Project 1

Massive parallelization of PIC simulation using CUDA

Abstract:

An efficient massively parallel GPU program has been developed for speeding up one-dimensional electrostatic plasma simulations based on the Particle-in-Cell(PIC) method. A two-stream instability benchmark case has been implemented on CUDA, and the performance of the algorithm has been compared with the serial CPU code. The performance analysis demonstrates that the GPUs can accelerate the plasma simulation by order of magnitude. Simulating large numbers of particles and grids in a relatively shorter time due to GPU's massive parallelization can help us simulate very complex plasma physics leading to the advancement in plasma science.

Introduction and Methodology:

The particle-in-cell (PIC) method is a technique used to solve a particular class of partial differential equations. In this method, individual particles (or fluid elements) in a Lagrangian frame are tracked in continuous phase space. For the current case, an electrostatic particle in cell method has been implemented to simulate two-beam instability using C++ and CUDA programming. The PIC method has two parts, particle mover and electric field calculator. The particle mover updates the particle position and velocity using the following differential equations:

$$\frac{d\vec{x}}{dt} = \vec{v} \tag{1}$$

$$\frac{d\vec{v}}{dt} = \frac{q}{m}\vec{v} \tag{2}$$

Using Euler method, the above equation can be discretized as below. These equations are used to update the particle position and velocity for each time step.

$$\vec{v}^{n+1} = \vec{v}^{n} + \Delta t \left(\frac{q}{m}\vec{E}\right) \tag{3}$$

$$\vec{x}^{n+1} = \vec{x}^{n} + \Delta t\ \vec{v} \tag{4}$$

The code for equation 1 to 4 is implemented in C++ as shown in figure 1. Further, The electric field is calculated using following equation,

$$\vec{E} = -\vec{\nabla}\phi \tag{5}$$

For which following Poisson equation needs to be solved.

$$\vec{\nabla}^2\phi = \frac{\rho}{\epsilon_0} \tag{6}$$

Where, $\rho = e(Z_i n_i - n_e)$, and $n_e = n_0 exp\left(\frac{\phi-\phi_0}{k_B T_e}\right)$

```cpp
// Simulation main loop
auto start = high_resolution_clock::now(); //starting wall clock
for (int niter = 0 ; niter < Nt ; niter++)
{

    for(j = 0 ; j < N ; j++)
    {
        vel[j] += 0.5*dt * (-E[j]); // (1/2) kick  //particle velocity gets updated

        //  drift (and apply periodic boundary conditions)
        pos[j] += vel[j] * dt;          //particle position gets updated
        pos[j] = fmod(pos[j] , boxsize);    // if the particle moves out of box , it throws it to the intial as periodic does
                                            //i.e if new ps is 51 it will be updated as 1
        if(pos[j] < 0)
            pos[j] = pos[j] + 50.0;
    }                                   // if new posn is -ve it will throw correspondinly from other side of box

    // update accelerations
    getAcc( E, phi_grid, pos, N, Nx, boxsize, n0, Gmtx );

    for(j = 0 ; j < N ; j++)
    {
        vel[j] += 0.5*dt * (-E[j]); // (1/2) kick  // Giving two kicks, used as leap frog method.

        // update time
        t += dt;
    }
}
```

