A Multi-GPU Python Solver for Low-Temperature Non-Equilibrium Plasmas

James Almgren-Bell, Nader Al Awar, Dilip S Geethakrishnan, Milos Gligoric, George Biros
{jalmgrenbell, nader.alawar, dilip.sg, gligoric}@utexas.edu
gbiros@acm.org
The University of Texas at Austin
Austin, Texas, USA

***Abstract*—The collisional Boltzmann kinetic equations for low-temperature plasmas find important applications in industry, for example semiconductor processing. Particle-in-cell (PIC) methods are the state-of-the art solvers for such problems, but they can be quite expensive. We present GPU acceleration of PIC codes and ways to increase programming productivity for rapid prototyping and algorithmic exploration. First, we present algorithms that minimize data movement and take advantage of modern GPU architectures. Second, we discuss their HPC implementation using Python-based productivity tools: CuPy, Numba, and PyKokkos. We analyze their performance, interoperability, portability, and overheads. We present performance analysis, comparing different algorithms for the main computational kernels. On a single GPU we observe 1.4 ns/particle/time step. We also report scaling results on up to 16 NVIDIA Volta V100 GPUs using MPI.**

***Index Terms*—Boltzmann, plasma, glow discharge, Kokkos, GPU, Numba, Python, HPC, particle-in-cell methods**

I. Introduction

Particle-in-cell (PIC) solvers combine grid-based methods with Lagrangian particle tracking methods. We focus on PIC solvers for industrial, non-equilibrium, ***low-temperature plasmas (LTPs)*** [1]. LTPs find applications in semiconductor processing, advanced manufacturing, and materials design.

LTPs are modeled by set of time-dependent partial differential equations (PDEs). These include the formidable *electron Boltzmann transport equation*: a 6D integro-differential equation for the electron velocity distribution function f as function of space, velocity, and time [2]. Characterizing f for different plasma conditions is required for calculating parameters like rate and transport coefficients, and enabling downstream tasks like uncertainty quantification and experimental diagnostics.

LTP calculations are expensive. Scales in LTPs range from microns for the electron mean free path, to the centimeter-scale of the device; and from picosecond time scales to resolve electron collisions to the microseconds required to obtain steady state. An additional challenge is that the ***collision kernel comprises elastic, ionization, excitation, and recombination*** terms each presenting different computational challenges, e.g., creation and destruction of particles, and two- and three-body interactions. The creation and destruction through collisions and boundary conditions introduces irregular data access patterns and concurrent data structure updates. Accurately calculating rate coefficients may require 1000s of particles per cell and the total number of particles can be in the billions.

Contributions. Using a standard PIC discretization (section II), we design and analyze computational kernels for particle computations and their coupling to a grid solver (section IV). Moreover, we examine the feasibility of undertaking the implementation using Python-based rapid prototyping and performance portable techniques using Numba [3], CuPy [4], and PyKokkos [5] (section III). PyKokkos is a Python framework that implements the Kokkos programming model for performance-portable shared memory parallel programming [6]. We share a number of lessons and insights for Python-based scalable HPC algorithm design on heterogeneous architectures that would be applicable to other HPC applications (section VII). We introduce improvements to PyKokkos, mainly coupling to CuPy arrays and removing overheads for fine-grain computational kernels, and we provide a detailed discussion on Python overheads (section V-B). From an algorithmic perspective, we estimate that our scheme provides a 5× improvement over the state-of-the-art on a single GPU [7] (sections V and VIII).

II. Mathematical formulation and discretization

Examples of LTP gases include argon, helium, air, carbon oxides, combustion gases, and others [1], [8]. The basic mechanisms for these gases are multiple elastic, ionization, excitation, and recombination collisions coupled to species transport and possibly reacting compressible fluid flow. These plasmas follow similar mathematical structure described here. To make ideas concrete and present our numerical experiments, we consider an argon LTP.

An argon LTP state is described by the following: (i) the ***"heavies"***, comprising the ***neutral argon*** $n_n(x,t)$; (ii) the ***ionized argon*** $n_i(x,t)$, and ***metastable argon*** $n_m(x,t)$ (particles per unit volume), which are evolved using transport continuum PDEs; (iii) the ***free electrons***, which are treated kinetically with $f(x,v,t)$ being the ***electron distribution function*** (in m^{-6}/s); (iv) and $\phi(x,t)$ (in volts), the instantaneous ***electric potential*** driven by a radiofrequency imposed potential and charge imbalances in the plasma. Let Ω represent the physical domain, so that $x \in \Omega$ and $v \in \mathbb{R}^3$. Then, the evolution of the state variables n_n, n_i, n_m, ϕ, f is given by

$$\frac{\partial f}{\partial t} + v \cdot \nabla_x f + q\nabla\phi \cdot \nabla_v f_e = \sum_{c_k k} \mathcal{C}_{ck}[f], \tag{1a}$$

$$-\Delta\phi = z(n_i - n_e[f]), \tag{1b}$$

$$\frac{\partial n_k}{\partial t} - \nabla \cdot (\mu_k n_k \nabla\phi + D_k \nabla n_k) = \sum_{c} \mathcal{C}_{kc}[f]. \tag{1c}$$

TABLE I *Collisions.*

C1:	$\mathrm{Ar} + e \to \mathrm{Ar} + e$	elastic electron-neutrals
C2:	$\mathrm{Ar} + e \to \mathrm{Ar}_+ + 2e$	ionization
C3:	$\mathrm{Ar} + e \to \mathrm{Ar}_m + e$	metastable excitation
C4:	$\mathrm{Ar}_m + e \to \mathrm{Ar}_+ + 2e$	2-step ionization
C5:	$\mathrm{Ar}_+ + 2e \to \mathrm{Ar}_m + e$	recombination

TABLE II *PIC Scheme. Index c indicates collision type; index k species; l particle; and j spatial cell.*

S1a:	$\mathcal{F}^c = \{x_l^c, v_l^c, w_l^c\} = \sum_{ck} \mathcal{C}_{ck}[\mathcal{F}^0]$	DSMC-Coll
S1b:	$E_l^c = -q\nabla\phi(x_l^c),\ \forall i$	DSMC-C2P
S1c:	$\mathcal{F}^+ = \{x_l^+ = x_l^c + \delta v_l^c;\ v_l^c + \delta E_l^c\}_l$	DSMC-Push
S2a:	$\forall j,\ n_e^+(\omega_j) = \sum_{x_l^+ \in \omega_j} w_l$	P2C
S2b:	$\forall j, k, c$, compute $\mathcal{C}_{kc}^+[\mathcal{F}^c](\omega_j)$	P2C
S3a:	$\forall k,\ n_k^+ =$ Solve eq. (1c) w/ ϕ^0 and $\mathcal{C}_k^+$	PDE
S3b:	$\phi^+ =$ Solve eq. (1b) w/ n_i^+, n_e^+	PDE

Here $k \in \{i, m, n\}$ for ion (Ar_+), metastable (Ar_m), and neutral argon atoms; q, z are constants related to vacuum properties, electron mass, and charge; D_k and μ_k are the corresponding diffusion and mobility constants for each species; $n_e[f] = \int_v f$ is the number of electrons per unit volume. $\mathcal{C}_{ck}$ represents the c_{th} collision operator [9] between electrons and species k and depends on the ***collision cross-section*** $\sigma_{ck}(\|v\|_2)$. Equation (1) is furnished with boundary and initial conditions (section V). Depending on the physical conditions, LTPs models include different collisions. Our plasma model includes five collisions, which we summarize in Table I.

These collisions represent multispecies elastic and inelastic collisions found in most LTPs. C2 and C4 "create" electrons by knocking them off neutral argon atoms; C5 removes free electrons by recombining them with Ar_+ to form Ar. The second electron in C5 acts as a "catalyte" in this reaction, and therefore is a three-body collision, which is more computationally expensive and involves special algorithms that do not seem to be discussed in the literature for GPU implementations.

