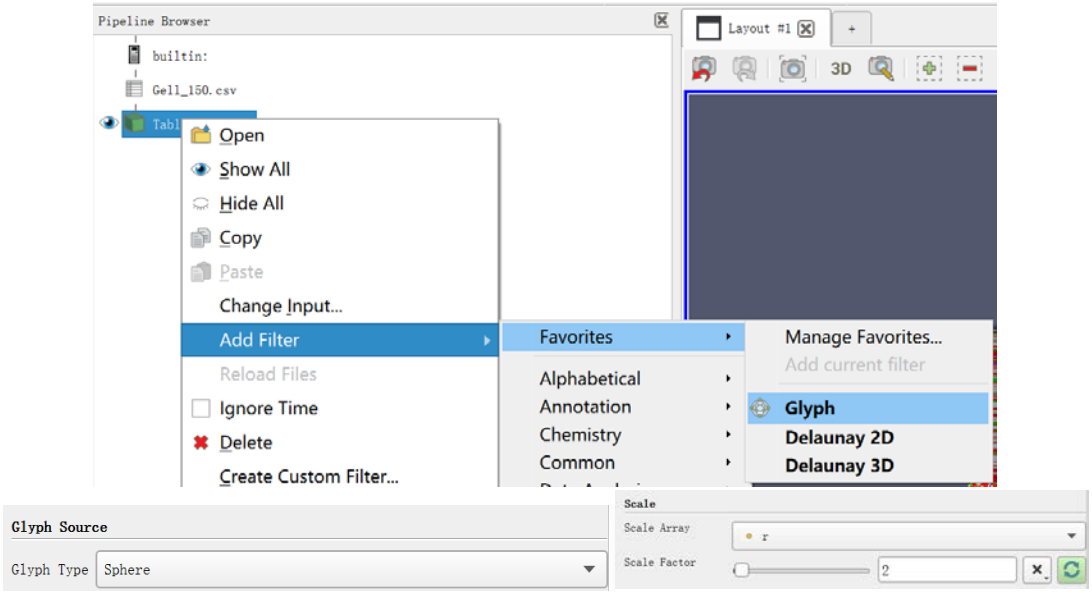
Configure the **x y z** position corresponding column and **Apply**. Then you should be able to see a point cloud in the rendered window**.**



Select **Phase** data to determine the point color in the **Properties window** and click the **interpret values as categories** option in the **Color Map Editor window**. Then you can get a colored point cloud. The color map and legend can be further customized in the **Color Map Editor**.

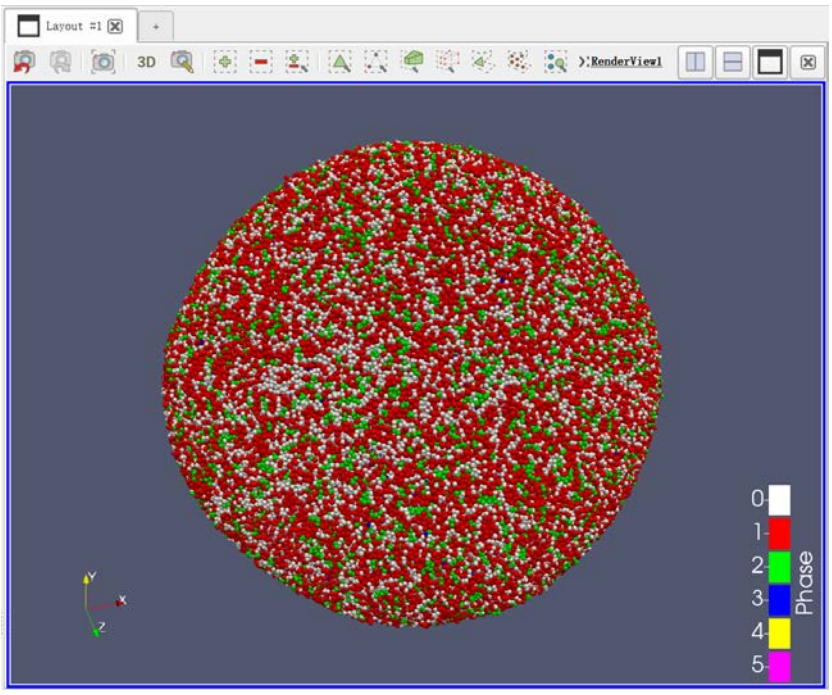


To render the points as spheres of certain size, you can apply the **Glyph** filter to the point data and select **Sphere** as the Glyph type and use cell radius **r** to scale the sphere size.