Figure 1: main loop of the function executed in C++

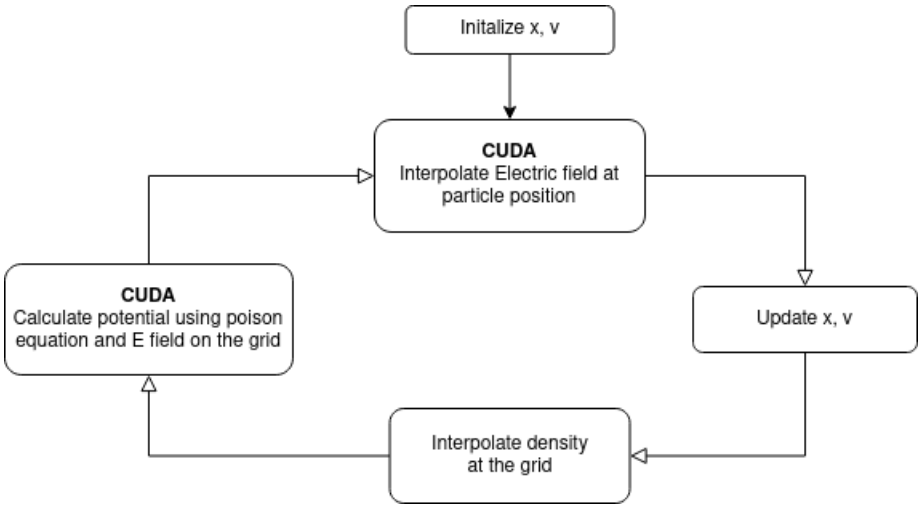


Figure 2: Algorithm

Figure 2 shows the flow chart of the PIC algorithm where the Poisson equation, the electric field update, and the part where the electric field from the nodes is mapped to the particles are executed in GPU whereas, the updating of the velocity and positions of the particles is done in the CPU. The Poisson equation is discretized using finite difference method and is given as:

$$\frac{\phi_{j+1} - 2\phi_j + \phi_{j-1}}{\Delta X^2} = \frac{\rho}{\epsilon_0} \tag{7}$$

$$\phi_j = \frac{1}{2}\Big(\phi_{j+1} + \phi_{j-1} - \Delta X^2 \frac{\rho_i}{\epsilon_0}\Big) \tag{8}$$

Where, ΔX is the distance between the two neighbouring nodes. ρ_i at the node is calculated by distributing each particle on the grid. Following equation has been used to distribute the density formulated as:

$$\rho_j += \Big(\frac{X_{j+1} - x_i}{\Delta X}\Big)\rho_i \tag{9}$$

$$\rho_{j+1} += \Big(\frac{x_i - X_j}{\Delta X}\Big)\rho_i \tag{10}$$

Here, ρ_j is density on the j^{th} node of the grid and ρ_i is density of the i^{th} particle. X_j is the j^{th} node x co-ordinate on grid.

After computing potential on grid, the electric field at grid is calculated as:

$$E_j = -\frac{\phi_{j+1} - \phi_{j-1}}{2\Delta X} \tag{11}$$

and electric field at the i^{th} particle position is interpolated from grid as

$$E_i = \Big(\frac{X_{j+1} - x_i}{\Delta X}\Big)E_j + \Big(\frac{x_i - X_j}{\Delta X}\Big)E_{j+1} \tag{12}$$

A brief description about Electrostatic method can be found in this PIC blog's link. After normalising and simplifying above equations, we get following equations that are implemented in the code.

$$\vec{v}^{n+1} = \vec{v}^{n} + \Delta t \ \Big(\frac{q}{m}\vec{E}\Big) \tag{13}$$

$$\vec{x}^{n+1} = \vec{x}^{n} + \Delta t \ \vec{v} \tag{14}$$

$$n_j += \Big(\frac{X_{j+1} - x_i}{\Delta X}\Big)n_i \tag{15}$$

$$n_{j+1} += \Big(\frac{x_i - X_j}{\Delta X}\Big)n_i \tag{16}$$

$$\phi_j = \frac{1}{2}\Big(\phi_{j+1} + \phi_{j-1} - \Delta X^2(n - n_0)\Big) \tag{17}$$

$$E_j = -\frac{\phi_{j+1} - \phi_{j-1}}{2\Delta X} \tag{18}$$

$$E_i = \Big(\frac{X_{j+1} - x_i}{\Delta X}\Big)E_j + \Big(\frac{x_i - X_j}{\Delta X}\Big)E_{j+1} \tag{19}$$