A. Discretization

Following state-of-the-art hybrid PIC-formulations [10], we discretize eq. (1a)–eq. (1c) as follows. We assume that Ω is meshed using a regular grid with M ***"cells"*** $\{\omega_j\}_{j=1}^M$; eq. (1b) and eq. (1c) can be discretized on the regular grid using any standard method, we use finite-differences. We discretize eq. (1a) using a ***Direct-Simulation-Monte-Carlo scheme (DSMC)***: a Lagrangian scheme that tracks the motion of $N(t)$ particles with positions x_l, velocities v_l, and weights w_l. These particles represent a $N(t)$-point sample from $f(x, v, t)$: we define $\mathcal{F}(t) = \{x_l, v_l, w_l\}_{l=1}^N$ and thus, $\int_{\omega_j}\int_v f(x, v, t)g(x, v) \approx \sum_{x_l \in \omega_j} w_l g(x_l, v_l), \forall \omega_j \subset \Omega$. $N(t)$ varies in time due to collisions and boundary conditions. Particle position and velocity updates correspond to advecting f in the left-hand-side of eq. (1a); the Monte-Carlo step approximates the right-hand side of eq. (1a). Then, given the ***current state*** n_k^0, $\mathcal{F}^0$, ϕ^0, and time step δ, we advance to the ***new state*** n_k^+, $\mathcal{F}^+$, ϕ^+ using the operator-split scheme summarized in table II. Here ϕ and n_k are M-dimensional grid functions and F are particles sampled from f. DSMC steps 1 include collision; cell-to-particle (C2P) interpolation of $\nabla\phi$ to a particle; and advection ("Push"). S2a and S2b are ***particle-to-cell operations (P2C)*** for the right-hand side terms of eq. (1c). The steps are detailed in section IV.

We briefly discuss the cost of the scheme and revisit it in section V-A. Ignoring particle creation and destruction (section IV-B), steps S1 have $\mathcal{O}(N)$ work and $\mathcal{O}(1)$ depth (for regular grids). Steps S2 require sorting particles to cells followed by segmented reductions, thus, requiring $\mathcal{O}(N + M)(\log N)$ work and $\mathcal{O}(\log N + \log M)$ depth [11]. An alternative implementation uses atomic operations for the reductions while avoiding sorting. We discuss it in section IV-D. steps S3 are PDE solves that require $\mathcal{O}(M \log M)$ work and $\mathcal{O}(\log M)$ depth, assuming that optimal algorithms like multigrid are used for eq. (1b). *Fast solvers for steps S3 are not in the scope of the paper as they have been studied extensively in the literature* [10]. These solvers are crucial in 3D3V, but in 1D3V and 2D3V, the main cost is the Boltzmann solver [1].

Here we use explicit second-order finite differences with upwinding for the transport equations; a standard-second order stencil; a sparse Cholesky solver for the Poisson problem; and focus on the 1D3V case for the PDEs. But in the DSMC and P2C parts, we assume 3D3V [12] so that our algorithms correspond to that of 6D simulations with regular 3D grids. For irregular 3D grids, the only change is that sorting particles to cells requires a binary search.

Summary. Using a well-known PIC scheme, we focus on steps 1a–1c (Boltzmann kernel) and steps S2a and S2b for the P2C calculations. Optimizing these steps is of extreme importance as they constitute the bulk of the computations in most LTP simulations. We detail these kernels in section IV.

III. HPC Productivity in Python

In recent years, Python has seen increasing use in the field of scientific computing [13]–[15]. In this section, we discuss the Python based HPC libraries and frameworks that enabled us to write parallel, high-performance code on CPUs and GPUs.

Accelerated libraries. Python's rise to prominence has been mostly enabled by accelerated libraries and frameworks such as NumPy [16], CuPy [4], and SciPy [17]. These libraries provide high-performance data structures and kernels that are typically implemented in low-level languages such as C, and are then exposed to Python via language bindings.

Numba. While the aforementioned libraries provide a wide variety of high-performance kernels, they are limited when it comes to implementing ***custom*** kernels for different devices. Numba [3] is a just-in-time compiler for Python that can improve performance by compiling Python functions into machine code. It memoizes the compiled functions so that they only need to be compiled once per run. Numba can target a number of devices, including CPUs and NVIDIA GPUs (the AMD GPU backend is currently not maintened). We use Numba to implement parallel CPU and GPU kernels.

PyKokkos. Unlike Numba, PyKokkos provides Python abstractions for ***performance-portable*** shared memory parallel programming of custom kernels. These abstractions allow us to run the same code on multiple devices with minimal to

TABLE III *Comparing the Frameworks (High, Medium, and Low).*

Framework	Usability	Performance Portability	Overhead
NumPy/CuPy/SciPy	H	M	M
Numba	H	M	H
PyKokkos	H	H	H
CUDA + pybind11	L	L	L

no code changes and no losses in performance. PyKokkos then translates kernels written by the user into Kokkos [6] and C++, using pybind11 for interoperability between the two languages. Similar to Numba, PyKokkos memoizes the compiled functions to avoid the cost of re-compiling the kernel after the first call. ***PyKokkos through Kokkos uses OpenMP for CPUs and CUDA for NVIDIA GPUs***. Support for other devices, such as AMD GPUs through HIP or SYCL, is planned [18]. Due to the performance portability abstractions, we do not need to change our code in the future to run it on these devices. We implemented our GPU kernels using PyKokkos: for every Numba GPU kernel, we also have a PyKokkos implementation.

CUDA. Another option for writing kernels is to implement them directly in CUDA, and then use language bindings to call them from Python using pybind11, PyCUDA [19], and other techniques. We used this approach for the segmented prefix reduction in our P2C kernel (table II).

Table III shows a high level comparison between the different approaches to writing high-performance Python code. ***Usability*** refers to general ease-of-use of the framework: CUDA + pybind11 suffers from poor usability as it requires additional effort on the user's part to write the bindings, while the other approaches provide Python interfaces which are intuitive to Python programmers. ***Performance portability*** refers to the portability of the code across different devices (e.g., CPUs and GPUs) without loss of performance. PyKokkos excels here as it is built on top of Kokkos, which was designed specifically for this purpose, while other approaches either support only one type of device (i.e., CUDA) or require rewriting the code to target specific devices (i.e., Numba). Finally, ***overhead*** refers to the performance overhead of each approach, meaning how much of the execution time is spent outside of the GPU kernel. We expect the overhead to be minimal for CUDA + pybind11 compared to the other frameworks, which execute Python code internally before calling the kernel. We measure and report these overheads for our kernels in section V-B.

IV. Particle kernels

In this section, we describe the data structures we used (section IV-A), details regarding algorithms, implementation, and performance for the DSMC collision kernel (section IV-B), the recombination collision C5 in table I, (section IV-C), and the particle-to-cell kernels (section IV-D). We remark that although C5 is mathematically part of the DSMC step in table II, we discuss it separately in the remainder of the paper.

A. Data Structures

We store particle positions and velocities together in a two-dimensional N-by-6 CuPy array for all particles. CuPy arrays are stored row major, improving locality during the collision calculations. Other values required in the particle to cell kernel—weights, source term contributions, and energies—are stored together in a separate two-dimensional N-by-4 CuPy array. This separation allows our particle to cell kernel (section IV-D) to run over contiguous memory.

TABLE IV *DSMC Collision Step for a particle l.*

Draw six random numbers R_l	CS0
Access particle data x_l, v_l, w_l	CS1
$\forall k, c$ cross-sections $\sigma_{ck}(\|\|v_l\|\|_2)$	CS2
$\forall k, c$ probabilities $\pi_{lck}(\sigma_{ck})$	CS3
Collision: update x_l, v_l	CS4
Collision: if applicable, create particle $l' : x_{l'}, v_{l'}, w_{l'}$	CS5
Advection (Push): update x_l, v_l	CS6
Boundary conditions: if applicable, set $w_l = 0$	CS7

During the simulation, the number of particles typically grows due to ionization until it reaches steady state. We therefore preallocate additional memory during the setup phase for all data structures. The size of this memory can be estimated using the plasma parameters, e.g., by the expected ionization and recombination rates; and can be adaptively adjusted every 1000s of time steps if necessary.

