GPU Accelerated Simplified Implicit Particle-in-Cell Simulation

Rishi Vijayvargiya[†]
KTH Royal Institute of Technology
Stockholm, Sweden
rishiv@kth.se

Max Harrison[†]
KTH Royal Institute of Technology
Stockholm, Sweden
maxharr@kth.se

Paul Mayer[†]
KTH Royal Institute of Technology
Stockholm, Sweden
pmayer@kth.se

I. Introduction

Plasma physics simulation are integral tools used from astrophysics, fusion energy research to weather prediction. The Particle-in-Cell method is a widely used simulation technique. It mainly consists of a three step computational loop.

1) **Particle Mover** (e.g. fluid particles): Movement of individual particles is integrated in continuous space.
2) **Interpolate P2G**: Respective particle charges are interpolated onto a grid mesh
3) **Field solver**: solve Maxwell's equations on the mesh to update the electromagnetic field.

The goal of this project was to accelerate a simplified Particle-in-Cell simulation [1], [2] through the use of GPUs. The PIC method is highly suited for GPU optimisation: it is highly parallel and is numerically intensive. Computing the values of particles and density information has a high arithmetic intensity - e.g. simulating a grid of 256×128 for 100 cycles took almost 5 minutes on our test system. A particle simulation tracks typically millions of particles, computing a trajectory for all of them independently. The embarrassingly parallel nature of the problem structure makes this a prime target for GPU optimisation. As all particle data is stored in continuous arrays, throughput can additionally benefit from coalesced memory access. We can thus demonstrate large performance improvements for the simulation in our test environment.

Crucially, the problem scope has been reduced for our case: we skip solving for the electromagnetic field (the third step of the computational loop) as specied by the project assignnment. We only focus on integrating the particles in space and interpolating the charges onto the grid points. This simplification not only reduces the overall code complexity of the simulation, but also allows for some aggressive optimisation. For example, we will exploit the fact that the field information doesn't change and therefore can remain in GPU memory for the whole simulation.

II. Methodology

We focused on 2 functions defined in the `src/Particles.cu` file: `move_PC` and `interpP2G`, where the former is responsible for the movement of the particles and the latter interpolates densities based on the new positions of the particles for each species. In the starter code, both these implementations used the CPU. Thus, our project consisted of the following:

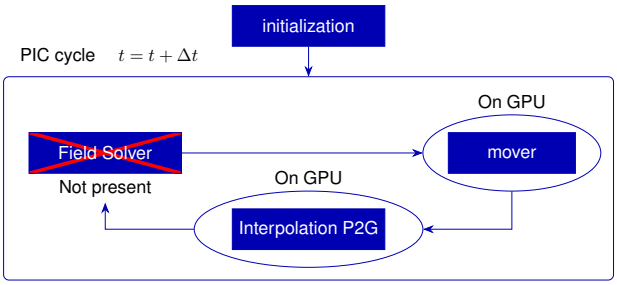


Fig. 1: Structure of Simplified PIC-Solver, [3]

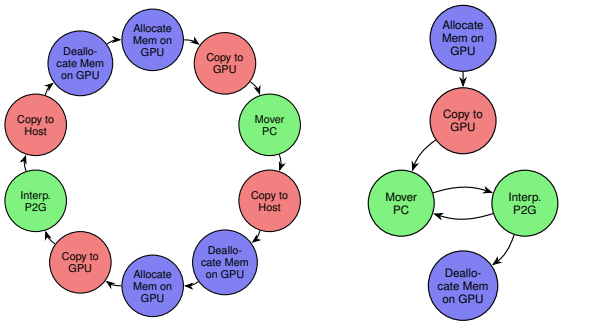
1) GPU Port: We first ported both of these functions to work on the GPU using CUDA programming (Topic 1 from Group A).
2) Profiling then Optimizing: After the initial, naïve port was completed on the GPU, we profiled the GPU version and developed several optimizations, with each change building on top of the other. (Topic 2 from Group A).

In this section, we first give an overview of our approach for the GPU port, and then outline successive optimisation steps yielding several different code versions. A comparison between the successively optimised versions is presented in Section III.

A. Naïve GPU Port

The main challenge in porting the existing code to run on the GPU is to allocate and copy all structs correctly. Because the data is stored in arrays within said `structs`, if simply copied to the GPU using *cudaMemcpy*, the copied `struct` would contain invalid pointers (pointing to CPU memory). This is solved by a two stage process. We first allocate the array on the GPU, and then allocate the `struct` by replacing the CPU pointers with the GPU pointers. Listing 1 demonstrates this for the particle data. This allows us to split up the `mover_PC` and `interpP2G` functions into two functions: `mover_PC_gpu` and `interpP2G_gpu`. The `mover_PC` function becomes `mover_PC_gpu`, which specifies the kernel information (Threads Per Block (TPB), Number of Blocks (NBlocks)), calls the GPU kernel and does

[†]Authors made equal contribution to the project



(a) First naïve port to GPU

(b) Moved allocation to initialization (consolidation)

Fig. 2: Computation graphs of simulation cycle

CUDA error handling. For TPB we chose 1024 and thus NBlocks is $\lceil \frac{\#particles}{1024} \rceil$. In the sequential version of the code, the computation is written as two nested for loops where the outer loop specifies subcycling and the inner loop iterates over each particle. The ordering of these loops are independent and each particle can be computed independently, so we can write the kernel by reordering the loops and replace the particle index with block and thread id.

