CUDA PARALLELIZATION OF A 2D PARTICLE-IN-CELL CODE

STEVEN TRAN AND BRIAN TRAN

Contents

1. Introduction

In this project, we examine the parallelization of a 2D particle-in-cell (PIC) code utilizing CUDA in C. The particle-in-cell method is a numerical algorithm for simulating an ionized plasma whose motion generates an electromagnetic field. Since the PIC method requires many particles and also sufficiently fine grids, one would expect significant runtime improvement in parallelizing this algorithm. In this project, we investigate implementing a 2D PIC algorithm and subsequently parallelizing it. We demonstrate the speedup of the parallelized code versus the serial code in a scaling test.

For background, we will first discuss the continuous theory of this system. The starting point for this system is a description of the plasma, which we take to be non-relativistic. The plasma is described by a distribution function $f(t, \vec{r}, \vec{v})$ which gives the phase space density of particles at each point $(\vec{r}, \vec{v}) \in \mathbb{R}^n \times \mathbb{R}^n$ at time t. This distribution function is advected along the flow of the

Phys 244: Parallel Computing in Science and Engineering, Spring 2021. Final Project.

E-mail address: `s4tran@eng.ucsd.edu, b3tran@ucsd.edu`.

particles, given by the Vlasov equation

$$0 = \frac{d}{dt} f(t, \vec{r}, \vec{v}) = \frac{\partial}{\partial t} f + \dot{\vec{r}} \cdot \nabla_{\vec{r}} f + \dot{\vec{v}} \cdot \nabla_{\vec{v}} f.$$

The motion of the plasma generates an internal electromagnetic field; the evolution of the electromagnetic field is governed by Maxwell's equations (written here in dimension $n = 3$)

$$\begin{aligned} \nabla \times \vec{B} - \partial_t \vec{E} &= \vec{J}, \\ \nabla \times \vec{E} + \partial_t \vec{B} &= 0, \\ \nabla \cdot \vec{E} &= \rho, \\ \nabla \cdot \vec{B} &= 0, \end{aligned}$$

where we set the physical constants $\epsilon = \mu = 1$ (and hence the speed of light $c = 1$). These two systems are coupled by the usual relations between charged particles and electromagnetic fields. Namely, the particles evolve under the electromagnetic field by the Lorentz force law $\dot{\vec{v}} = \frac{q}{m}(\vec{E} + \vec{v} \times \vec{B})$ (where q and m are the charge and mass of a particle, respectively) and the relation $\dot{\vec{r}} = \vec{v}$. In turn, the electromagnetic field evolves under the current and charge densities determined by the distribution function, given by integrating over velocity space:

$$\begin{aligned} \vec{J}(t, \vec{r}) &= q \int d^n v \vec{v} f(t, \vec{r}, \vec{v}), \\ \rho(t, \vec{r}) &= q \int d^n v f(t, \vec{r}, \vec{v}). \end{aligned}$$

Since we are interested in a problem with full electromagnetic interaction, we take the simplest case where a magnetic field can exist, which is the spatial dimension $n = 2$. In this case, the motion of the particles in the plane allow for a magnetic field to exist normal to the plane, which we denote B_z. In this case, the components of the evolutionary equations in the Maxwell system can be written

$$\begin{aligned} \partial_y B_z - \partial_t E_x &= J_x, \\ -\partial_x B_z - \partial_t E_y &= J_y, \\ \partial_x E_y - \partial_y E_x + \partial_t B_z &= 0. \end{aligned}$$

Similarly, the components of the Lorentz force law can be written

$$\begin{aligned} \dot{v}_x &= \frac{q}{m}(E_x + v_y B_z), \\ \dot{v}_y &= \frac{q}{m}(E_y - v_x B_z). \end{aligned}$$

In Section 2, we will discuss the specifics of the discretization of this system. To conclude the introductory section, we provide a brief overview of the PIC method. The basic idea is, instead of evolving the continuum distribution function, one instead samples the initial distribution function N times and represents the distribution function by this collection of sampled "particles". The particles are updated according to a numerical integrator applied to the Lorentz force law and the relation $\dot{\vec{r}} = \vec{v}$ (we will use the leapfrog method), where the electromagnetic field at the particle is determined by interpolation of the discrete electromagnetic field. By assuming that each particle carries some weight in phase space (that is, each particle is not localized to its exact position but is spread out in phase space), one can determine the charge and current densities, a process known as charge/current deposition. Subsequently, with the charge/current deposited, one chooses a numerical integration scheme to evolve the electromagnetic fields (we will use the Yee scheme). This process is continued recursively in time until some final stopping time T. The workflow for the PIC method is shown in Figure 1. For a detailed overview of the PIC method, see for example [1].

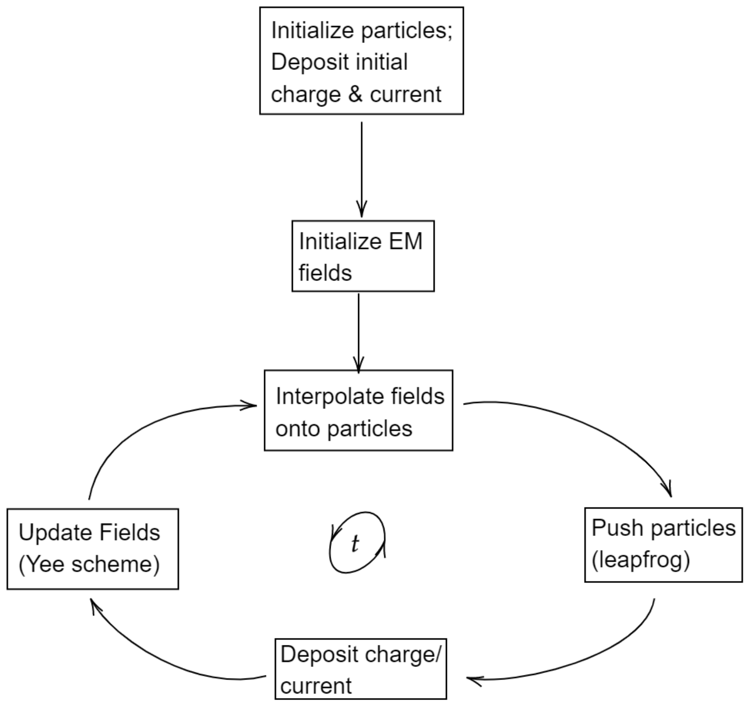
Initialize particles; Deposit initial charge & current
Initialize EM fields
Interpolate fields onto particles
Push particles (leapfrog)
Deposit charge/ current
Update Fields (Yee scheme)
t

Figure 1. Workflow for the particle-in-cell method

2. Methodology

In this section, we elaborate on the specifics of the PIC discretization of the Maxwell–Vlasov system and subsequently discuss the parallelization of the PIC algorithm.

We take the domain of our simulation to be a rectangle, $[0, L_x] \times [0, L_y]$; we subdivide each interval into N_x and N_y equal length subintervals, respectively, and denote $\Delta x = L_x/N_x, \Delta y = L_y/N_y$. We take reflecting boundary conditions for the particles; namely, if the x position of a particle is pushed to a value $-s < 0$, then it gets reflected $s > 0$ or if it is pushed to a value $L_x + s > L_x$, then it gets reflected to $L_x - s$ (and the velocity v_x gets reversed); the boundary condition is analogous for the y direction.. For the electromagnetic field, we set the potential to be zero on the boundary so that no longitudinal fields can exist on the boundary. Thus, on the boundary, only electric fields orthogonal to the boundary can exist; note also that with the given boundary conditions, the magnetic field (out of the plane) can exist on the boundary. This is shown in Figure 2.

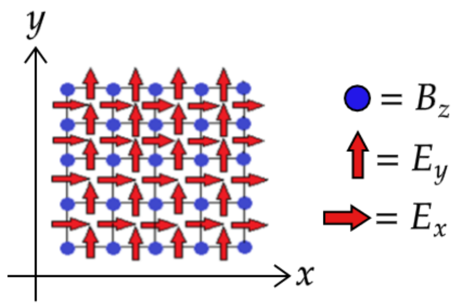

FIGURE 2. Electric and magnetic fields on the $x - y$ grid

The system is evolved from $t = 0$ to a final time $t = T$ in steps Δt. In order for the Yee scheme (discussed below) to be stable, one requires the step-size to satisfy CFL condition

$$\Delta t \leq \frac{1}{\sqrt{\Delta x^{-2} + \Delta y^{-2}}}.$$

To satisfy this condition throughout all tests, we take $\Delta x = \Delta y$ and thus we need $\Delta t \leq \Delta x/\sqrt{2}$.