B. DSMC Collision Kernel

We now discuss the details of the DSMC collision kernel, which performs steps S1a-c (table II) of the PIC scheme. This step is not embarrassingly parallel due to the recombination collision C5, which we discuss separately in section IV-C), and due to particle creation (e.g., ionization) or loss (e.g., absorption by electrodes or dielectric walls, or recombination).

Algorithm. Table IV details the steps. In the ***DSMC-Coll step*** for each particle l and each collision c with species n_k, we compute collision probabilities $\pi_{lkc} = 1 - \exp(-\delta\|v_l\|_2\sigma_{ck}(\|v_l\|_2)n_k)$. Using π_{lck}, we decide whether and how to collide particle l, using (six in our case) random numbers. For each timestep, we precompute these random numbers for all particles using `cupy.random.rand()` before the kernel invocation (see section VII). If the particle collides, we compute the post-collision velocities using conservation of mass and energy [9]. Last, advection in physical and velocity space is applied and boundary conditions are enforced. Referring to table II, notice that before CS6 we need the ***C2P operation*** to compute $\nabla\phi(x_l)$. Depending on the scheme this can be an expensive operation. Most PIC-DSMC codes are low-order accurate and use constant interpolation; we do the same. We precompute a constant $\nabla\phi$ per cell, and then retrieve it using the particle's cell id.

Accuracy and numerical stability dictate overall collision frequency to be low, say less than 10%. As a result, the kernel spends most of its time in determining which collision (if any) a particle will undergo (steps CS3-4 in table IV), rather than performing collision calculations (step CS6). To determine π_{lck}, all particles need to compute $\sigma_{ck}(\|v_l\|_2)$, a function of the particle's speed; typically, this is done via lookup tables [8]. Instead, we predetermine a piecewise polynomial interpolation of each cross section. We thus reduce calculation time while keeping cross section error under 2%.

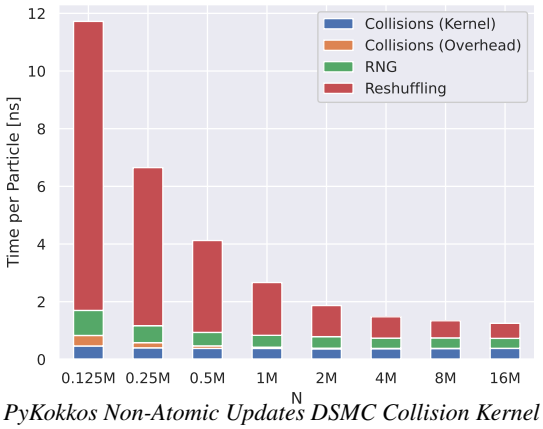


Fig. 1 *PyKokkos Non-Atomic Updates DSMC Collision Kernel Breakdown.*

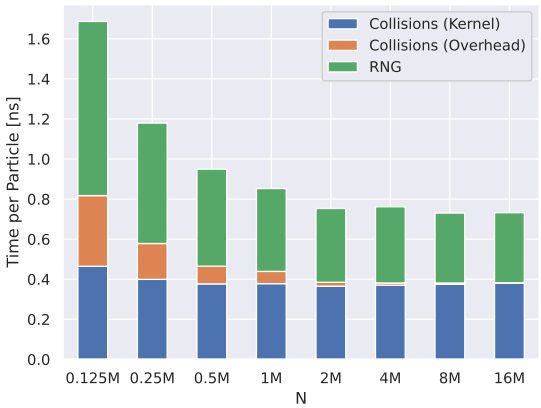


Fig. 2 *PyKokkos Atomic Updates DSMC Collision Kernel Breakdown.*

A potential efficiency pitfall is thread divergence due to particles undergoing different collisions. Due to low collision rates, most particles undergo no collisions within each time step. Implementations designed to minimize thread divergence proved to be more costly due to the extra calculations required, which outweighed the minimal thread divergence.

As mentioned, a challenge in implementing the collision kernel is the varying number of particles in each time step. Before executing the collision kernel, it is unknown how this number will change due to creation or destruction of particles. We discuss two algorithms for ***data structure updates*** due to particle gain/loss: an atomic-operations-based version and a non-atomic-operation version, the latter of which is easier to analyze and it only requires concurrent load operations.

Non-Atomic Updates. Given $N(t)$ particles at time step t, we store new particles in an auxiliary buffer of size $N(t)$. In the event particle l ionizes a neutral and creates particle l', it writes the data for l' into the buffer at $l + N$. Once all particles are done, a separate GPU kernel is needed ***to reshuffle*** particles so that they are stored contiguously. This requires a parallel select to find which particles were created and destroyed and then rearrange them in a contiguous buffer of size $N(t + 1)$. Reshuffling must be performed at every time step to allow storing new particles in the auxiliary buffer using the $l + N$ rule. The theoretical complexity of a parallel select has linear work and logarithmic depth [11]. However, it requires significant data movement. Despite its elegance, this scheme is slow because it does not exploit the hardware-supported atomics available in modern GPU architectures.

Atomic Updates. Instead of using a size N buffer with unique write indices, we maintain a single write index I shared by all particles. When a particle is created, each thread increments I atomically and writes the new particle into the array at that index; thus, a full size N buffer is not required. While the atomic operation adds some computational cost, low ionization rates mean few atomic operations are performed, keeping this extra cost minimal.

Performance. We implemented the collision steps as Numba and PyKokkos kernels. Figures 1 and 2 show the breakdown of the execution time of the particle steps for the PyKokkos implementations of both the non-atomic and atomic versions of the DSMC collision kernel respectively. We split the runtime into kernel time (measured with the NVIDIA profiler `nvprof`) and overhead time (wall clock time minus kernel time). We also show the RNG time and the mandatory reshuffling step that follows the kernel for the non-atomic version. Figure 1 shows that reshuffling at every time step dominates the execution time, particularly for smaller problem sizes. In contrast, the cost of using atomics within the collision kernel is almost nonexistent (fig. 2). Therefore, we adopt the atomics-based collision version.

We also note that implementing the atomics-based version allows for a simple extension to using operator split methods, where multiple collision kernels are performed for each time step of the continuum physics due to different time step constraints. The use of atomics removes the need for a synchronization step after each call to the collision kernel, greatly improving the efficiency with which operator split schemes can be implemented. The use of operator splitting is particularly beneficial when considering communication costs in multi-GPU implementations, as demonstrated in fig. 3.

TABLE V *Recombination Steps. RS0 is precomputed; RS1 is a separate GPU kernel; RS2–RS5 are in the same GPU kernel.*

CS3 (in table IV) computes $\mathcal{P}$	RS0
$\forall$ cell ω_j, construct $\mathcal{P}_j^c$	RS1
$\forall$ particle $l \in \mathcal{P}$:	
Read $\omega_j(l)$	RS2
Atomically pop $l' \in \mathcal{P}_j^c(\omega_j(l))$	RS3
Update catalyte velocity $v_{l'}$	RS4
Destroy primary: $w_l = 0$	RS5

C. Kernel for Recombination Collision C5

Collision C5 in table I is a three body interaction between Ar_+ and two electrons, ***the primary and the catalyte***. This collision requires global synchronization across GPU-blocks to set up primary-catalyte pairs.

Algorithm. C5 is summarized in table V. The catalyte particle must be in the same cell ω_j as its primary. Each particle in ω_j is exclusively designated either as catalyte or primary. Furthermore, primaries must be ***uniquely*** matched to catalytes. We do this as follows. We define $\mathcal{P}$ to be the list of all primaries, precomputed in step CS3 (table IV) of the

collision kernel. For each primary l we also have precomputed its cell $\omega_j(l)$. For each cell ω_j, we define $\mathcal{P}_j^c$ to be the ***set of all possible catalyte particles*** l' ***in*** ω_j such that $l' \notin \mathcal{P}$. Then for a particle $l \in \mathcal{P}$ we execute steps RS2–RS5 in table V.