For the current project, the Jacobi method has been employed to solve for potential equation 17. The potential is iteratively updated till the difference between the values of consecutive iteration reaches a tolerance error value. This iterative method has been made significantly faster using GPU by implementing the code in CUDA language as shown in the snippet figure 3 of cuda kernal. This CUDA kernal named "Jacobi_Egrid_Kernel" uses the number of nodes to calculate its thread and block size. Once the potential field is obtained, electric field is updated using equation 18 in the CUDA wrapper as shown in figure 4. Moreover, equation 19, has also been implemented in a seprate CUDA kernal named "Eparticle_Kernel", where the threads are defined based on the number of particles.

```
// Jacobi solver to solve poission equation for potential
for (iter = 0 ; iter < stop_flag ; iter++)
{
    if( (col>0) && (col<(P.width-1)) )
        Egvalue = 0.5*(P.elements[col+1]+P.elements[col-1]-B.elements[col]);

    if( col == 0 )
        Egvalue = 0.5*(P.elements[1]+P.elements[P.width-1]-B.elements[0]);

    if( col == (P.width-1) )
        Egvalue = 0.5*(P.elements[0]+P.elements[P.width-2]-B.elements[P.width-1]);

    __syncthreads(); // This syncs all the threads after the iteration

    G.elements[col] = abs(P.elements[col] - Egvalue);

    if((iter % 100 == 0) && (col == 0))
    {
        float error = 0.0;
        for( int i = 0 ; i < P.width ; i++ )
            error += G.elements[i];

        if(error <= 1e-6)
            stop_flag = 0; //global variable is used here
    }

    P.elements[col] = 0.7*Egvalue + (1.0-0.7)*P.elements[col];
    // weighted average, (this helps in convergence)
}
```

Figure 3: Jacobi iterations carried out in CUDA kernal: Jacobi_Egrid_Kernel

```
// Calculate Efield on grid from potential
Egvalue = 0.0;
for (int icolG = 0 ; icolG < G.width ; icolG++)
    Egvalue += (G.elements[col*G.width + icolG]*P.elements[icolG]);


*d_iter_cache = iter; //to pass variable in iter

Eg.elements[col] = Egvalue;
```

Figure 4: Electric field updated in CUDA kernal: Jacobi_Egrid_Kernel

Results of two-stream instability using PIC solver simulation

The simulation output has been plotted in figure 7a at different time steps. Here, we get similar output for both the implementation i.e GPU and CPU.

Performance between GPU and CPU

Below is the performance comparison of similar code on GPU vs. CPU. One can see a significant speed-up with GPU as the number of nodes in the grid increases. This speed-up is because the entire code and operations corresponding to the mesh or grid have been exported to the GPU. It can be seen in figure 7a that the execution time of CPU increases exponentially with an increase in the number of nodes. In contrast, due to massive parallelization, the time required to execute a larger grid remains the same for GPU. This leads to enormous speed up as compared to CPU as shown in figure 7b, where for 500 nodes, the speed-up is as large as 12 times. It is to be noted that the number of particles in the comparison explained above is kept constant at 40000.

Further, when the grid is kept constant at 100 and as the number of particles are increased, it can be seen in figure 7c that there is not much of a distinct advantage of GPU over CPU. This was because the particle mover algorithm, i.e., equation 13 and 14 were not implemented on the GPU. Therefore to further enhance the performance of the code, we can take the numerical operations involved in equations 13 and 14 to GPU, which can be done as future work leading to further speed-ups.

```
// Each thread computes one element of Eg(Efield)
//Ep=Electric filed at particle posn, Eg= Electric field calc at grid from wrapper above
//W1,W2= left and right node weight, p1,p2=left and right node index of the particle
int col = blockIdx.x * blockDim.x + threadIdx.x;
// E[i] = weight_j[i] * E_grid[pidx[i]] + weight_jp1[i] * E_grid[pidx1[i]]
Ep.elements[col] = W1.elements[col] * Eg.elements[p1.elements[col]] + W2.elements[col] * Eg.elements[p2.elements[col]];
```

Figure 5: Electric field mapped from grid to particles in CUDA kernal: Eparticle_Kernel

Brief description of the structure of the files

The file structure for the project is as follows:

1. main.cpp: This file contains the main execution commands where entire execution of the code using C++ has been done. The code can run either CPU or GPU or both based on the preprocessor, i.e, flags (device_GPU or device_CPU) defined in the input_file.h file.