2.1. **Initialization.**

The first step in the PIC algorithm is to initialize the particles and electromagnetic fields. To initialize the particles, one starts with a given initial distribution $f_0(\vec{r}, \vec{v}) = f(0, \vec{r}, \vec{v})$. One then randomly samples this distribution N times to produce a collection of particles described by their initial positions and velocities, $\{\vec{r}^{\alpha}(0), \vec{v}^{\alpha}(0)\}_{\alpha=1}^{N}$, where the index α labels the particle. To actually sample such a distribution, we utilize the rejection method. Namely, instead of sampling directly from f_0, we sample a function which we know how to sample (the uniform distribution); subsequently, one additionally randomly samples a random number $r \in [0, \max(f)]$, and if r is less than or equal to the value of f_0 at the randomly sampled point, we accept this sample. Otherwise, we reject and sample again. This process continues until N samples have been accepted. This is implemented in the `while` loop under the comment

```
// initialize particles by sampling f0(x,y,vx,vy)
// using the rejection method
```

in the serial code `twod_pic.c`.

For the electromagnetic field, there are various ways to initialize the field, depending on the problem of interest. One way to initialize is to utilize the constraint $\nabla \cdot \vec{E} = \rho, \nabla \cdot \vec{B} = 0$ in order to determine the fields. Namely, after initializing the particles, one can deposit the charge to determine ρ and use, for example, the Jacobi iteration to solve Poisson's equations $\nabla^2 \phi = -\rho$ and determine $\vec{E}$ from ϕ by a finite difference applied to $\vec{E} = -\nabla \phi$. Then, one can also take $\vec{B} = 0$ to satisfy the above constraints. This is the approach that we took in the 1D PIC code `oned_pic.c` (this code was submitted in the Appendix of the TPR, but we do not test the parallelization of this code). For the 2D code, we instead initialize using $\vec{E} = 0 = \vec{B}$. Although this violates the initial constraints $\nabla \cdot \vec{E} = \rho, \nabla \cdot \vec{B} = 0$, one can visualize this physically as follows. Suppose we have determined the electromagnetic field generated from the plasma and introduced a background electromagnetic field which exactly cancels the internal electromagnetic field. At time $t = 0$, we turn the background source off and we subsequently allow the plasma to evolve with initial conditions $\vec{E} = 0 = \vec{B}$. Mathematically, choosing the initial conditions to violate the constraints is fine (i.e., well-posed), since the two evolutionary Maxwell's equations form a first-order symmetric hyperbolic system and are well-posed as an initial-boundary value problem. We choose this initial condition to demonstrate the Weibel instability, which we will elaborate on in Section 3.

2.2. **Particle Push and Current Deposition.** To update the particle position, we apply the leapfrog method to the system

$$\begin{aligned}
\dot{x}^\alpha(t) &= v_x^\alpha(t),\\
\dot{y}^\alpha(t) &= v_y^\alpha(t),\\
\dot{v}_x^\alpha(t) &= \frac{q}{m}[E_x(x^\alpha(t), y^\alpha(t)) + v_y(t)B_z(x^\alpha(t), y^\alpha(t))],\\
\dot{v}_y^\alpha(t) &= \frac{q}{m}[E_y(x^\alpha(t), y^\alpha(t)) - v_x(t)B_z(x^\alpha(t), y^\alpha(t))].
\end{aligned}$$

To do this, recall our time interval $[0, T]$ is divided into subintervals with spacing Δt, with nodes which we denote $t^n = n\Delta t$. At the center of each node, we place a half-integer node $t^{n+1/2} = (n + 1/2)\Delta t$. The leapfrog algorithm is given by placing the velocities at the half-integer nodes and the positions at the integer nodes; subsequently, one evolves the velocities in time to the next half-integer node by forward difference and then evolves the positions in time to the next integer node, using the newly calculated velocities. For the above system, this is given by

$$v_x^\alpha(t^{n+1/2}) = v_x^\alpha(t^{n-1/2}) + \Delta t\frac{q}{m}[E_x(x^\alpha(t^n), y^\alpha(t^n)) + v_y^\alpha(t^{n-1/2})B_z(x^\alpha(t^n), y^\alpha(t^n))], \tag{2.2.1a}$$

$$v_y^\alpha(t^{n+1/2}) = v_y^\alpha(t^{n-1/2}) + \Delta t\frac{q}{m}[E_y(x^\alpha(t^n), y^\alpha(t^n)) - v_x^\alpha(t^{n-1/2})B_z(x^\alpha(t^n), y^\alpha(t^n))], \tag{2.2.1b}$$

$$x^\alpha(t^{n+1}) = x^\alpha(t^n) + \Delta t\ v_x^\alpha(t^{n+1/2}), \tag{2.2.1c}$$

$$y^\alpha(t^{n+1}) = y^\alpha(t^n) + \Delta t\ v_y^\alpha(t^{n+1/2}). \tag{2.2.1d}$$

In the serial code `twod_pic.c`, this is implemented in the function `void Particle_Push(...)`. The leapfrog stepping is shown schematically in Figure 3.

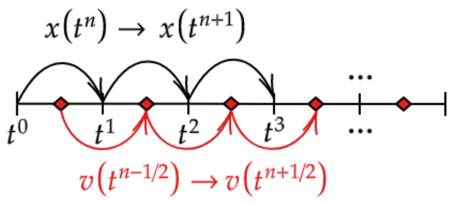
$x(t^n) \rightarrow x(t^{n+1})$
t^0
t^1
t^2
t^3
$v(t^{n-1/2}) \rightarrow v(t^{n+1/2})$

FIGURE 3. Leapfrog stepping for the particle push

As can be seen from equations (2.2.1a) and (2.2.1b), we need the value of the electromagnetic fields at the particle position, whereas the discretized electromagnetic fields are only defined at nodes on the grid. To determine the field values at the particle, we will have to interpolate the electromagnetic field onto the particle location. We will discuss this in Section 2.3, after discussing the Yee scheme used to evolve the fields. Also, note that for each particle α, the equations (2.2.1a)-(2.2.1d) are independent of any other particle $\beta \neq \alpha$, and hence are amenable to parallelization, which we will discuss in Section 2.5.

There are several advantages of using the leapfrog method as the algorithm for the particle pushing. First, note that the leapfrog method is a symplectic integrator which preserves the Hamiltonian flow associated to the system. Roughly, this means that phase space areas are preserved numerically, which means that no artificial (numerical) expansion or contraction is introduced into the evolution of the particle distribution (for a more detailed discussion, see [4]). Such symplectic integrators also have excellent energy conservation properties (although an integrator cannot both be symplectic and energy-preserving, symplectic integrators are near-energy-preserving); this will allow us to physically demonstrate the Weibel instability. Furthermore, the velocities are defined on the half-integer time steps and thus the deposited current will be defined on the half-integer time steps; thus, we have the advantage of already having the correct time step for the current in the Yee scheme discretization of Maxwell's equations.

To conclude this section, we discuss the deposition of current after the particles have been updated. After the particle push, consider a single particle with position (x, y) and velocity (v_x, v_y). Label the nodes of the spatial grid by (i, j) where $x_i = i\Delta x, y_j = j\Delta y$. Assume that the particle is located in a rectangle in the grid with nodes $\{(i, j), (i + 1, j), (i, j + 1), (i + 1, j + 1)\}$. The values of the bottom left node (i, j) are simply given by

```
lowerx[i] = (int)floor(x[i]/dx);
lowery[i] = (int)floor(y[i]/dy);
```

where `dx`, `dy` represent Δx and Δy respectively and `i` labels the i^{th} particle; keeping track of these indices allows us to simply deposit the current by interpolation. As we will see in the Yee scheme, the currents in the x direction live on edges of constant x and currents in the y direction live on edges of constant y. To deposit the current, we first initialize the current at the new time step to be zero. Then, we linearly interpolate v_x and v_y onto their respective edges and add these to the stored current (scaled by the charge q); we then repeat this over all particles. This process is summarized in Figure 4. In the serial code `twod_pic.c`, this is implemented in the function `void Deposit_Current(...)`.