Also make sure to render all the points to get an accurate visualization.

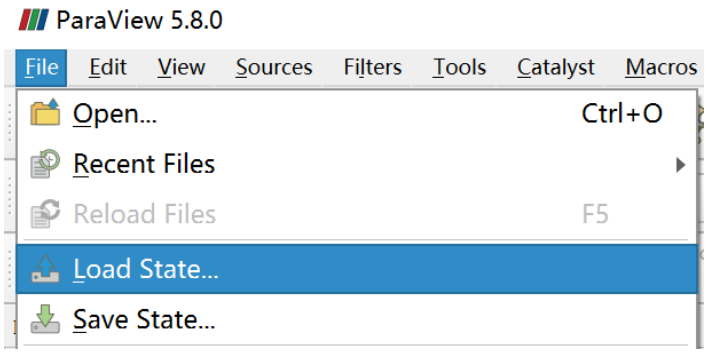
The rendered image will be like this.



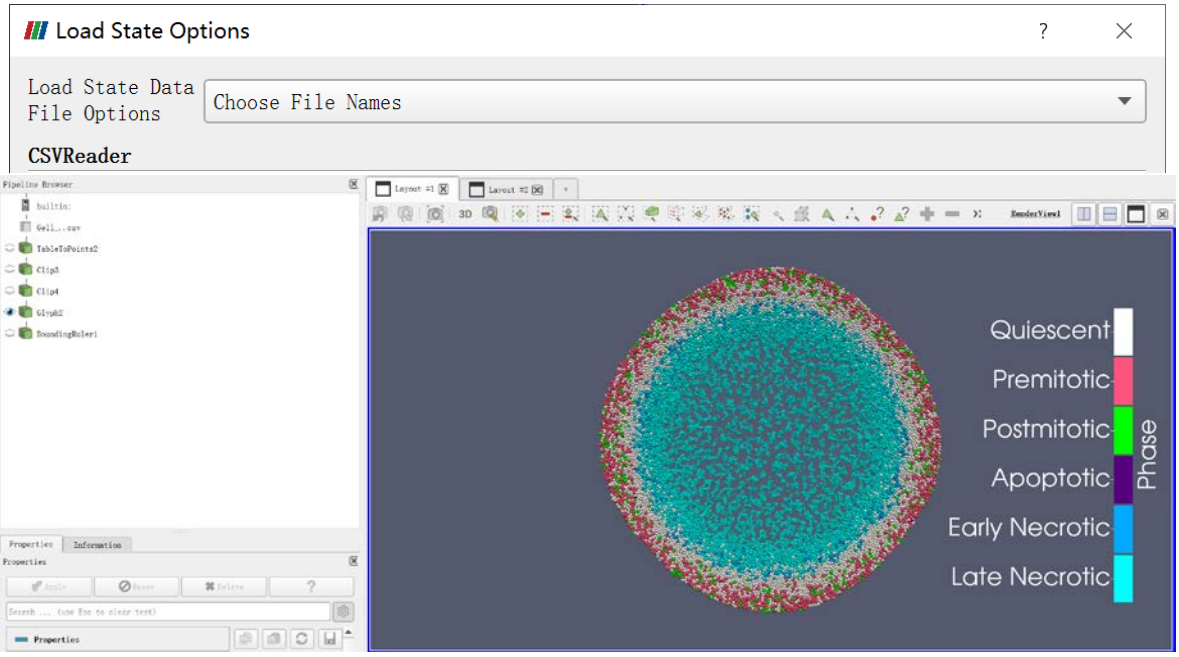
Then you can further improve your visualization by hiding part of the cells using the **Clip** filter to make the internal microstructures more visible. The final spheroid image can be like the image showed in the **HDS example section**.

Use provided ParaView state

Click **Load State** in the **File** menu and select the **HDS_Slice.pvsm** file provided with the code. Choose the data csv in the **Load State Option** window and **apply**.



Then you are all set.



Code structure

Gell is a lightweight but fast and memory-saving GPU-based simulation code base. The whole codebase contains less than 2k lines while providing computation templates for all the computationally demanding parts of a typical agent-based hybrid simulation, e.g., cell-cell interaction and PDE solving.