We used a similar approach for the `interpP2G` method, where the `interpP2G_gpu` function sets up the kernel calls and the `interp_particle_gpu` kernel contains the computations. The CPU interpolation looped over all the particles, where each of the 10 density quantities were interpolated based on particle positions. The `interp_particle_gpu` kernel removes the outer loop over the particles, meaning that one GPU thread is responsible for interpolating the densities from one particle. Because of this parallelism, multiple threads might modify the same density quantity at once. Thus, an `atomicAdd` is used to ensure integrity of the computation while updating each density quantity. Finally, we also observed that in the CPU version there is each density quantity is interpolated using its own nested loops. Instead our kernel uses one nested loop and interpolates all density quantities at once, resulting in more readable code and a less complex kernel.

This implementation structure for `mover_PC_gpu` and `interpP2G_gpu` resulted in a structure depicted in figure 2a. We allocated memory, moved data to the GPU, and called the kernel, which was after computation was followed by moving the data back and deallocating memory.

B. Consolidation

The naïve implementation had the obvious problem of allocating and deallocating memory structures as well as copying data (like the grid) that is needed for both kernels (see Figure 2a) This is inefficient compared to first allocating memory on the device before the simulation cycles, copying the required data over to the GPU *once*, and then deallocating once the simulation is complete. Profiling our first implementation reveals that we spend up to 75% of our time allocating and moving data. Figure 3 clearly shows that this is the main bottleneck of the computation for the mover kernel and is a non-trivial factor in the runtime for the interpolation. To ”consolidate” these memory structures into permanent memory, we identified data that is used by both kernels and which does not change during the duration of the simulation. This is the case for the grid, parameters, and field data, which we allocate and copy once and then leave in the GPU memory for the whole computation. The life-cycle for these objects now looks like as depicted in figure 2b. We term this step *consolidation*.

Performing more runtime tests revealed that the performance gains from this step were negligible: particle information was the largest contribution to memory transfer overhead. To further reduce memory overhead, we overlap the memory movement with the actual computation by introducing streams.

C. Streaming

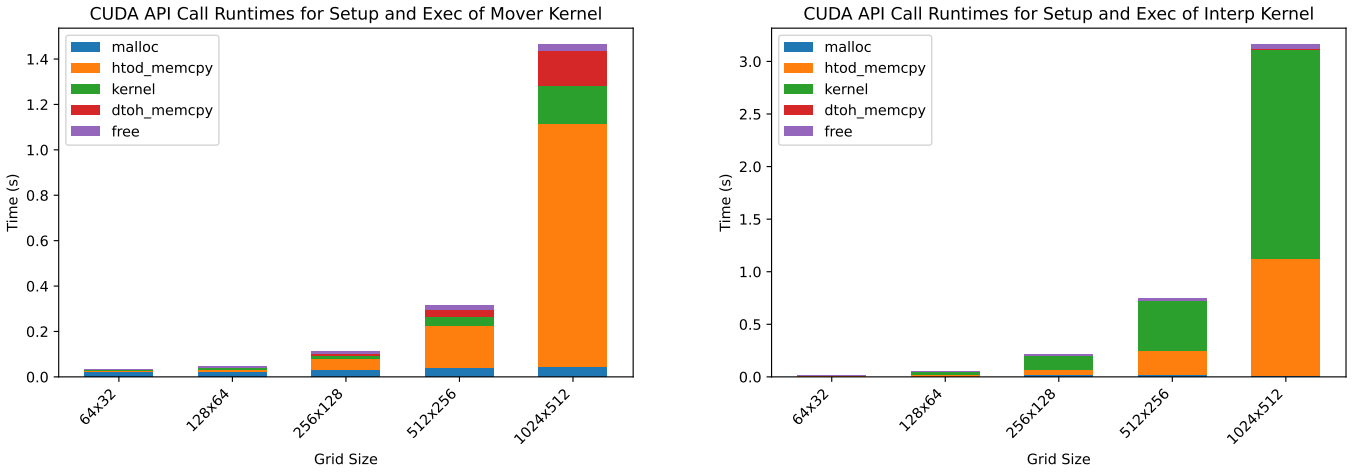
To address the main bottleneck we switched our attention to optimising the transfer of information which is updated during simulation: particles (`struct particles`) and interpolated densities (`interpDensSpecies`).