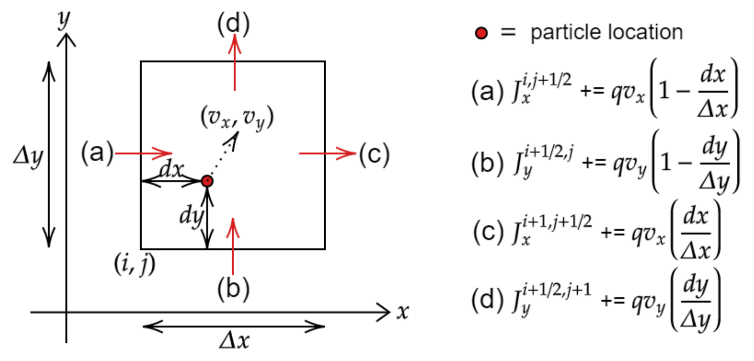
y
(d)
● = particle location
(v_x, v_y)
Δy
(a)
(c)
dx
dy
(i, j)
(b)
x
Δx
(a) $J_x^{i,j+1/2}$ += $qv_x\left(1 - \frac{dx}{\Delta x}\right)$
(b) $J_y^{i+1/2,j}$ += $qv_y\left(1 - \frac{dy}{\Delta y}\right)$
(c) $J_x^{i+1,j+1/2}$ += $qv_x\left(\frac{dx}{\Delta x}\right)$
(d) $J_y^{i+1/2,j+1}$ += $qv_y\left(\frac{dy}{\Delta y}\right)$

FIGURE 4. Schematic for current deposition corresponding to a single particle

2.3. **Field Update and Interpolation.** In this section, we discuss the remaining steps to complete the PIC discretization of the full Maxwell–Vlasov system, which is to discretize and update the fields and subsequently, interpolate the fields onto the particles.

We discretize the electric and magnetic fields using the Yee scheme, which is as follows. Consider a cell on the spacetime grid, where the lowest node in the cell (in the x, y, and t directions) is given by (x_i, y_j, t^n). The value of the electric field is placed on integer time steps, where E_x is specified along the midpoint of edges of constant x and E_y is specified along the midpoint of edges of constant y; these are denoted $E_x(t^n, x_i, y_{j+1/2})$ and $E_y(t^n, x_{i+1/2}, y_j)$ respectively. The value of the magnetic field is placed on half-integer time steps, located along the integer nodes of the spatial grid, which is denoted $B_z(t^{n+1/2}, x_i, y_j)$. The currents, as discussed in the previous section, are located on the same spatial locations as the electric field; however, the current is stored on half-integer time steps, like the magnetic field. The nodal values for the electric and magnetic field in a single cell of the spacetime grid is shown in Figure 5.

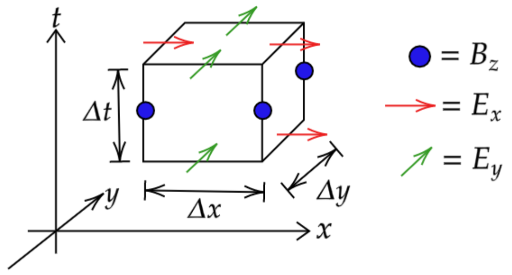

FIGURE 5. Electric and magnetic field locations in a spacetime cell for the 2D Yee Scheme

The Yee scheme then approximates the evolutionary Maxwell's equations,

$$\begin{aligned} \partial_y B_z - \partial_t E_x &= J_x, \\ -\partial_x B_z - \partial_t E_y &= J_y, \\ \partial_x E_y - \partial_y E_x + \partial_t B_z &= 0, \end{aligned}$$

via the finite-difference stencil

$$\begin{aligned} &(2.3.1a)\quad E_x(t^{n+1}, x_i, y_{j+\frac{1}{2}}) \\ &\qquad = E_x(t^n, x_i, y_{j+\frac{1}{2}}) + \frac{\Delta t}{\Delta y}[B_z(t^{n+\frac{1}{2}}, x_i, y_{j+1}) - B_z(t^{n+\frac{1}{2}}, x_i, y_j)] - \Delta t J_x(t^{n+\frac{1}{2}}, x_i, y_{j+\frac{1}{2}}), \\ &(2.3.1b)\quad E_y(t^{n+1}, x_{i+\frac{1}{2}}, y_j) \\ &\qquad = E_y(t^n, x_{i+\frac{1}{2}}, y_j) - \frac{\Delta t}{\Delta x}[B_z(t^{n+\frac{1}{2}}, x_{i+1}, y_j) - B_z(t^{n+\frac{1}{2}}, x_i, y_j)] - \Delta t J_y(t^{n+\frac{1}{2}}, x_{i+\frac{1}{2}}, y_j), \\ &(2.3.1c)\quad B_z(t^{n+\frac{3}{2}}, x_i, y_j) \\ &\qquad = B_z(t^{n+\frac{1}{2}}, x_i, y_j) - \frac{\Delta t}{\Delta x}[E_y(t^{n+1}, x_{i+\frac{1}{2}}, y_j) - E_y(t^{n+1}, x_{i-\frac{1}{2}}, y_j)] \\ &\qquad\quad + \frac{\Delta t}{\Delta y}[E_x(t^{n+1}, x_i, y_{j+\frac{1}{2}}) - E_x(t^{n+1}, x_i, y_{j-\frac{1}{2}})]. \end{aligned}$$

For a thorough discussion of the Yee scheme, see for example [3]. In the serial code `twod_pic.c`, this is implemented in the function `void Yee_Field(...)`. With these equations updating the fields to the new time step, the only step left to complete the discretized system is to interpolate the fields onto the particle positions, so that the particles can be pushed by equations (2.2.1a)-(2.2.1d).

The interpolation of the electric field onto the particle location is given by linear interpolation of the E_x field with respect to the x position of the particle in a cell and linear interpolation of the E_y field with respect to the y position of the particle in a cell. This is shown in Figure 6.

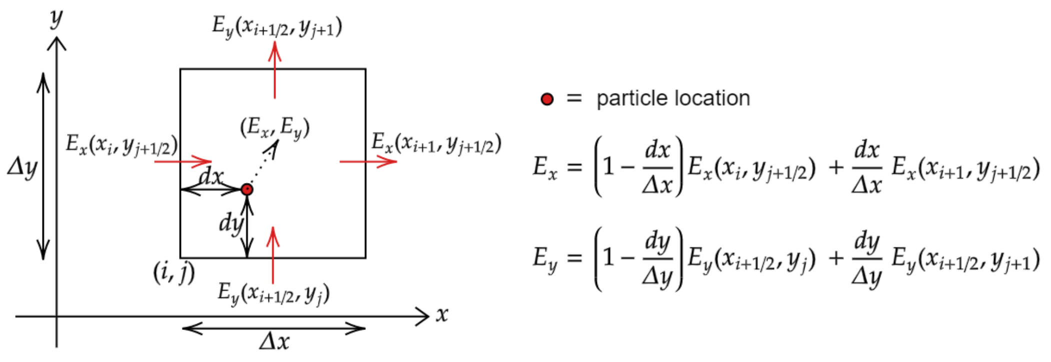
y
$E_y(x_{i+1/2}, y_{j+1})$
● = particle location
$E_x(x_i, y_{j+1/2})$
(E_x, E_y)
$E_x(x_{i+1}, y_{j+1/2})$
Δy
dx
dy
(i, j)
$E_y(x_{i+1/2}, y_j)$
x
Δx
$E_x = \left(1 - \frac{dx}{\Delta x}\right) E_x(x_i, y_{j+1/2}) + \frac{dx}{\Delta x} E_x(x_{i+1}, y_{j+1/2})$
$E_y = \left(1 - \frac{dy}{\Delta y}\right) E_y(x_{i+1/2}, y_j) + \frac{dy}{\Delta y} E_y(x_{i+1/2}, y_{j+1})$

FIGURE 6. Interpolation of the electric field onto a particle

In the serial code `twod_pic.c`, this is implemented in `double Exatparticle(...)` and `double Eyatparticle(...)`.

For the magnetic field, the four vertices of the cell each carry a nodal value of the magnetic field. Hence, the value of the magnetic field at a position in the cell is given by bilinear interpolation. This is shown in Figure 7.