Implementation. We now consider steps RS1 and RS3, the two major steps of this kernel. One potential implementation of these steps is locally in space; if particle data is sorted by position, these steps can be done by mapping of primaries to non-primaries within a given cell. Since our particle data is unsorted, we instead incorporate atomics in the same manner as the collision kernel. Step RS1 atomically increments a counter for each cell ω_j, such that potential catalyte partners in cell ω_j are written into $\mathcal{P}_j^c[I_j]$. Step RS2 is a load operation, as the ω_j is precomputed. Similar to step RS1, step RS3 reads values from $\mathcal{P}_j^c$ using atomically updated indices to ensure that no two primaries in cell ω_j select the same catalyte. Steps RS4 and RS5 can be done with minimal computation. We group the steps such that steps RS1 and steps RS2–RS5 are implemented as two separate kernels in both Numba and PyKokkos.

Performance. In the physical regime of the glow discharge problem we consider, recombination collisions are rare; thus, the cost for the collision steps RS2-RS5 is low. The majority of time spent in this kernel is therefore in step RS1, the assembly of the catalyte list, which is done for all grid cells. We report detailed timings for this kernel in section V, where we see that the performance of the kernel depends on the number of cells and the number of particles per cell. There we report aggregate timings for steps RS1–RS5.

D. Particle-to-Cell

The particle-to-cell kernel performs steps 2a-b in table II. It computes per-cell right-hand sides for the PDEs in eq. (1), as well as quantities of interest, for example the electron temperature $T_e(\omega_x) = 1/|\omega_x| \int_{\omega_x} \mathbb{E}_f[\|v\|^2]$ [2]. For our solver, we compute four variables, which are used for the calculation of $n_e(x)$, $n_i(x)$, $n_m(x)$, and $T_e(x)$. For each particle l, we define $\mathbf{V}_l$ to be the vector of these variables. We seek to calculate the sum of these values $\mathbf{V}^j$ corresponding to particles in each grid cell ω_j. That is, $\mathbf{V}^j = \sum_{x_l \in \omega_j} \mathbf{V}_l$. Therefore, our problem becomes a block reduction, where the prefix of a particle is the spatial cell in which it lies. We consider two algorithms for this kernel.

Algorithm 1. We first sort the particles to per cell arrays, and then we use per cell parallel reductions. This is a scheme that is implemented using only concurrent load operations.

Algorithm 2. Alternatively, we can use atomic reductions for each V^j. This scheme avoids the need for particle sorting. If the number of particles per cell is very large it pays off to introduce further partitioning of ω_j to auxiliary subdomains ω_{jm} to atomic updates congestion. Then, with each particle being assigned to a GPU thread, we find $\omega_{jm}(x_l)$, and then atomically update V^{jm}. Upon completion a number of threads are assigned to ω_j for the reduction $V^j = \sum_m V^{jm}$.

Implementation. We implemented Algorithm 1 using ThrustRTC, a Python wrapper to the Thrust library, which itself is written in CUDA for use in C++. We used `Sort_By_Key()` for sorting and `Reduce_By_Key()` for reductions. Thrust does not support vector prefix sums on the CuPy data structures we employ, therefore we require four kernel calls for V.

We implemented Algorithm 2 using both CUDA and PyKokkos. For N from $100k$ to $16M$, both CUDA and PyKokkos implementations are $10\times$–$20\times$ faster than the Thrust-based implementation of Algorithm 1.

E. Multi-GPU extension

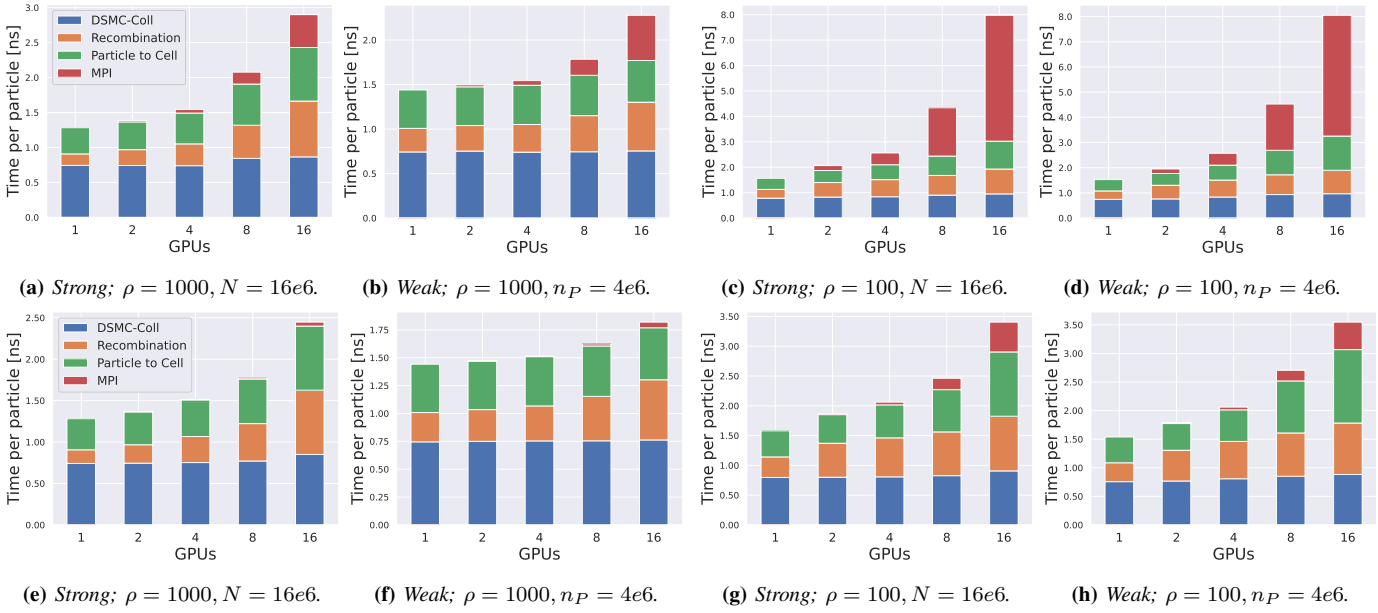
To give a flavor of the relative costs of the solver components in the context of large-scale applications, we implemented a straightforward MPI extension: the PDE-calculations (steps S3a and S3b in table II) are ***replicated*** on each MPI task. For 1D3V and 2D3V simulations this replication is not prohibitively expensive. In the scalability results in the next section, we show results with up to 160K cells, which is a typical resolution for 2D3V calculations [7]. Our scheme could be extended successfully to 3D3V calculations in which Ω is coarsely partitioned in large domains, and each large domain has its own MPI communicator with PDE-grid replication. ***Our scheme is suboptimal compared to domain-decomposing both*** Ω ***and particles, but this is beyond the scope of this paper.***

Algorithmically, our MPI extension is a single `MPI_AllReduce()`. Specifically, let P be the number of ***MPI tasks***; each task owns N/P electrons and replicates all $\{\omega_j\}_{j=1}^M$ cells. Steps S1a–1b in table II are done in an embarrassingly parallel manner at each task. Steps S2a and S2b is where the communication takes place: a local P2C operation (section IV-D), to compute a per task $\mathcal{C}_k^+, n_k^+$ followed by an all-reduce with message size $\mathcal{O}(M)$ and complexity $\mathcal{O}(M \log P)$ [20]. Finally steps S3 are replicated at each MPI task. Since $M = N/\rho$ the $\mathcal{O}(N/P)$ calculation dominates when $\rho \gg P$. This resolves load balancing issues since, although we destroy and create particles, all tasks are statistically equivalent as they view the entire domain Ω. The overall memory costs are $\mathcal{O}(N/P + M)$, where the prefactor is 25 and accounts for positions velocities, cross-sections, time-stepping and particle-to-cell interactions.