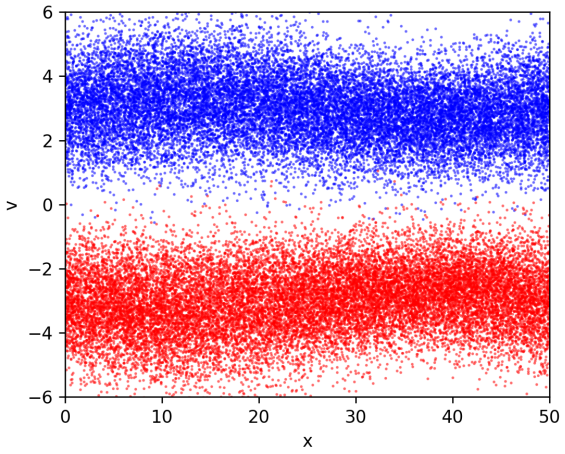
2. input_file.h: This files contains all the parameters required as inputs for the simulation like number of particles, time steps , number of nodes etc. It also contains all the thread size required as input to CUDA kernals.

3. Potential_Egrid_solver.cu: This file is a CUDA program which contains two kernal because simulation would require different thread sizes, one to do numerical operations on particles whereas other is used to do operations on the nodes. The required CUDA memory for different variables is allocated and copied here and then fed to the CUDA kernal.

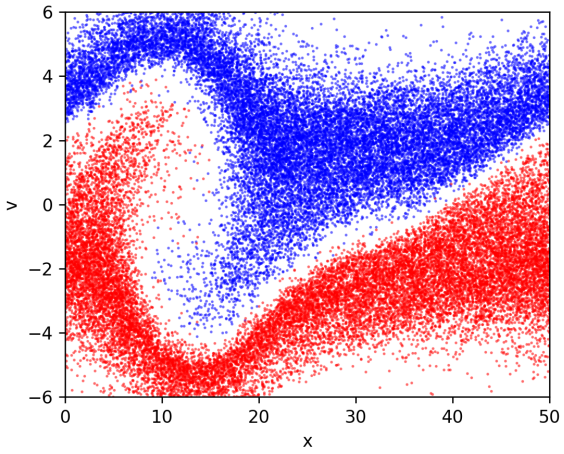
4. plot.py: This file is used to plot the text output generated by the main.cpp file.

Conclusion

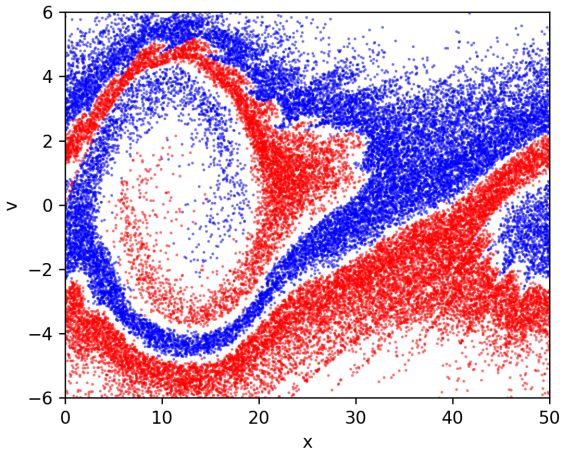
The Plasma science community has used the PIC since 1955[2], but the computational cost is prohibitively high. This is due to the involvement of large number of particles and the iterative methods employed to solve the potential equation. The use of GPU enables us to decrease this computational time by orders of magnitude by enabling massive parallelization. Using GPU, we can scale massively, allowing us to simulate very complex plasma physics and study them.