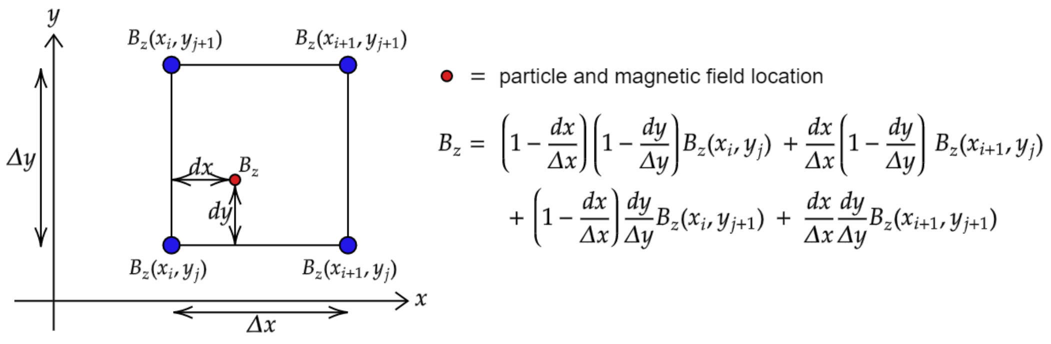
y
$B_z(x_i, y_{j+1})$
$B_z(x_{i+1}, y_{j+1})$
● = particle and magnetic field location
Δy
dx
B_z
dy
$B_z(x_i, y_j)$
$B_z(x_{i+1}, y_j)$
x
Δx
$B_z = \left(1 - \frac{dx}{\Delta x}\right)\left(1 - \frac{dy}{\Delta y}\right) B_z(x_i, y_j) + \frac{dx}{\Delta x}\left(1 - \frac{dy}{\Delta y}\right) B_z(x_{i+1}, y_j) + \left(1 - \frac{dx}{\Delta x}\right)\frac{dy}{\Delta y} B_z(x_i, y_{j+1}) + \frac{dx}{\Delta x}\frac{dy}{\Delta y} B_z(x_{i+1}, y_{j+1})$

FIGURE 7. Bilinear interpolation of the magnetic field onto a particle

In the serial code `twod_pic.c`, this is implemented in `double Hatparticle(...)`.

2.4. **Energy Calculation.** In our tests, we are interested in the particle and field energies. For a distribution $f(x, y, v_x, v_y)$, the particle kinetic energy is given by

$$K = \int d^2x \int d^2v \frac{1}{2} m(v_x^2 + v_y^2) f(x, y, v_x, v_y).$$

In the PIC method, each particle represents a contribution to f given by a finite size in phase space, which are represented as "shape functions", so that in terms of the collection of particles $\{x^\alpha, y^\alpha, v_x^\alpha, v_y^\alpha\}_{\alpha=1}^N$, the distribution function is

$$f(x, y, v_x, v_y) = \sum_\alpha S_x(x - x^\alpha) S_y(y - y^\alpha) S_{v_x}(v_x - v_x^\alpha) S_{v_y}(v_y - v_y^\alpha).$$

Take the shape functions in the velocity directions to be delta functions and the shape functions in space to be functions $S_x(x - x^\alpha), S_y(y - y^\alpha)$ where $S_x(x), S_y(y)$ are centered at 0, and the integral of S_x over x is Δx and the integral of S_y over y is Δy. Then,

$$K = \int d^2x \int d^2v \frac{1}{2}m(v_x^2+v_y^2)f(x,y,v_x,v_y) = \Delta x \Delta y \left[\sum_\alpha \frac{1}{2}m[(v_x^\alpha)^2+(v_y^\alpha)^2]\right].$$

The total field energy is given by the expression

$$\mathcal{E} = \int_\Omega \left[\frac{1}{2}E_x^2 + \frac{1}{2}E_y^2 + \frac{1}{2}B^2\right] dxdy,$$

where Ω is the spatial domain $[0, L_x] \times [0, Ly]$. This can be approximated by quadrature, corresponding to summing all of the squared nodal values of E_x, E_y, B, multiplied by $\Delta x \Delta y$. This gives the expression for the field energy

$$\mathcal{E} = \Delta x \Delta y \left[\frac{1}{2}\sum_{\text{nodes of } E_x} E_x(\text{node})^2 + \frac{1}{2}\sum_{\text{nodes of } E_y} E_y(\text{node})^2 + \frac{1}{2}\sum_{\text{nodes of } B} B(\text{node})^2\right].$$

The energy calculation for both the particle and field variables is implemented in the serial code `twod_pic.c` in the function `void Energy(...)` (note: both of the expressions for K and $\mathcal{E}$ carry a common factor of $\Delta x \Delta y$, so we omit this in the code).

The discussion thus far gives the complete PIC discretization of the Maxwell–Vlasov system as well as computing the energies; we implemented this serially in the code `twod_pic.c`. In the next section, we will discuss parallelizing the algorithm.

2.5. **Parallelization.** To parallelize the serial code, there are four processes inside the main time loop that can be parallelized; namely, we have in the serial code:

```
Particle_Push(...) // depends on particle and field variables,
Deposit_Current(...) // depends on particle and current variables,
Yee_Field(...) // depends on field and current variables,
Energy(...) // depends on particle and field variables.
```

Note that one could also parallelize the initialization step, and in particular, the sampling of the initial distribution (parallelized over the number of particles). However, this is only done initially (for $t = 0$) and not at every iteration of the time loop; consequently, the computational saving is not as significant as parallelizing the processes inside the time loop (that is, the computational savings from parallelizing the processes inside the main time loop get multiplied by a factor of the number of time steps taken, whereas the computational savings from the initialization does not). Thus, we do note parallelize the initialization procedure.

To begin parallelizing the serial code, we first allocate device memory using `cudaMalloc` corresponding to the host memory variables. For the particles, we have the positions, velocities, and the lower x and y indices; so we allocate `d_x, d_y, d_vx, d_vy, d_lowerx, d_lowery`. For the currents, we allocate `d_Jx, d_Jy`, and for the fields, we allocate `d_Hz, d_Ex, d_Ey`. For the energies, we allocate `d_fieldenergy` and `d_particleenergy`. We now discuss the parallelization of the four above processes; the parallel code is implemented in the file `twod_pic_cuda.cu`.

Parallel particle push. In the parallel code, the serial function `void Particle_Push` is replaced by the kernel `__global__ void Particle_Push` which takes as inputs the pointers to the device memory particle and field variables. There are three main differences between the serial function and the kernel.

The obvious main difference is of course using the grid-stride indexing to access elements of the device memory so that the kernel is run in parallel across the GPU's cores. Note that the particle variables are one-dimensional arrays, so when we call the kernel inside the time loop, we use `<<<grid1D, block1D>>>`; these are set by `grid1D(GS)` and `block1D(BS)`, where the gridsize is given by `GS` by `GS` blocks and the blocksize is given by `BS` by `BS` threads.

The second difference is that in the serial function, we utilize the variables `xnew, ynew, vxnew, vynew` to store the updated particle variables. However, in order to utilize less device memory for the parallel code, we instead update the old variables to the value after the push. If we look at the particle push equations (2.2.1a)-(2.2.1d), we see that we do not actually need the new variables if inside the kernel, we copy the initial value of $v_x^\alpha(t^{n-1/2})$, since this value is needed to update v_y^α. Thus, the kernel starts by storing the current value of v_x (which is called `vx_temp` in the kernel), updating v_x, updating v_y, and then updating x and y.

Finally, the last difference is that in the serial function, we call the functions `double Exatparticle, double Eyatparticle, double Hatparticle`, in order to interpolate the fields onto the particles, as discussed in Section 2.3. For the serial code, these functions are called on the host CPU. However, in the kernel, we have to call these functions on the device. Hence, we replace these for functions that are called on the device; e.g., the interpolation of the E_x field is given by `__device__ double Exatparticle(...)` and similarly for the others.

Parallel current deposition. In the parallel code, the serial function `void Deposit_Current` is replaced by the kernel `__global__ void Deposit_Current`, which takes as inputs pointers to the device memory particle and current variables. The main difference is using the grid-stride indexing as in the particle push. Since the deposition of current loops over the particles (to deposit the current for each particle), we call the kernel with `<<<grid1D, block1D>>>`.