This section will comprehensively illustrate what the code does in different simulation modules. One with even no CUDA experience could also understand without much difficulty.

File overview:

`Main.cu`	Main file for initialization and loop control
`/include/Definitions.h`	Definitions of basic simulation parameters
`/include/BasicFunctions.cuh`	Useful functions for other cuh
`/include/RandomFunctions.cuh`	
`/include/Initialization.cuh`	Cell spheroid initialization
`/include/FileSave.cuh`	Write simulation results to csv files
`/include/Cells.cuh`	Definitions of cell property and behavior
`/include/PDkernel.cuh`	Kernel for cell proliferation and death
`/include/LODkernel.cuh`	Kernel for PDE solver (LOD method)
`/include/FMkernel.cuh`	Kernel for cell force and movement calculation

Main loop

Initialization

When the simulation start, Gell will first allocate CPU and GPU memory for cell vectors with length equal to `Max_Cell_num` and meshes for PDEs. Data is first initialized on the CPU and then transferred to GPU. All following computations are entirely on GPU. The total cell number is synchronized in variable `gcurrentnum` (GPU) and `currentnum` (CPU).

Initialization of meshes for diffusive substance. **Main.cu** line 65

```
Mesh_struct O2Mesh(O2_Default_Concentration, O2_Diffusion_coef, O2_Decay_rate);
```

Memory allocation for cells on CPU and GPU. **Main.cu** line 86 and 81

```
thrust::host_vector<Cell> CpuCell(Max_Cell_num);
```

```
thrust::device_vector<Cell> GpuCell(Max_Cell_num);
```

CPU Cell initialization. **Main.cu** line 89

```
Cell_sphere_Initialization(CpuCell, initnum);
```

Copy the CPU data to GPU. **Main.cu** line 97

```
GpuCell = CpuCell;
```

Also, to setup random state for random number generating on GPU. **Main.cu** line 71

```
curandState* curand_states;
cudaMalloc(&curand_states, sizeof(curandState) * Max_Cell_num);
set_random_states << <(Max_Cell_num + BlockWidth1d - 1) / BlockWidth1d,
BlockWidth1d >> > (curand_states);
cudaDeviceSynchronize();
```

The Memory footprint Gell per additional million cells and per additional million domain voxels in the HDS simulation task are listed in the table below.

Memory Footprint of HDS Simulation	500 MB
Memory Footprint per Additional Million Cells (MB)	118.0 MB
Memory Footprint per Additional Million Voxels (MB)	15.38 MB

Loop

The main simulation loop is:

```
for (float current_cell_time = 0.f; current_cell_time <
Max_Simulation_Time;current_cell_time += Biology_dt) {
	CellBirth_kernel
	for (float mect = 0.0; mect < Biology_dt; mect += Mechanics_dt) {
		Force&Movement_kernel
		PreLOD
		for (float dift = 0.0; dift < Mechanics_dt; dift += Diffusion_dt) {
			LODsolver
		}
	}
```

```
CellDeath_kernel
Data save / information display related functions}
```

The `CellBirth_kernel` updates cell growth and phase transition. `CellDeath_kernel` cleans up the dead cells (dead cell reaches 95% shrinkage) from the cell array.
The `Force&Movement_kernel` sort cells, then calculates cell-cell interactions, and finally moves the cells according to the aggregated force.
The `PreLOD` kernel collects the oxygen consumption map for the entire domain for the following diffusion-reaction updates. `LODsolver` updates the diffusive molecular concentration in the extracellular fluid.
The time cost of a sign call of each simulation module is shown in the table below. The test problem contains one million living cells and the domain contains one million 25um isotropic voxels.

Module	Time cost per invocation
Morton code calculation	0.303 ms
Cell sorting	6.041 ms
Force calculation	3.613 ms
Cell movement	0.360 ms
Reaction-diffusion update	0.542 ms
Phase update with birth and death	4.744 ms
Memory copy between CPU and GPU	0.011 ns
Others	0.387 ms

Simulation settings

Simulation settings can be found in `./include/Definitions.h`
Including
Simulation step size. `Biology_dt` is for cell proliferation/death, phase transition, and volume update time step size. `Mechanics_dt` is for cell-cell interaction and movement update time step size. `Diffusion_dt` is for diffusion-reaction PDE update step size.

```
#define Biology_dt 6.f // min
#define Mechanics_dt 0.1f   //min
#define Diffusion_dt 0.01f  //min
```