V. Overall Scalability Results

Setup. All runs are in double precision. We used "Lassen", a system at Lawrence Livermore National Laboratory. Each node is equipped with an IBM Power9 CPU, 256GB of RAM, and four NVIDIA V100 Volta GPUs with 16 GB of RAM each. The V100 peak double precision performance is 7.8 TFLOPS and its memory bandwidth is 900 GB/sec. Lassen's infiniband interconnect's bandwidth is 12.5 GB/sec. For the strong and weak scaling experiments, we use one GPU per MPI task, with up to four MPI tasks per node. We use Python 3.8.8, CUDA 10.2, GCC 8.3.1, mpi4py 3.1.3, Numba 0.54.1, the "develop" branch of the PyKokkos GitHub repository, and the IBM Spectrum MPI. We performed several runs of each experiment and confirmed that the results are stable.

Normalized time results. In fig. 3 we report timings as a function of the number of GPUs for differ problem sizes. To present and discuss the results in fig. 3, we introduce the following variables. Let N be the number of particles (electrons). Let P be the number of GPUs or equivalently MPI tasks as we use one GPU per MPI task. We define the



(a) *Strong;* $\rho = 1000, N = 16e6$. **(b)** *Weak;* $\rho = 1000, n_P = 4e6$. **(c)** *Strong;* $\rho = 100, N = 16e6$. **(d)** *Weak;* $\rho = 100, n_P = 4e6$.

(e) *Strong;* $\rho = 1000, N = 16e6$. **(f)** *Weak;* $\rho = 1000, n_P = 4e6$. **(g)** *Strong;* $\rho = 100, N = 16e6$. **(h)** *Weak;* $\rho = 100, n_P = 4e6$.

Fig. 3 *Normalized time in nanoseconds for strong and weak scaling runs using the PyKokkos kernels. In particular, we report* $\hat{T}_P = T_P/n_P = T_P P/N$*, where* T_P *is the wall-clock time per simulation time step (averaged over 400 time steps);* N *is the total number of particles;* P *is the number of GPUs which is equal to the number of MPI tasks, and* $n_P = N/P$ *is the number of particles per GPU. It follows that the parallel efficiency* $\eta_P = \hat{T}_1/\hat{T}_P$*, thus, the closer the plots are to a straight line the better the scaling. (In the weak scaling runs* n_P *is fixed to* n *for all* P*.) Also,* ρ *is the number of particles per cell. "DSMC-Coll" refers to all the kernels in fig. 2; "Recombination" refers to C5 in table I; and "MPI" refers to the overall MPI communication costs. In the first row, we report timings for one DSMC step per PDE solve; in the second row, we report timings for 10 DSMC steps per PDE solve, as it is typically done in subcycled, multirate time-marching schemes in production runs. For each run the total unnormalized time* $T_P = \hat{T}_P n_P = \hat{T}_P N/P$*. For example, in fig. 3a,* $T_8 = 4.1$ *milliseconds and* $T_{16} = 2.8$ *milliseconds.*

grain size to be $n_P = N/P$ the number of particles per GPU. We report *the normalized time* on P GPUs defined by $\hat{T}_P = T_P/n_P = T_P P/N$. Notice that for an embarrassingly parallel algorithm that has complexity $T_P = \mathcal{O}(N/P)$, we have $\hat{T}_P = \mathcal{O}(\hat{T}_1)$ so that all weak and strong scaling runs should be straight lines; however, our algorithm has overheads due to the MPI communication and the fact that the grid cells are replicated at each MPI task. We give details and a performance analysis further below.

In fig. 3 we report normalized times for the DSMC-Coll step, the recombination collision, the P2C step, and the MPI communication step. All other costs are negligible. The first row results were obtained with runs that use one DSMC-Coll and recombination step per PDE solve. The second row results were obtained with runs that used a subcycled multirate scheme that takes ten DSMC-Coll/recombination steps per PDE solve [9], [21]. Therefore for the second row runs, we have one P2C and `MPI_AllReduce()` every ten DSMC-Coll/recombination kernel calls. This explains the reduced costs for those two kernels.

In figs. 3a and 3b we report strong and weak scaling results for $\rho = 1000$; and figs. 3c and 3d show results for $\rho = 100$. These timings depend on the number of cells (elements of finite-difference grid points). Similar definitions are used in figs. 3e to 3h. Recall that in fig. 2, we presented the breakdown for the stand-alone DSMC-Coll kernel (table II) without the recombination kernel. *The total time in fig. 2 corresponds to the DSMC-Coll time in fig. 3.* For all strong scaling runs in figs. 3a and 3c, we use $N = 16e6$ particles; thus $M = 160$K for $\rho = 100$ and $M = 1600$ for $\rho = 1000$. For all weak scaling runs we use $n = 4e6$.

The parallel speedup on P GPUs is defined as $s_P = T_1/T_P = \hat{T}_1 P/\hat{T}_P$. The parallel efficiency is defined as $\eta_P = s_P/P = \hat{T}_1/\hat{T}_P$. For example, in fig. 3e, $\hat{T}_1 = 1.29$ ns, $\hat{T}_8 = 1.77$ ns, and $\hat{T}_{16} = 2.43$ ns. Thus, $\eta_8 \approx 73\%$ and $\eta_{16} \approx 53\%$. In fig. 3c, $\hat{T}_1 = 1.59$ ns, $\hat{T}_8 = 4.33$ ns, and $\hat{T}_{16} = 7.97$ ns. Thus, $\eta_8 \approx 37\%$ and $\eta_{16} \approx 20\%$. What causes this efficiency drop especially in the small ρ case? Next we derive performance models of the four different kernels in fig. 3, which we then use to interpret the observed efficiencies. As we will see, the main reason for the efficiency drop is the cell replication across MPI tasks.

A. Performance Analysis

To better interpret the results, we first introduce performance models for steps reported in fig. 3. We define T_{DSMC}, T_{MPI}, and T_{P2C}, the wall-clock time for the DSMC-Coll kernel, the MPI communication, and the particle to cell kernel respectively. We define the normalize times $\hat{T}_{\mathsf{DSMC}} = T_{\mathsf{DSMC}}/n_P$, $\hat{T}_{\mathsf{MPI}} = T_{\mathsf{MPI}}/n_P$, and $\hat{T}_{\mathsf{P2C}} = T_{\mathsf{P2C}}/n_P$. The normalized times are the ones shown in fig. 3. We omit discussion of the recombination kernel as the analysis is nearly identical to the P2C one.

The MPI communication kernel. Since $\hat{T}_{\mathsf{MPI}}$ is the most prominent one, especially in the first row for $\rho = 100$, we start with it. T_{MPI} is the cost of an all-reduce operation, that should scale as $\tau_c \log_2(P) M$, with τ_c being the inverse of internode bandwidth and assuming a hypercube topology with negligible latency costs [20]. It follows that $\hat{T}_{\mathsf{MPI}} = \tau_c \log_2(P) P/\rho$.

Therefore, the cost should decrease with increasing ρ as it should because the main work is done in the per-particle kernels where the MPI communication scales with the grid size. Also the normalized per particle time increases with P as the percentage of communication per particle increases. Thus, the formula for $\hat{T}_{\mathsf{MPI}}$ explains the results we observe. We benchmarked the `MPI_Allreduce()` on "Lassen", and we show a factor of two difference, which we attribute to the possibility that the default implementation uses a reduction and a broadcast for the message size we're using.

The DSMC-Coll kernel. The DSMC-Coll kernel corresponds to the whole kernel analyzed in fig. 2. Here we ignore the random generation as it is an external library call. For the analysis we assume an infinite cache system in which the main cost is the loads and stores of the necessary data, followed by floating-point calculations. We ignore all costs related to atomic operations. To present our analysis we define the following: τ_f is the ***machine time per double precision FLOP***, τ_m is the ***RAM access/byte***, $f_D = 405$ is the ***number of FLOPs/particle***, which includes advection as well as collisions C1–C4 and $m_D = 18 \times 8$ is the ***memory operations per particle***. This includes loads and stores for 3D positions, velocities, weights, and reads for random number and 3D electric field, precomputed in the C2P step. The arithmetic intensity is $f_D/m_D = 2.8$ and the machine imbalance is $\tau_m/\tau_f = 70$ for double precision calculations on the NVIDIA V100. Then, the ***time per particle*** is $T_{\mathsf{DSMC}} = f_D\tau_f + m_D\tau_D$, $\approx \tau_m m_D$. For $\tau_m = 1/900$ ns, the estimated cost per particle is 0.16 ns. In fig. 2, we observe roughly 0.42 ns. We attribute the discrepancy to register pressure; the compiler shows about 96 registers per thread, which leads to reduced occupancy. Restricting the number of registers helps, but the benefits are not as dramatic due to register spilling. There is also some sensitivity on the number of blocks and threads per block.