Parallel field update. In the parallel code, the serial function `void Yee_Field` is replaced by the two kernels `__global__ void Yee_E_Field` and `__global__ void Yee_H_Field` which update the electric and magnetic fields respectively (note that in our discussion, we called the magnetic field B but in the code, we call it `H`), which takes as inputs pointers to the device memory field and current variables. As we will discuss below, we replace the single serial function by two kernel functions so that we do not need to store the updated E and B fields into new variables.

The first difference between the parallel and serial functions is that the kernel uses the grid-stride indexing. Since the variables are two-dimensional arrays, corresponding to the field values on the spatial grid, we call the kernels with `<<<grid2D, block2D>>>`, where these are set by `grid2D(GS)` and `block2D(GS)`. Inside the kernel functions, we utilize linear indexing to access the arrays, which are two-dimensional in physical space but of course one-dimensional in memory space.

The second difference is that we split the serial function into two kernel functions. To see why we did this, note that in the update equations for the Yee scheme, (2.3.1a)-(2.3.1c), the update of the magnetic field depends on the new values of the electric field. However, it depends on various nodal values of the electric field, which when run in parallel, may be updated on separate GPU threads. Thus, if we try to naïvely replace the serial function with one kernel which updates both the electric and the magnetic field, we run into a race condition where the electric field values may or may not be updated by the time the magnetic field needs to be updated. One solution to this is to store the updated field variables into new arrays, but for the sake of memory, this solution is not optimal. Instead, we utilize two kernels to first update the electric field and then the magnetic field, which eliminates the race condition as well as not needing extra memory to store new field variables.

Parallel energy calculation. In the parallel code, the serial function `void Energy` is replaced by the kernel `__global__ void Energy_Reduc`, which takes as inputs pointers to the device memory particle and field variables (as well as pointers to the device memory double variables to hold the field energy and particle energy).

The major difference in parallelizing the energy calculation is that since we want to sum all of the particle energies and all of the nodal field values, we have to implement a reduction in order to obtain the global sums. That is, since threads running in parallel cannot access a single global summation variable without introducing a race condition, reduction is required. Recall that the particle energy is given by

$$K = \Delta x \Delta y \left[\frac{1}{2} m \sum_{\alpha} [(v_x^{\alpha})^2 + (v_y^{\alpha})^2] \right],$$

whereas the field energy is given by

$$\mathcal{E} = \Delta x \Delta y \left[\frac{1}{2} \sum_{\text{nodes of } E_x} E_x(\text{node})^2 + \frac{1}{2} \sum_{\text{nodes of } E_y} E_y(\text{node})^2 + \frac{1}{2} \sum_{\text{nodes of } B} B(\text{node})^2 \right].$$

Note that there is a single sum for the particle energy since v_x and v_y have the same array dimension, while there are three sums for the field energy since E_x, E_y, B all have different array dimensions. Each thread traverses a stride depending on the block size, and sums to temporary variables Q_i^{temp}, where Q can be $\frac{1}{2}(v_x^2 + v_y^2)$, $\frac{1}{2}E_x^2$, $\frac{1}{2}E_y^2$, or $\frac{1}{2}B^2$. Once the individual energies of each thread are determined, the global energy of each Q is determined as $\sum_i Q_i^{\text{temp}}$.

An array is created for both K and $\mathcal{E}$ with length equal to the number of threads T in the block. The sum is computed in a converging binary tree style, with each element summing with the element $T/2n$ away from it in the array. The step n increases until the result is finally stored in the 0^{th} element of the array. This is sequential addressing parallel reduction. The kernel is called with `<<<1,BS2>>>` where `BS2` is the blocksize for the parallel reduction kernel. For a thorough discussion of parallel reduction in CUDA, see [2].

The main time loop. With all of the serial functions in the main time loop replaced by their corresponding kernels, we now discuss the structure of the main time loop and, in particular, optimizing the memory usage on the devices. The basic structure of the time loop is as follows:

```
for (double t =0; t < TMAX; t += dt)
{
     Particle_Push<<<grid1D, block1D>>>(...);
     ...
     Deposit_Current<<<grid1D, block1D>>>(...);
     ...
     Yee_E_Field<<<grid2D, block2D>>>(...);
     Yee_H_Field<<<grid2D, block2D>>>(...);
     ...
     Energy_Reduc<<<1,BS2>>>(...);
     ...
}
```

In order to reduce the amount of on-device memory at the calls of each kernel, we notice that each kernel does not need all of the device variables. Namely, the particle push needs the particle and field variables, the current deposition needs the particle and current variables, the field update needs the field and current variables, and the energy reduction needs the field and particle variables. Consequently, inside the time loop, we only allocate device memory when needed in the corresponding kernel, and free it otherwise (using the host memory to store updated values in the interim). This is summarized in Figure 8.

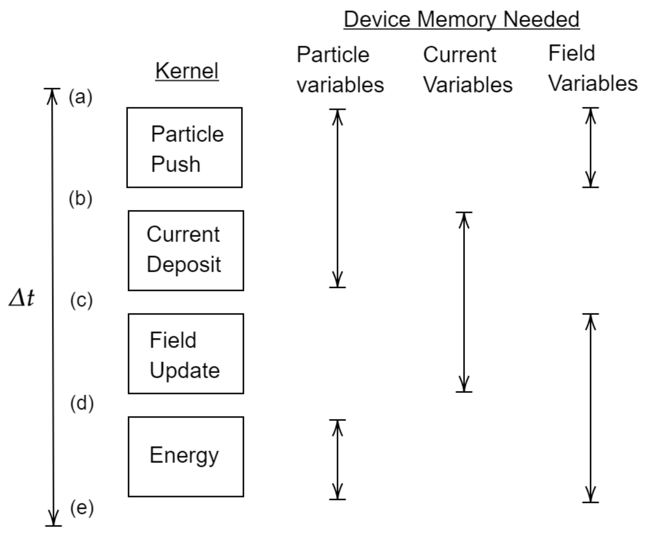

FIGURE 8. Device memory needed for the calls of the kernel functions inside the time loop

Thus, at (a) in Figure 8, we start with device memory allocated for the particle and field variables, holding the currently updated values. At (b), we can free the field variables from device memory (copying them to host memory), while allocating device memory for the current variables. At (c), we can free the particle variables (copying them to host memory) and allocate device memory for the field variables (and copying the host memory back to the device). At (d), we can free the current variables (we do not need to copy these to host memory, since current deposition always starts with $J_x = 0 = J_y$) and allocate device memory for the particle variables (and copying the host memory back to the device). The loop ends at (e) and starts again at (a), corresponding to the next timestep. By doing this, we observe that the on-device memory only ever needs two sets of variables, storing particle, current, and field variables, but not all three simultaneously.

This will ultimately allow us to test large scales in both the spatial grid and number of particles. While the number of particles N is generally much greater than the number of spatial steps, $N >> N_x, N_y$, note that the current and field arrays require memory which scales like $\mathcal{O}(N_xN_y) = \mathcal{O}(N_x^2)$ (for two spatial dimensions) as do the number of computations in the field update. On the other hand, the particle arrays require memory which scales like $\mathcal{O}(N)$, as do the number of computations in the particle push and current deposition. Hence, we decided to parallelize all of the kernels, since, for example, if $N = 10^6$ and $N_x = N_y = 10^3$, then the currents, fields, and particles all require roughly the same memory and the corresponding kernels require roughly the same number of computations. For a one-dimensional PIC simulation, it would suffice to parallelize over the particle pushing and current deposition, since the field update requires computations of order $\mathcal{O}(N_x)$, which is much less than the $\mathcal{O}(N)$ needed for the particle pushing and current deposition.

In Section 3.2, we will look at the scaling test for the serial code versus the parallel code with various block sizes, for various problem scales. We will compare the runtimes of the above four main processes. To do this, we use the timing functions from `time.h` to measure the runtimes for each each functional block of the main time loop (the total time for each functional block is summed over all timesteps; that is, we determine the total particle push, current deposit, field update, and energy calculation runtimes over all time steps). For the parallel code, we use `cudaDeviceSynchronize()` to make sure that all GPU devices are finished computing and synchronized before calling the command `clock()` on the host CPU to measure the time.