CUDA block size related settings. These values can be GPU specific and tuned for better performance.

```
#define BlockWidth3d 4
#define BlockWidth2d 4
#define BlockWidth1d 64
```

Max cell number. Gell will pre-allocated memory for all the cells and adjust the random number generator according to the max cell num. Make sure to choose a suitable max cell number to save memory.

```
#define Max_Cell_num (2*1000*1000)
```

Initialization settings. The cell number after initialization.

```
#define Cell_Init_num 2347
```

Domain size and voxel size. The voxel number is equal for all x, y, and z coordinates. The Voxel_length is in um. Total domain length is `Voxel_num*Voxel_length.`

```
#define Voxel_num 100
#define Voxel_length 25
```

Simulation time. Total simulation time is determined here, the unit is in minute.

```
#define Max_Simulation_Time (20.f*24*60)
```

Data saving gap. The during between two data saving process. Number is in minute.

```
#define Save_data_gap 360 // min
```

Oxygen diffusion and consumption related parameters. The death cell will continue to consume oxygen at a decreased consumption rate until it get eliminated from the system when it reaches 95% volume shrink.

```
#define O2_Default_Concentration 38.f  // mmHg
#define O2_Diffusion_coef 100000.f // um2 per min
#define O2_Decay_rate 0.1f //  per min
#define Tumor_O2_consumption -10.f  //per min
#define Dead_O2_consumption_factor 0.1f
```

Cell **size and mechanics** related parameters and **phase transition** related parameters. Meaning of these values can be found in the paper.

```
#define Default_V 2494.f
#define Ccca 0.4f
#define Cccr 10.f
#define Ra_ratio 1.25
#define Tumor_Cell_Radius 8.4f
#define Tumor_CC_Max_Dist 25.f

#define Tumor_O2_Proliferation_Thres 5.f
#define Tumor_O2_Proliferation_Sat 10.f // mmhg
#define Tumor_Necrosis_02_Thres 5.f
#define Tumor_Necrosis_02_Max 2.5f
#define Tumor_Necrosis_Rate (1.f / 6.0 / 60.0) // 6hour
#define Tumor_Apoptosis_rate (1.f / 7.0 / 24.0 / 60.0)
#define duration_pre2post (13.f * 60) // premitotic to postmitotic
#define duration_post2q (2.5f * 60) // postmitotic to mature
```

Data structure of cell

We adopted the “Ki67 advanced” cell model from PhysiCell (https://doi.org/10.1371/journal.pcbi.1005991). Data structure for cell is defined in `./include/Cells.cuh.`

```
struct Cell {
bool sign = false;
int mesh_idx = -1;
float3 pos = { -1.f,-1.f,-1.f };
float3 force = { 0.f,0.f,0.f };
float3 oldforce = { 0.f,0.f,0.f };
float r = -1.f;  // in um
// total volume
float V = -1.f;
// fluid volume
float Vf = -1.f;
// cytoplasmic volume
float Vcs = -1.f;
// nuclear solid volume
float Vns = -1.f;
int phase = -1;
float cell_clock = 0.f;
__host__ __device__ float O2_consume (float O2);
__host__ __device__ void Volume_update();
__host__ __device__ bool Phase_update_ki67adv(float rand, float O2);}
```

The value **sign** is used to indicate whether this is a living cell. **mesh_idx** is used for voxel sorting. **oldforce** records the previous aggregated force experienced by the cell for the second-order Adam-Bashforth method position update.

$$x_i(t+\Delta t) = x_i(t) + \frac{\Delta t}{2}\left(3 \cdot v_i(t) - v_i(t-\Delta t)\right)$$

The **phase** value indicates the cell phase.

0	1	2	3	4	5
quiescent	**premitotic**	**postmitotic**	**apoptotic**	**early necrotic**	**late necrotic**

The phase transition is handled by function `Phase_update_ki67adv.` This function directly updates the phase and returns a `bool` value indicating whether this cell is about to divide.
As shown in the data structure, we divide the total cell volume **V** into the fluid volume **Vf** and the solid biomass volume Vs. The solid biomass volume is further divided into total nuclear volume **Vns** and cytoplasmatic volume **Vcs**. **Vf**, **Vns**, and **Vcs** are updated individually by function `Volume_update()`, while **V** is simply the sum.