P2C kernel. Recall that in the P2C kernel we need to do four segmented reductions on particle properties, where the number of segments is equal to the number of field cells. In our implementation, the P2C kernel requires $\mathcal{O}(1)$ work per particle to find its cell and then $O(M)$ concurrent reductions of approximately M/n_P size each. The computation for the first part is negligible and we ignore it. The main computation cost is managing the $O(M)$ atomics-based reductions.

The memory bandwidth cost m is loading the per-particle properties and thus, $m = 4 \times 8$ ***double precision memory operations per particle***. In all, we obtain $\hat{T}_{\mathsf{P2C}} = m\tau_m + O(M/n_P) = m\tau_m + O(P/\rho)$. The $m\tau_m$ cost is 0.03 ns and thus negligible compared to the observed $\hat{T}_{\mathsf{P2C}}$. For example, in fig. 3a, we observe $\hat{T}_{\mathsf{P2C}} = 0.4$ ns for $P = 1$. Thus the dominant term is the atomic reductions cost $O(P/\rho)$. This analysis explains the slight increase of $\hat{T}_{\mathsf{P2C}}$ with increasing P, which is amplified with smaller ρ.

Summary. Within $O(1)$ factors we can explain the scaling and absolute numbers of our code. The degradation in scalability is due to the field mesh replication. However, our solution is simple and for many practical problems of interest the multi-GPU version offers significant speedups especially for large ρ and subcycled time marching schemes.

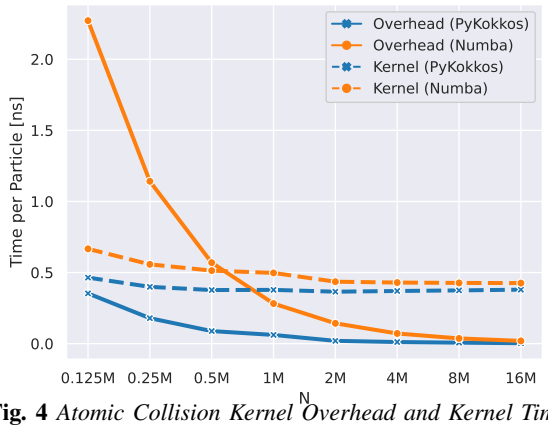


Fig. 4 *Atomic Collision Kernel Overhead and Kernel Time.*

B. Overheads

In this section, we look at the performance overheads introduced by Numba and PyKokkos and we present PyKokkos optimizations that enable its efficient use for light-weight kernel calls. The CuPy overheads are minimal, typically on the order of 30-40 μs per kernel call; however, some CuPy kernels are slow due to poor default options for kernel grid sizes (section VII). CuPy also allows easy wrapping of raw CUDA kernels; while slightly more convenient than using pybind11, this is not portable.

We calculate all overheads by subtracting the kernel time reported by nvprof (the NVIDIA CUDA kernel profiler) from the wall-clock time of each kernel call measured in Python. Note that we do not include timing data from the first call to each kernel, to avoid standard first invocation overheads.

Initially, PyKokkos did not interoperate properly with CuPy arrays, meaning that we had to pass data on the host and copy it before every kernel call. This introduced a lot of overhead, so we added a feature to PyKokkos to wrap existing CuPy arrays, avoiding the memory copies. Our initial experiments with custom GPU kernels written in Numba and PyKokkos showed that while the kernels themselves are quite fast, and are close to native CUDA kernels in terms of performance, both frameworks introduced significant overhead. For our smallest problem size, $N = 125$K, we observed that the overhead was at least double than the kernel itself. Since PyKokkos presents a more portable framework we decided to focus on PyKokkos: we profiled the framework's run-time code and identified two opportunities for optimization.

The first optimization we applied (CachingOpt) was caching the results of intermediate function calls internal to PyKokkos. An example of such an optimization would be caching the module import step. On every call to a kernel, PyKokkos has to import the module that holds it: this module is a shared object file stored on the file system. Our optimization fixes this by importing the module on the first call and maintaining a handle to it which can be reused on subsequent calls.

The second optimization we applied (HandleOpt) involves the mechanism by which PyKokkos retrieves the handle to the kernel and calls it. On every kernel call, PyKokkos will first lookup the handle to the kernel in the imported module and then process the user-passed arguments to pass them to the

TABLE VI *PyKokkos vs other Frameworks;* $N = 125\text{K}$.

	Time per call (ms)		
Kernel	Implementation	Kernel	Overhead
Collision	PyKokkos	0.08	0.05
Kernel	Numba	0.10	0.37
Recombination	PyKokkos	0.05	0.07
Kernel	Numba	0.04	0.49
Prefix	PyKokkos	0.06	0.30
Reduction	CUDA	0.03	0.07

kernel. This optimization caches the function handle and the processed arguments so subsequent calls can reuse them.

Following these optimizations, overhead per particle for our smallest problem size decreased from 2.48ns to 0.36ns, going from 84.3% of total time to just 43.8%.

Figure 4 shows the overhead and kernel times per particle of the atomic collision kernel using both Numba and PyKokkos (with our optimizations) for different problem sizes. We first note that the kernel execution times are similar. Looking at overheads, for all listed problem sizes, the PyKokkos overhead is lower than the both the corresponding Numba overhead and even the PyKokkos kernel time. In contrast, the Numba overhead greatly exceeds the kernel time for smaller problem sizes. In summary, PyKokkos with our optimizations has far less overhead than Numba, and this is especially notable for smaller problem sizes which is significant for strong scaling.

Table VI compares the PyKokkos overhead and kernel times per call of our GPU kernels to alternative approaches, namely Numba and CUDA + pybind11. Note that the overheads listed here are not divided by the number of particles and are constant for each kernel for different problem sizes.

As stated previously, PyKokkos overhead is much smaller than that of Numba, while the kernel times are similar. Comparing PyKokkos to CUDA, we see that PyKokkos has significantly higher overhead: this is expected, as each kernel call executes PyKokkos Python run-time code before invoking the kernel itself. Additionally, we can see that the CUDA kernel is faster than the PyKokkos implementation. Using the profiler, we found that the PyKokkos kernel ran with a much smaller block size, which is set automatically by Kokkos itself and cannot currently be controlled by the PyKokkos user (although this will be supported in the future). Meanwhile, we had to run a search over different block and grid sizes for our CUDA implementation to find an optimal configuration. Therefore, PyKokkos achieves better performance than Numba for the collision and recombination kernels and falls behind a native CUDA implementation for the prefix reduction kernel, but has superior usability and portability characteristics.

In summary, PyKokkos has much lower performance overhead than Numba for custom kernels and mostly similar kernel execution times. Compared to CUDA + pybind11, PyKokkos has additional overhead and slower kernel execution time for one of our kernels, although we expect future releases of PyKokkos to further improve both of these aspects.

VI. Simulation of a Glow Discharge Apparatus

As a demonstration, we present the simulation of a glow discharge apparatus in which a pulsating voltage is applied across the discharge tube and drives the formation of a low-temperature ionized plasma. Data from the simulation is shown in fig. 5. For the simulation discussed, we used the following parameters: • Electron, ion density $n_e(x,0) = n_i(x,0) = 5 \times 10^9 \frac{\#}{\text{cm}^3}$; • Electron temperature $T_e(x,0) = 5\text{eV}$; • Metastable density $n_m(x,0) = 0\frac{\#}{\text{cm}^3}$; • Neutral density $n_n(x) = 3.22 \times 10^{16}\frac{\#}{\text{cm}^3}$; • Driving voltage amplitude $V_0 = 75V$; • Radio frequency $\omega = 13\text{MHz}$; • Pressure $P = 1\text{Torr}$; • Electrode gap $L = 2.54\text{cm}$; • Initial number of particles $N_0 = 2 \times 10^6$; • Number of cells $M = 256$; and • Time step: $\delta = \frac{1}{8000\omega} \approx 1 \times 10^{-11}\text{s}$. The simulation, with time horizon 6 ms, took less than an hour on a single V100 GPU.