3. Results

In this section, we discuss the results from our experiments. In Section 3.1, we perform tests to demonstrate the correct physical behavior of our code. Subsequently, in Section 3.2, we perform scaling tests of the serial code `twod_pic.c` versus the parallel code `twod_pic_cuda.cu`.

3.1. **Weibel Instability.**

The Weibel instability is a physical process in plasmas for which an initial distribution of particles having an anisotropy in velocity space leads to the rapid generation of an electromagnetic field (for more details, see e.g. [5]). This instability damps the particle energy, which is transferred to the field energy; the instability continues until nonlinearities in the system damp the energy transfer between the particles and fields. To test the physical behavior of our code, we utilize the Weibel instability as a test, noting that the observed decrease in the particle energy should be compensated by an increase in the field energy. We demonstrate this, as well as examining the total energy (particle plus field energy), which should not drift far from the initial energy of the system.

To demonstrate the Weibel instability, we take our initial distribution to be

$$f_0(x, y, v_x, v_y) = (1 + \cos(\pi x))(y - y^2) \exp(-20v_x^2 - 25v_y^2),$$

which is an anisotropic Gaussian in velocity space. The particle mass and charge are set to 1. For the plots below, we took the number of spatial nodes (minus one) to be $N_x = N_y = 400$ (so $\Delta x = \Delta y = 1/N_x$), the number of particles $N = 5000$. We took $\Delta t = \Delta x/50$ to satisfy the CFL condition. We ran the simulation until final time $T = 10$, since at around $t = 5$, the energy transfer between the fields and particles slowed down. The plots were produced using the parallel code `twod_pic_cuda.cu`, where we added a printout inside the main time loop to print out the variables plotted every 100 time steps. The energy plots for the run are shown in Figure 9.

We observe the correct physical behavior: an electromagnetic field is generated which causes the particle energy to decrease; furthermore, the total energy drift is significantly less than the energy transferred between the particles and fields (the energy drift is roughly 5% of the energy transferred, so that the we attribute the energy transfer to the physical process, and not to numerical energy drift).

In the Appendix, we additionally include several snapshots of the evolution of the system. Qualitatively, time $t = 0$ of Figure 14 shows that the distribution function is correctly sampled in space and time $t = 0$ of Figure 17 shows that the distribution function is correctly sampled in velocities. Figures 14 and 15 show the generation of an electromagnetic field, as expected in the Weibel instability. Furthermore, Figure 16 shows that the current deposition is correct, with currents localized near the particle location (the snap shots in this figure are zoomed in for clarity).

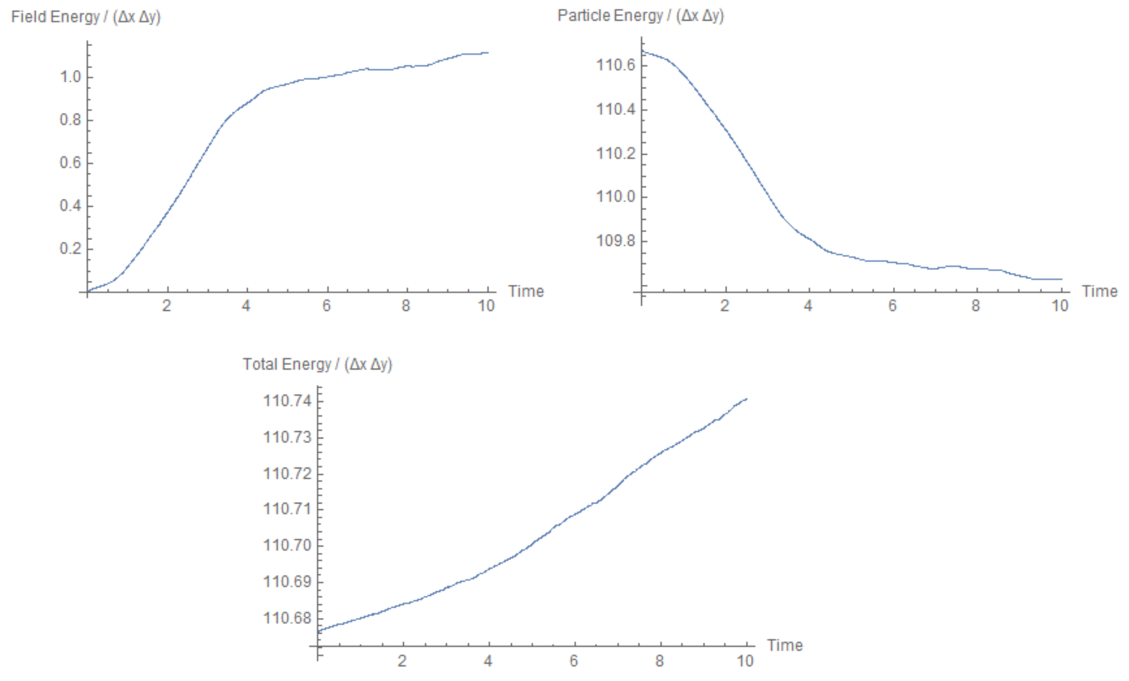

FIGURE 9. Energy versus time plots for testing the Weibel Instability (Top Left: $\mathcal{E}/(\Delta x \Delta y)$, Top Right $K/(\Delta x \Delta y)$, Bottom Center $(K+\mathcal{E})/(\Delta x \Delta y)$)

3.2. **Scaling Test.** Having verified the correct physical behavior of our code, we now turn to performing a scaling test, to compare the performance of the serial versus parallelized codes. As mentioned in Section 2.5, we determine the total time (over all time steps) for the particle push, current deposition, field update, and energy calculation.

Let S denote a scale parameter. For the scaling test, we take $N_x = N_y = 250 \times S$ ($\Delta x = \Delta y = 1/N_x$), the number of particles $N = 3125 \times S$, and the time step $\Delta t = \Delta x/2$ running to a final time $T = 1.0$ (thus, since the number of time steps is $N_t = 1/\Delta t$, the number of time steps is also $\mathcal{O}(S)$). For the runs, we took $S = 1, 2, 3, 4, 5$ to test the problem at various scales.

We ran the scaling test on Expanse, where the serial code was run on CPU and the parallel code was run on GPU.

For the serial code, we loaded the modules `slurm, cpu, gcc` and compiled the serial code `twod_pic.c` using

```
gcc twod_pic.c -lm -o twod_pic
```

and subsequently submitted the runs using the sbatch file `twod_serial_slurm.sb`.

For the parallel code, we loaded the modules `slurm, gpu, cuda` and compiled the parallel code `twod_pic_cuda.cu` on an interactive GPU node using

```
nvcc twod_pic_cuda.cu -o twod_pic_cuda
```

and subsequently submitted the runs using the sbatch file `twod_parallel_slurm.sb`. For the scaling test, we ran the parallel code over several block sizes, `BS = 4,8,16,32`, and for the energy calculation, we used several reduction block sizes `BS2 = 32,64,128,256`. The results of the scaling test are shown in the plots below (Figures 10, 11, 12, 13).