The oxygen consumption of cell is calculated by function `O2_consume.` This function returns a cell volume and voxel volume normalized consumption rate.

Proliferation and death kernel

This section is about how Gell handles cell birth and death. These processes both changes the cell data storage and access in the vector, so they are handled specially. The related code is in file `./include/PDkernel.cuh.`

There are two main kernels in this file, the **CellBirth_kernel** and the **CellDeath_kernel**. Both kernels return the current cell number.

```
int  CellBirth_kernel(float*  p,  thrust::device_vector<Cell>&  GpuCell,  curandState*
curand_states, int currentnum, int* gcurrentnum)
int CellDeath_kernel(thrust::device_vector<Cell>& GpuCell, curandState* curand_states,
int currentnum, int* gcurrentnum)
```

Cell volume update

In **CellBirth_kernel**, we first update the cell volume using `void Cell::Volume_update().`
In the `void Cell::Volume_update()` function, the desired volume is noted as `Vx_0`, related volume change rate is `rx`, and `ff` represents the desired fluid volume fraction of the cell.

```
Vf_0 = V * ff;  // 1871
Vf = Vf * (1.f - Biology_dt * rf) + rf * Biology_dt * Vf_0;
Vns = Vns * (1.f - Biology_dt * rns) + rns * Biology_dt * Vns_0;
Vcs = Vcs * (1.f - Biology_dt * rcs) + rcs * Biology_dt * Vcs_0;
V = Vf + Vns + Vcs;
```

During the premitotic phase, cells grow; during programmed death, cells shrink. The parameters involved are listed in the table. Parameters involved are listed in the table.

	Premitotic Phase	Postmitotic Phase/ Quiescent Phase	Apoptotic Phase	Early Necrotic Phase	Late Necrotic Phase
V_F	$0.7502V$	$0.7502V$	$0\ um^3$	V	$0\ um^3$
r_F	$3\ hour^{-1}$	$3\ hour^{-1}$	$3\ hour^{-1}$	$0.67\ hour^{-1}$	$0.05\ hour^{-1}$
V_{NS}	$270\ um^3$	$135\ um^3$	$0\ um^3$	$0\ um^3$	
r_{NS}	$0.33\ hour^{-1}$	$0.33\ hour^{-1}$	$0.35\ hour^{-1}$	$0.013\ hour^{-1}$	
V_{CS}	$976\ um^3$	$488\ um^3$	$0\ um^3$	$0\ um^3$	
r_{CS}	$0.27\ hour^{-1}$	$0.33\ hour^{-1}$	$1\ hour^{-1}$	$0.0032\ hour^{-1}$	

Phase transition, Proliferation, and Death

Cell phase update is handled by `bool Cell::Phase_update_ki67adv(float rand, float O2)`. This function directly changes the **`phase`** value of a living cell.

Living cells stochastically enter the **premitotic** phase for proliferation preparation (doubling). After a fixed **premitotic** duration, function `Cell::Phase_update_ki67adv` will return **`true`** to trigger the cell division, let the cell divide into two **postmitotic** daughter cells, each having half of the mother cell mass. To avoid race conditions, Gell changes total cell number using atomic function **`atomicAdd(gcurrentnum, 1)`**.

For cells finished the programmed death phase (apoptotic or necrotic), the function `Cell::Phase_update_ki67adv` will switch the **`sign`** to **`false`** to label the cell for the following removal from the current cell vector. The **CellDeath_kernel** will recognize these cells and remove them from the cell vector using **`thrust::remove_if`** function.

```
thrust::remove_if(GpuCell.begin(), GpuCell.begin() + currentnum, isDeadCell());
```

As listed in the transition rate table, the transition rates for stochastic transitions are listed as rate r and the transition rate for deterministic phase duration is noted as one over the fixed phase duration.