This glow discharge scenario generates a quasi-neutral plasma [22], [23], in which electron and ion density are approximately equal and thus there is zero net charge almost everywhere in the plasma. A notable physical phenomenon present in glow discharge problems can be seen near the edges of the domain, where electron density is very low; this region is known as the ***sheath***. Strong electric fields prevent most electrons from exiting the plasma. At the inner edge of the sheath, we observe an oscillatory behavior in the electron density function but not in the ion one. This well-studied behavior, seen in fig. 5a, is known as plasma oscillation. Data outputs of particular interest are species density functions and electron energy distribution functions, as seen in fig. 5.

TABLE VII *Wall Clock Times for CuPy Sums* (μs).

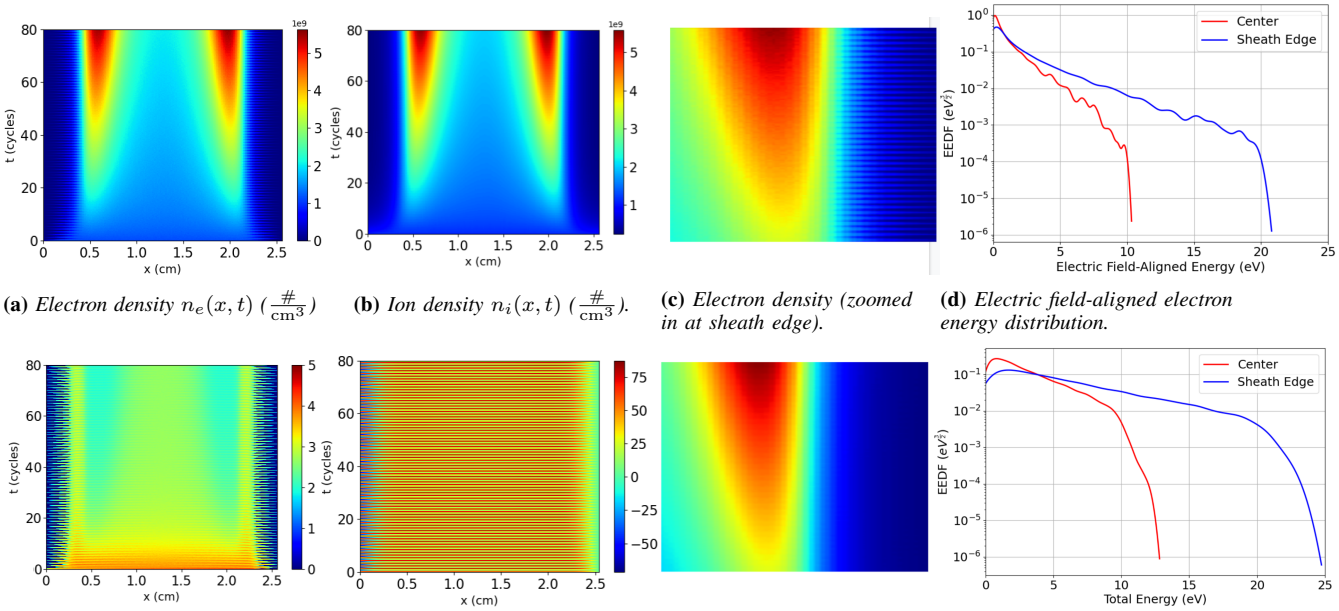
Problem Size	0.125M	1M	8M
1 Variable	58	68	140
4 Variables	405	2,770	21,983

VII. Performance Considerations

In this section, we discuss some lessons we found in Python for HPC, in particular, CuPy/Numba optimizations, atomic operations, and random number generation.

CuPy:. As part of the reshuffling step in section IV-B, our algorithm requires a vector reduction. Using the default implementation of `cupy.sum()` resulted in poor performance; upon further investigation, we found that the default setup for CuPy uses only the fixed values of 512 threads and 1 block (and these values cannot be configured), regardless of the size of the input array. As a result, it scales poorly for larger problem sizes. The CuPy developers suggested we set the `CUPY_ACCELERATORS` environment variable to `cub`, which switches CuPy to the CUB backend. While this improved scalability in a test problem, it did not help in our case since our data is not stored in a contiguous axis, as we setup our memory layout to store each particle's data contiguously in our CUDA reduction kernel. This caused CuPy to fall back to the default implementation. The discrepancy between single and multiple variable sums can be seen in table VII.

Numba:. Our initial CUDA kernel implementations using Numba showed higher overheads than what was reported in Section V-B. Further investigation showed that the main cause was that we were passing in CuPy arrays directly as arguments, which causes Numba to convert these arrays to it's own device array type at every kernel call. We fixed this by



(a) *Electron density* $n_e(x,t)$ *(*$\frac{\#}{\text{cm}^3}$*)*

(b) *Ion density* $n_i(x,t)$ *(*$\frac{\#}{\text{cm}^3}$*).*

(c) *Electron density (zoomed in at sheath edge).*

(d) *Electric field-aligned electron energy distribution.*

(e) *Electron temperature* $T_e(x,t)$ *(eV).*

(f) *Voltage* $V(x,t)$ *(V).*

(g) *Ion density (zoomed in at sheath edge).*

(h) *Total electron energy distribution.*

Fig. 5 *Glow discharge simulation for time horizon up to* $= 80$ *RF Cycles. We observe the sheath near the left and right electrodes, and plasma oscillations. In the last column, we plot the electron energy density distribution function (EEDF) at two spatial locations: at the center of the discharge (*$x = 1.25$*) and the sheath (*$x = 0.5$*) at the end of the simulation. Notice that at the sheath, where the temperature is higher, the EEDF has longer tails—as expected. Once the EEDF is obtained, quantities of interest like rate coefficients can be computed.*

defining a Numba device array from each CuPy array during the setup phase and then passing the Numba array to the kernel, eliminating the need for conversion at each time step. We also found that overhead can be further reduced by adding the kernel argument types to the function signature.

Atomics:. One kernels with atomic operations, in particular the prefix reduction, we observed poor scalability for high numbers of grid points. The workload size of the prefix reduction is the number of particles, so we expect the runtime to be solely a function of the number of particles. This is true for up to approximately 50k cells. However, when the number of cells grows large (50k and above), the runtime grows almost linearly with the number of cells.

Random Number Generation. We consider two methods of drawing these random numbers in two different ways: with CuPy and with Numba or PyKokkos. With CuPy, the random numbers are drawn before the collision kernel; with Numba, they are drawn on the fly. Predetermining the random number generation with CuPy provides significantly better performance than on the fly with Numba.

VIII. Related Work

Related work on PIC codes. There is a rich literature on HPC algorithms and software for PIC methods for collisionless plasma simulations. Examples include [10], [21], [24]. There is also work on Vlassov-Fokker-Plank collisional models of fusion plasmas [24], but LTPs operate in a different regime. Surprisingly, there is little work on HPC algorithms for LTPs. We only discuss methods that solve directly for f and not just upscaled PDE models for T_e and n_e. The main metric we use to compare the methods is time in [ns/particle/time step] for all non-PDE components; that is, the collision kernel, the particle-to-cell kernel, and any data structure modifications to account for the creation and destruction of particles. Most codes either use single precision or do not specify it.

In [25] the authors consider a simulation with M =10K. No detailed timings are reported. [26] achieves ***44 ns/particle per time step in single precision*** on one V100 NVIDIA GPU. It uses 1D-PIC calculations similar to ours, but the effect of the number of cells is not discussed ($\rho = 20$K); also, it requires replication of the whole grid on the GPU-block shared memory to improve particle-to-cell calculation and thus does not work for large M. [27] uses sorting to assign particles to cells at each time step with $M \approx 4$K and $\rho = 1000$. It does not support multi-GPUs and achieves about ***130 ns per particle***.