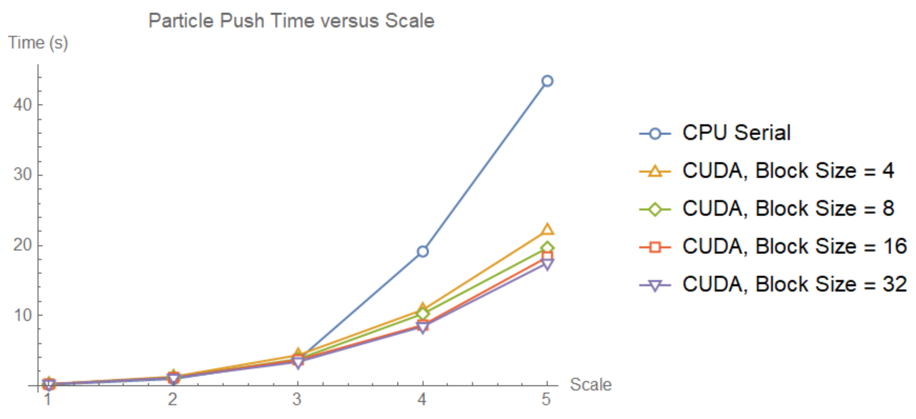

FIGURE 10. Scaling test: Total runtime for particle pushing

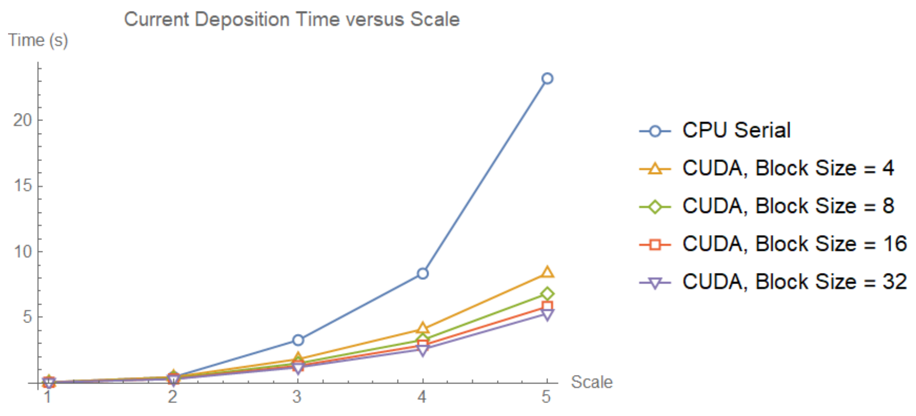

FIGURE 11. Scaling test: Total runtime for current deposition

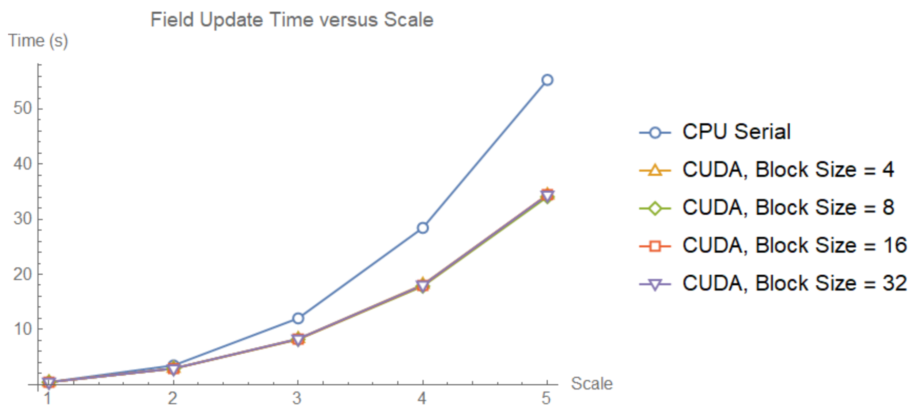

FIGURE 12. Scaling test: Total runtime for field update

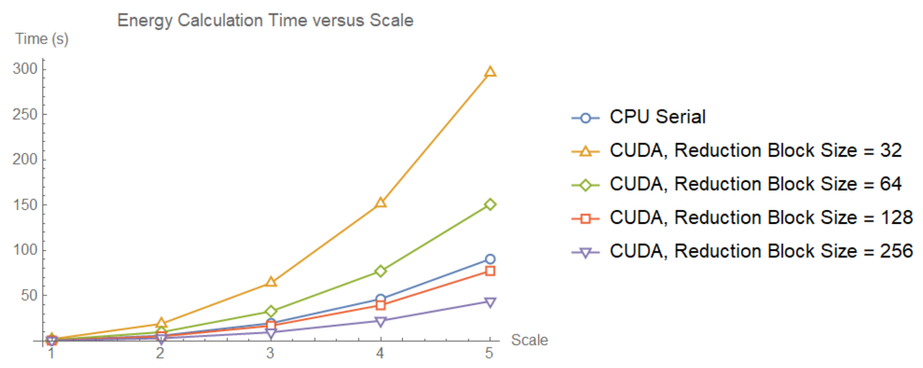

FIGURE 13. Scaling test: Total runtime for energy calculation

At the smaller scales, $S = 1, 2$, we see practically no speedup in the runtime of the various processes; of course, this is expected, due to overhead in transferring memory between host and device (which was included in our time measurements for the various processes). At the largest scale $S = 5$ of the test, for the particle pushing, we see an approximately 50% runtime reduction for the serial versus parallel with block size 4, up to an approximately 62% runtime reduction for the serial versus parallel with block size 32. Similarly, for the current deposition at the largest scale, we see runtime reductions ranging from 65% up to 78% across block sizes versus the serial runtime. For the field update at the largest scale, there is approximately a 40% runtime reduction (for all block sizes) compared to the serial runtime; there is not significant improvement in changing the block sizes here, which we believe is due to the fact that field update at each time step requires $\mathcal{O}(S^2)$ calculations (whereas the particle push and current deposition at each time step requires $\mathcal{O}(S)$); the effects of increasing the block size should become more evident at larger scales. For the energy calculation, we only see runtime reduction versus the serial code when the block size is sufficiently large (the runtime for reduction block size 64 was slower than the serial runtime but for reduction block size 128 was faster than the serial runtime); this is due to the necessity of syncing the threads when performing a reduction, so there is only advantage in the reduction when the reduction block size is sufficiently large. At reduction block size 256, there is approximately a 50% reduction in the runtime for the energy calculation versus the serial runtime.

The significance of these runtime reductions are given by the fact that the runtime of particle push and current deposition is quadratic with the scale $\mathcal{O}(S^2)$ (one has to perform $N \sim \mathcal{O}(S)$ operations over $N_t \sim \mathcal{O}(S)$ time steps to reach a fixed final time) and the runtime of the field update is cubic with the scale $\mathcal{O}(S^3)$ (one has to perform $N_x N_y \sim \mathcal{O}(S^2)$ operations over $N_t \sim \mathcal{O}(S)$ time steps). Similarly, the runtime of the energy calculation is cubic (it contains a quadratic runtime contribution from the particle energy sum and a cubic runtime contribution from the field energy sum; the cubic contribution dominates as S becomes large). Thus, we have improvements in two regards: the coefficients of the quadratic and cubic runtimes are reduced in the parallel algorithm (for a fixed block size). For example, if one does a quadratic fit of the particle push serial runtime and particle push parallel runtime (with blocksize 32), as functions of the scale parameter S, one sees a 60% reduction in the quadratic coefficient. Furthermore, if we increase the block size as we increase the scale of the problem, one can get subquadratic runtime for the particle push and current deposition, and subcubic runtime for the field update and energy calculation. For the interest of physical simulations where the scales becomes significantly large, this leads to dramatic runtime reduction.

4. Conclusion

In this project, we investigated the parallelization of a two-dimensional particle-in-cell algorithm. By studying the memory structure relevant to the algorithm (as discussed in Section 2.5), we found that we were able to parallelize all of the necessary functional processes (particle push, current deposition, field update, and energy calculation) while minimizing the necessary device memory. This allows our algorithm to be applicable to large scales where memory becomes a bottleneck. After discussing the implementation of the serial and parallel code, we performed a test in Section 3.1 to demonstrate the physical accuracy of our algorithm, and subsequently we performed a scaling test in Section 2.5 to demonstrate the runtime improvement of our parallel algorithm versus the serial algorithm.

References

[1] J. M. Dawson. Particle Simulation of Plasmas. *Rev. Mod. Phys.*, 55(403), 1983.
[2] M. Harris. Optimizing Parallel Reduction in CUDA. NVIDIA CUDA, 2011.
[3] J. Jin. Theory and Computation of Electromagnetic Fields. *Wiley*, 2010.
[4] M. Kraus, K. Kormann, P. Morrison, E. Sonnendrücker. GEMPIC: Geometric ElectroMagnetic Particle-In-Cell Methods. *Journal of Plasma Physics*, 83(4), 2017.
[5] L. Spitzer, Jr. Physics of Fully Ionized Gases. *Wiley*, 1962.
[6] The codes described in this report were implemented in C and CUDA C.
[7] The figures in this document were produced using the online graphics editor Mathcha 2021 (mathcha.io).
[8] The plots in Section 3 were produced using Mathematica Version 11.0, Wolfram Research, Inc., Champaign, IL (2017).
[9] The plots in the Appendix were produced using Python 3 (https://www.python.org/)