Transition Rate	Premitotic Phase	Postmitotic Phase	Quiescent Phase	Apoptotic Phase	Necrotic Phase
Premitotic Phase		$1/T_{prem}$	0	0	0
Postmitotic Phase	0		$1/T_{postm}$	0	0
Quiescent Phase	$r_{pro}(p_{oxy})$	0		r_{apop}	$r_{nec}(p_{oxy})$

Rates and durations are defined in `/include/Definitions.h`

```
#define Tumor_Necrosis_Rate (1.f / 6.0 / 60.0)
#define Tumor_Apoptosis_rate (1.f / 7.0 / 24.0 / 60.0)
#define Tumor_Proliferation_Rate (1.f/8.5/60.0)
#define duration_pre2post (13.f * 60) // premitotic to postmitotic
#define duration_post2q (2.5f * 60) // postmitotic to mature
```

The proliferation and necrosis transition rates are oxygen concentration dependent:

$$r_{pro}(P_{oxy}) = \begin{cases} r_{pro\_max} & P_{oxy} \geq Sa_{pro} \\ r_{pro\_max}\dfrac{P_{oxy} - Th_{pro}}{Sa_{pro} - Th_{pro}} & Th_{pro} < P_{oxy} < Sa_{pro} \\ 0 & P_{oxy} \leq Th_{pro} \end{cases}$$

$$r_{nec}(P_{oxy}) = \begin{cases} 0 & P_{oxy} \geq Th_{nec} \\ r_{nec\_max}\dfrac{Th_{nec} - P_{oxy}}{Th_{nec} - Sa_{nec}} & Sa_{nec} < P_{oxy} < Th_{nec} \\ r_{nec\_max} & P_{oxy} \leq Sa_{nec} \end{cases}$$

Oxygen dependent stochastic phase transition related parameters are defined in `/include/Definitions.h`

```
#define Tumor_O2_Proliferation_Thres 5.f
#define Tumor_O2_Proliferation_Sat 10.f
#define Tumor_Necrosis_02_Thres 5.f
#define Tumor_Necrosis_02_Max 2.5f
```

Force and movement kernel

Cell-cell mechanical interaction and cell movement related functions are defined in file `/include/FMkernel.cuh`. Gell creates a `MechanicsMesh_struct` structure for accelerated cell-cell interaction calculation. The domain is discretized into `Voxel_num * Voxel_num * Voxel_num` isotropic voxels and each has a Morton index (calculated using function `int Morton_encode(int3 ijk)` defined in `/include/BasicFunctions.cuh`). Array `int* key` records the Morton code of each cell and is updated in the function `__global__ void updateCellMeshIdx(Cell* cell, int cnum, int* key)`. Cells are sorted according to their Morton code before every force calculation.

```
thrust::sort_by_key(&key[0], &key[num], GpuCell.begin());
```

After sorting, cells in the same voxel are stored contiguously in the GPU memory, and cells in adjacent voxels are stored relatively close. Array `int* Mech_Mesh_s` and `int* Mech_Mesh_e` are arrays recording the index of the first and last cell in each voxel. To loop over cells in voxel `v` you can simply:

```
for (int i = Mech_Mesh_s[v]; i <= Mech_Mesh_e[v]; i++) {
      access cell data with cell[i];}
```

For force calculation, Gell will loop over the cells in the nearest 27 voxels of each target cell to calculate the aggregated force. Then the `movementkernel << <(num + BlockWidth1d - 1) / BlockWidth1d, BlockWidth1d >> > (GC, num)` kernel will update the cell position based on the force data:

```
pos.x += dt / 2.f * (3.f * f.x - of.x);
pos.y += dt / 2.f * (3.f * f.y - of.y);
pos.z += dt / 2.f * (3.f * f.z - of.z);
```

`float3 f` is the current aggregate force, and `float3 of` is the old aggregate force calculated in the previous call.

PDE kernel

The locally-one dimensional (LOD) method-based PDE solver (Readers can check BioFVM for more information) and the related mesh for diffusive substances are defined in file `/include/LODkernel.cuh`. Gell creates a `Mesh_struct` structure for accelerated PDE related calculation. The PDE for the system can be written as:

$$\frac{\partial \rho}{\partial t} = \nabla \cdot (D\nabla\rho) - \lambda\rho + S(\rho^* - \rho) - U\rho$$

ρ is the substance concentration, ρ* is the saturation concentration, D is the diffusion coefficient, λ is the

decay rate, S is the supply rate, U is the uptake rate. A first-order splitting method is first applied to split the righthand side into simpler operators: a supply and uptake operator and a diffusion-decay operator.

$$\begin{cases} \dfrac{\sigma-\rho^{n}}{\Delta t}=\nabla\cdot(D\nabla\sigma)-\lambda\sigma \\ \dfrac{\rho^{n+1}-\sigma}{\Delta t}=S(\rho^{n+1}-\sigma)-U\sigma \end{cases}$$