Using the normalized time per particle as a metric, we consider the calculations in [7] to be the *state-of-the-art*. This code uses per-time-step sorting of particles to cells and the null-step method for collisions. It supports neither multi-GPUs nor recombination collisions. The largest problem (with unspecified precision) had 20M particles on a single GTX Titan X GPU with memory bandwidth 336.5 GB/sec vs V100's 900 GB/sec. For non-uniform particle distributions [28] it delivers ***8 ns per particle/time step***.

Comparing codes is tricky due to differences in low-level implementation details, e.g., cross-sections, hardware, unspecified features like single-vs-double precision, the spatial distribution of particles and possible other details. With these caveats in mind, we compare [7] to the strong-scaling results in fig. 3a for one V100 GPU. We deliver 1.4 ns/particle/time step in double precision, with recombination collisions; this amounts to 5.17× improvement over [7]. With 16 GPUs the

total time is nT_{16} and after normalizing to the *total number* of particles N we obtain 0.2 ns/particle/time step.

Related work on poductivity environments. Kokkos [6] and RAJA [29] are C++ libraries that provide abstractions for performance portable shared memory parallelism; we use PyKokkos, which provides Kokkos-like abstractions in Python. In addition to PyKokkos and Numba, a number of HPC Python frameworks have been developed recently. DaCe [14] and Intrepydd [30] provide Python abstractions for writing portable high-performance code which is then compiled to C++, CUDA, or other device specific frameworks or languages. We chose Numba as it is older and more well-established, and PyKokkos as it emphasizes performance portability. PyOMP [31] extends Numba by adding OpenMP style pragmas for CPU parallelism. As most of our kernels are more suited for GPUs, we did not attempt to use CPU specific frameworks in our code. Legate [32] is a drop-in replacement for NumPy providing accelerated and distributed computing. Other Python task-based programming systems include Pygion [33], which provides a Python interface to Legion, and Dask [34]. We did not need tasking abstractions, but we were able to scale to multiple nodes using mpi4py.

IX. Conclusion

To our knowledge this is the first multi-GPU solver for LTPs and the first to report details for recombination collisions. Although we included only three species and five collisions, our algorithms apply to more species and collision types [22]. We presented a simple but effective MPI algorithm for up to 64M particles and 160K cells. Using Python and PyKokkos we were able to rapidly perform algorithmic exploration while ensuring code portability. We also shared a set of insights that will be helpful to other HPC researchers. We extended PyKokkos to improve its interoperability with CuPy and NumPy and reduce overheads for fine-grained kernels. Future work includes using particle reweighing to adaptively control the approximation of the velocity distribution function; extending the MPI implementation to partitioned Ω; and using multi-resolution methods to accelerate convergence to a steady state. We also look forward to testing performance-portability on the upcoming AMD and Intel GPU architectures.

References

[1] L. Alves *et al.*, "Foundations of modelling of nonequilibrium low-temperature plasmas," *Plasma Sources Science and Technology*, vol. 27, no. 2, 2018.

[2] C. Villani, "A review of mathematical topics in collisional kinetic theory," *Handbook of mathematical fluid dynamics*, 2002.

[3] S. K. Lam *et al.*, "Numba: A LLVM-based Python JIT compiler," in *Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015.

[4] R. Okuta *et al.*, "CuPy: A NumPy-compatible library for NVIDIA GPU calculations," in *Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[5] N. Al Awar *et al.*, "A performance portability framework for Python," in *International Conference on Supercomputing*, 2021.

[6] C. Trott *et al.*, "The Kokkos ecosystem: Comprehensive performance portability for high performance computing," *Computing in Science and Engineering*, vol. 23, no. 5, 2021.

[7] M. Y. Hur *et al.*, "Model description of a two-dimensional electrostatic particle-in-cell simulation parallelized with a graphics processing unit for plasma discharges," *Plasma Research Express*, vol. 1, no. 1, 2019.

[8] "Phelps and TRINITI databases," 2021, www.lxcat.net.

[9] V. Vahedi and M. Surendra, "A Monte Carlo collision model for the particle-in-cell method: applications to argon and oxygen discharges," *Computer Physics Communications*, vol. 87, no. 1-2, 1995.

[10] W. Zhang, A. Myers, K. Gott, A. Almgren, and J. Bell, "AMReX: Block-structured adaptive mesh refinement for multiphysics applications," *The International Journal of High Performance Computing Applications*, vol. 35, no. 6, 2021.

[11] G. E. Blelloch, "Programming parallel algorithms," *Communications of the ACM*, vol. 39, no. 3, 1996.

[12] "3D3V" refers to 3D in space and 3D in velocity. The majority of LTP codes are 1D3V or 2D3V, i.e., 1D in space 3D in velocity or 2D in space and 3D in velocity respectively.

[13] T. E. Oliphant, "Python for scientific computing," *Computing in Science and Engineering*, vol. 9, no. 3, 2007.

[14] A. N. Ziogas *et al.*, "Productivity, portability, performance: Data-centric Python," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[15] M. Bauer, *et al.*, "Code generation for massively parallel phase-field simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[16] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, 2020.

[17] P. Virtanen *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, 2020.

[18] A. S. Dufek *et al.*, "Case study of using kokkos and sycl as performance-portable frameworks for milc-dslash benchmark on nvidia, amd and intel gpus," in *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2021.

[19] A. Klöckner *et al.*, "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation," *Parallel Computing*, 2012.

[20] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *An Introduction to Parallel Computing: Design and Analysis of Algorithms*, 2nd ed. Addison Wesley, 2003.

[21] M. Bettencourt *et al.*, "EMPIRE-PIC: a performance portable unstructured particle-in-cell code," *Communications in Computational Physics*, vol. 30, no. SAND-2021-2806J, 2021.

[22] P. K. Panneer Chelvam and L. L. Raja, "Computational modeling of the effect of external electron injection into a direct-current microdischarge," *Journal of Applied Physics*, vol. 118, no. 24, 2015.

[23] Q. Liu *et al.*, "Numerical study of effect of secondary electron emission on discharge characteristics in low pressure capacitive RF argon discharge," *Physics of Plasmas*, vol. 21, no. 8, 2014.

[24] S. M. Mniszewski *et al.*, "Enabling particle applications for exascale computing platforms," *The International Journal of High Performance Computing Applications*, vol. 35, no. 6, 2021.

[25] C. H. Kim *et al.*, "Investigation of dual-frequency effect on the ion energy and flux in Torr-regime plasma by a two-dimensional GPU-PIC simulation," *Plasma Sources Science and Technology*, 2021.

[26] Z. Juhasz *et al.*, "Efficient GPU implementation of the particle-in-cell/Monte-Carlo collisions method for 1D simulation of low-pressure capacitively coupled plasmas," *Computer Physics Communications*, vol. 263, 2021.

[27] A. A. Romanenko *et al.*, "High performance collisional PIC plasma simulation with modern GPUs," *Journal of Physics: Conference Series*, vol. 1336, no. 1, nov 2019.

[28] Our 8 ns estimate is done using Fig 3 and algorithm C in [7]. The faster Algorithm D assumes a uniform distribution of particles, which for many LTP applications is not valid.

[29] D. A. Beckingsale *et al.*, "RAJA: Portable performance for large-scale scientific applications," in *Workshop on Performance, Portability and Productivity in HPC*, 2019.

[30] T. Zhou *et al.*, "Intrepydd," in *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2020.

[31] T. G. Mattson *et al.*, "PyOMP: Multithreaded parallel programming in Python," *Computing in Science and Engineering*, vol. 23, no. 06, 2021.

[32] M. Bauer and M. Garland, "Legate NumPy: Accelerated and distributed array computing," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[33] E. Slaughter and A. Aiken, "Pygion: Flexible, scalable task-based parallelism with Python," in *Parallel Applications Workshop, Alternatives to MPI*, 2019.

[34] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Python in Science Conference*, 2015.