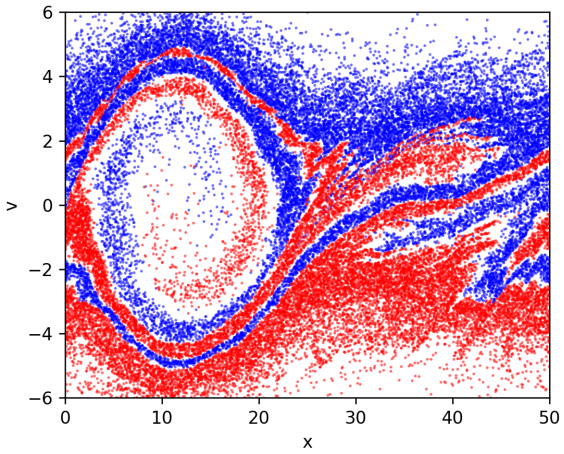
(a) t=0 s



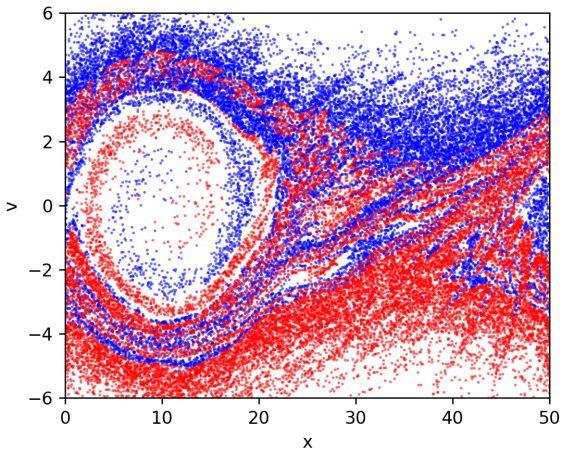
(b) t=10 s



(c) t=20 s

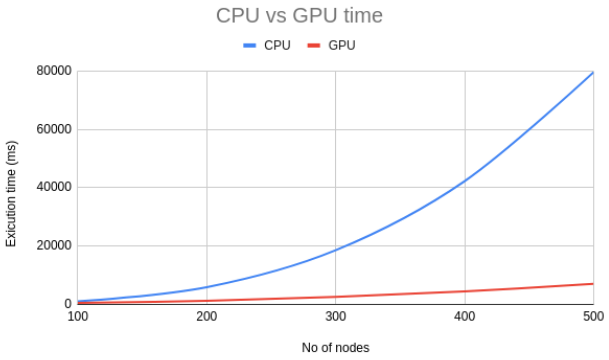


(d) t=30 s

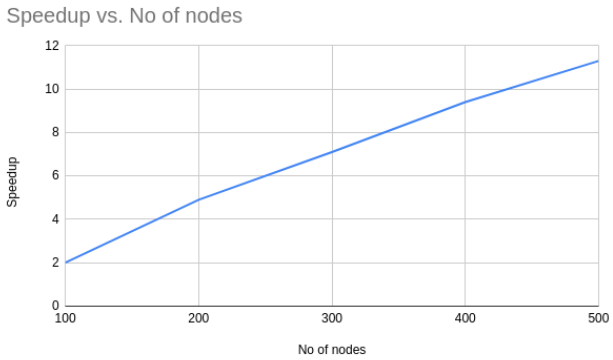


(e) t=50 s

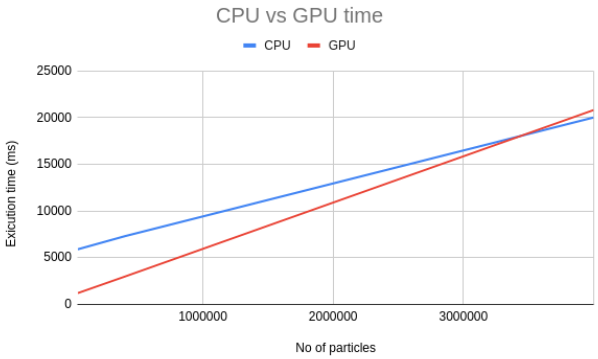
Figure 6: Time evolution of two-stream instability.



(a) Time vs nodes



(b) Speed-up of GPU compared to CPU vs nodes



(c) Time vs nodes

Figure 7: GPU vs CPU comparison.