The supply and uptake operators are handled analytically by `meshupdatekernel << <gridSize3d, blockSize3d >> > (p, p_rate):`

```
p[meshi] = p[meshi] / (1.f - p_rate[meshi] * Diffusion_dt);
```

The consumption map `float* p_rate` is calculated after the mechanics/phase update by function `void PreLOD(Cell* cell, int num).`

The LOD method then splits the three-dimensional diffusion-decay operator into a series of related one-dimensional PDEs. Discretized using the finite volume method, the updated concentration of each strip of voxels for each direction can be obtained by solving the following equation (take one x strip for example) using the Thomas algorithm.

$$\begin{cases} \left(1+\frac{1}{3}\Delta t\lambda+\frac{\Delta t}{\Delta x^2}D\right)\circ\eta_{(0,y_f,z_f)}-\frac{\Delta t}{\Delta x^2}D\circ\eta_{(1,y_f,z_f)} & =\rho^{n}_{(0,y_f,z_f)} \\ -\frac{\Delta t}{\Delta x^2}D\circ\eta_{(n_x-1,y_f,z_f)}+\left(1+\frac{1}{3}\Delta t\lambda+2\frac{\Delta t}{\Delta x^2}D\right)\circ\eta_{(n_x,y_f,z_f)}-\frac{\Delta t}{\Delta x^2}D\circ\eta_{(n_x+1,y_f,z_f)} & =\rho^{n}_{(n_x,y_f,z_f)} \\ -\frac{\Delta t}{\Delta x^2}D\circ\eta_{(N_x-2,y_f,z_f)}+\left(1+\frac{1}{3}\Delta t\lambda+\frac{\Delta t}{\Delta x^2}D\right)\circ\eta_{(N_x-1,y_f,z_f)} & =\rho^{n}_{(N_x-1,y_f,z_f)} \end{cases}$$

This process is handled by three LODKernels `__global__ void LODkernel_x/y/z(float* p, float* E, float* F).`

Utilities

`RandomFunctions.cuh` contains random number involved functions, including the function to generate random numbers and the function to generate unit length 3D random vector.

```
__global__ void set_random_states(curandState* curand_states)
__device__ float3 randomvect(curandState* curand_states, int i, float scale = 1.f)
```

`BasicFunctions.cuh` contains trivial functions for various purposes and functions to encode/decode Morton code.

```
__device__ __host__ int Morton_encode(int3 ijk)
__device__ __host__ int3 Morton_decode(int m)
```

`Initialization.cuh` defines `Cell_sphere_Initialization(thrust::host_vector<Cell>& C, int N)` that initializes the cell spheroid with the given number `N` at the center of the simulation domain.

`FileSave.cuh` is for csv format file saving. Details about this file are described in the visualization section.

Other tips for development

General coding

CUDA basics

https://www.olcf.ornl.gov/wp-content/uploads/2019/12/01-CUDA-C-Basics.pdf

Check ubuntu version

lsb_release -d

Useful resource to write makefile

https://github.com/TravisWThompson1/Makefile_Example_CUDA_CPP_To_Executable

Gell specific

Define your own cell property

Go to `/include/Cells.cuh` and add variables/functions to `struct Cell`.

Your own cell-cell interaction

Go to `/include/FMkernel.cuh` and change the cell-cell mechanical interaction calculation from **line 93** to **line 111**. Please be aware that if your cell-cell interaction is long-range, you may need to adjust the voxel size or the voxel looping policies to cover the maximum interaction range.

Your own diffusive substance

Following the oxygen template, create a mesh for your substance x in **Main.cu line 65.**

```
Mesh_struct xMesh(x_Default_Concentration, x_Diffusion_coef, x_Decay_rate)
```

Define how the substance is secreted/consumed. If it is secreted/consumed by cell, add the related function in `/include/Cells.cuh` and create the variant of `void PreLOD(Cell* cell, int num)` and `void LODsolver(Cell* cell, int num) in /include/LODkernel.cuh.`

Provide the update policy in **Main.cu line 120**

```
xMesh.PreLOD_forx(GC, currentnum);
for (float dift = 0.0; dift < Mechanics_dt; dift += Diffusion_dt) {
      xMesh.LODsolver_forx (GC, currentnum);
      }
```