Appendix

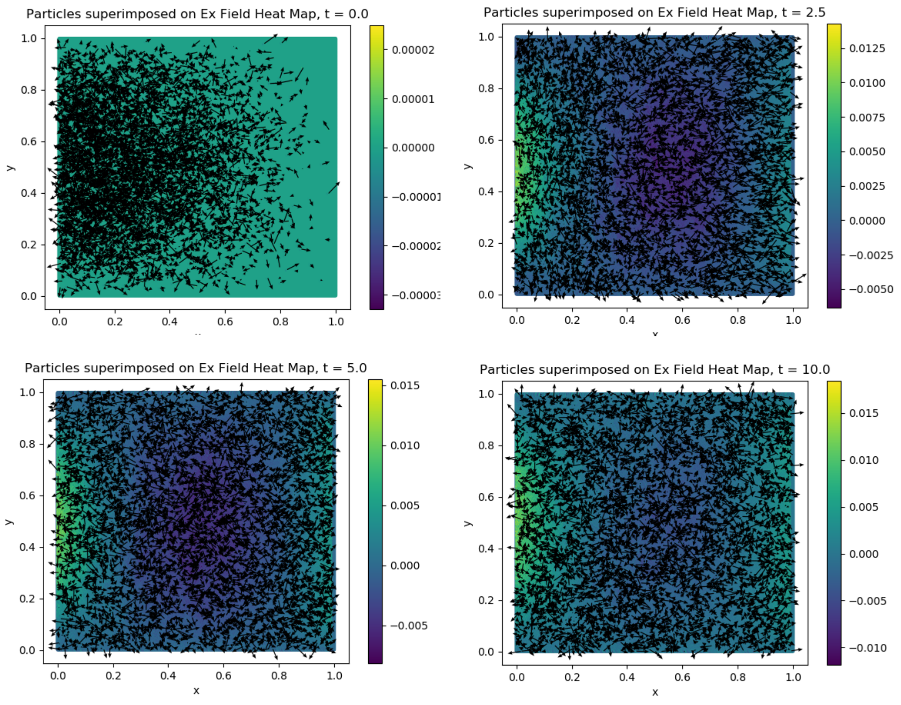

Figure 14. Snapshots of the evolution of E_x; superimposed with the particle quivers

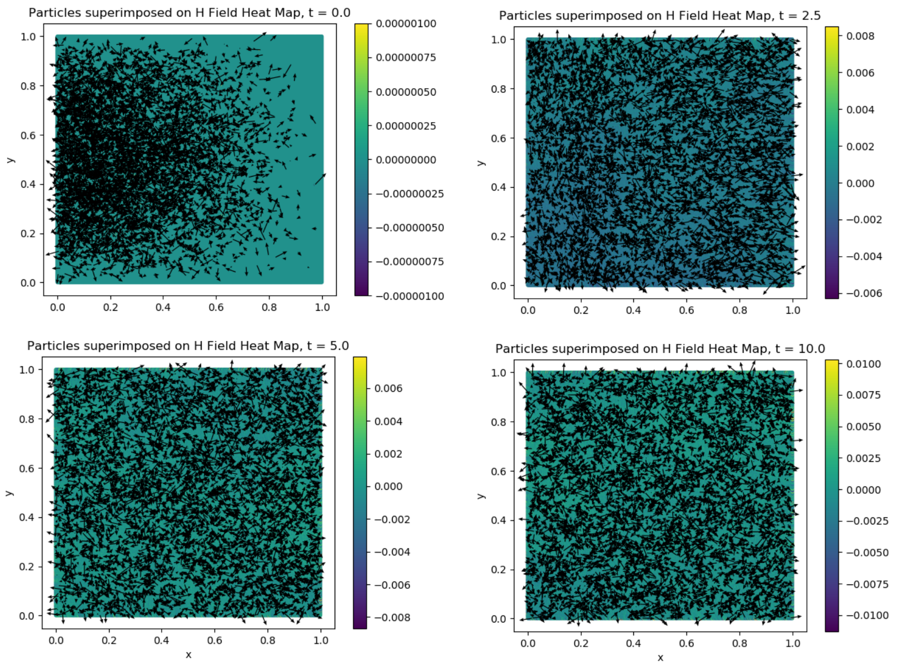

Figure 15. Snapshots of the evolution of H; superimposed with the particle quivers

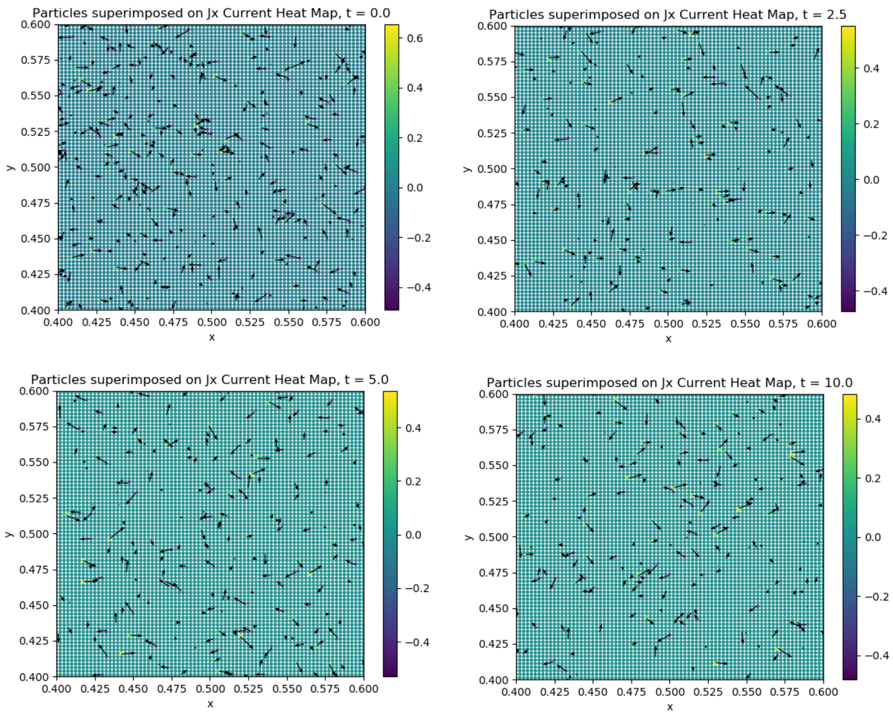

FIGURE 16. Snapshots of the evolution of J_x; superimposed with the particle quivers (zoomed in for clarity)

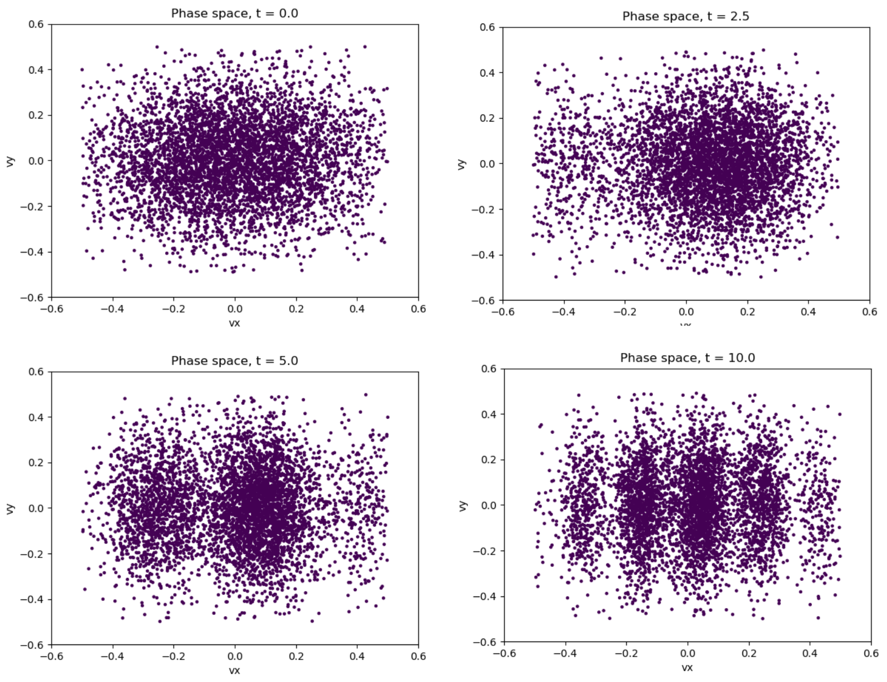

FIGURE 17. Snapshots of the evolution of velocity distribution