We used *streams* to address the bottleneck of memory transfer time for these entities. Our hope was to exploit CUDA streams to *mask GPU-CPU latency* with the overlap of communication and computation. With this, computation on mutually independent parts of the problem can occur asynchronously, meaning that not all of the data needs to be on the GPU before the computation can begin, and not all of the computation needs to have been completed before the computation which has completed can be transferred back to the CPU.

In addition to using streams, we also noticed that before the computation cycle, the interpolated densities for each of the species is set to 0. In this optimization, we also exploited this fact, using `cudaMemset` instead of `cudaMemcpy` (`async` versions of `memset` where appropriate) to initialize the interpolated densities on the GPU. We also realized that while copying the interpolated densities and particles back and forth during the simulation, we would only need to `malloc` and `free` the GPU memory responsible for holding this information once, which would save us minor runtime that is spent on `malloc` and `free` calls in the kernel setup functions for each of these kernels. Finally, we realized that we did not need to copy entire structures (such as `struct particles`) to the GPU, since we were already copying the contained data (such as `part->x`) that we were going to use. This allowed us to remove additional `cudaMemcpy` call, further simplifying our kernel setup at the expense of adding more arguments to the kernel, which seemed like a fair trade-off.

When it comes to splitting a problem into independent chunks for streaming, there were 2 natural approaches: split based on *species* or based on *particles*. For both these streaming approaches, we used 4 CUDA streams.

1) Species Streaming: In the original CPU version, most computation is performed species by species. It thus felt



(a) Mover Kernel

(b) Interpolation Kernel

Fig. 3: Runtimes grouped by overhead

natural to allocate each species to a stream. For each species in a simulation cycle, the movement of particles and interpolation of densities is then performed independently. This particle movement for each species can be given to one stream, allowing for data-movement and computation of new particle positions for one species to occur simultaneously with another species. The same occurs for the interpolation of densities. This idea is presented in pseudocode form in Listing 2 in the Appendix.

An additional thing to take care of when using streams is the utilization of *Pinned Memory* for data that will be transferred to and from the GPU asynchronously. The `mover_PC` step needs to asynchronously transfer the particles to and from the GPU, and the `interpP2G` step needs to perform this asynchronous transfer for particles and interpolated densities. To this end, we use `cudaHostAlloc` and `cudaFreeHost` to manage the lifecycles of the data involved in the asynchronous transfers. An example of this for particle arrays (such as `part->x`) can be seen in Listing 3 in the Appendix.

One possible downside of this species-wise streaming approach could be uneven load distribution across streams if one species is fairly large or involves more complex computation than the others. This led us to also explore a more *judicious* streaming approach, that of particles.

2) Particle Streaming: Instead here we break the particle arrays (such as `part->x` or `part->u`) for each species into smaller segments. This is done for each species in each call to the `moverPC` or the `interpP2G` steps in all cycles.

The movement of particles (the `mover_PC_gpu` call) for a given species is now broken down in to smaller chunks and performed asynchronously with the associated data-movement (both HtoD and DtoH), before this same step is repeated for the next species. The interpolation of densities (the `interpP2G_gpu` call) overlaps HtoD movement of particle information with the computation of the interpolated densities, but the initialization of the densities on the GPU and the DtoH transfer of the computed densities happens synchronously.

The reason for the synchronous movement of density related information for the interpolation kernel in this approach is the logic of the kernel. We parallelize both kernels by assigning one GPU thread to perform computation for one particle. The kernel access 8 cells for each density array, meaning that the access to elements of the density arrays from different segments, if they were streamed as well, could lead to incorrect answers or program crashes. Thus, the overlap of the HtoD transfer of the particle arrays and the kernel computation is sandwiched by the synchronous initialization and DtoH transfer of the density arrays.

When it came to dividing the particles array into different number of segments which could be worked on in parallel, we chose 32 segments. Thus, a maximum of $\lceil \frac{\#particles}{32} \rceil$ elements of particle arrays are transferred and computed on the GPU be each stream at a time, with the possibility of the last segment having fewer elements if the number of particles isn't perfectly divisible by 32. The kernel launch configurations also differ slightly for compared to the previous iterations. We still have 1024 threads per block, but now each kernel call handles around $\lceil \frac{\#particles}{32} \rceil$ elements, so this value is used to compute the number of blocks for each segment. This size of 32 for segments was chosen for the lowest observed runtime as compared to larger segment sizes. Pseudocode illustrating the particle streaming approach with async transfers and kernel calls can be seen in Listings 4 (for particle mover) and 5 (for density interpolation).

3) Verifying Communication-Computation Overlap: To verify that both versions of streaming execute with multiple streams and have overlaps in communication and computation, we utilized NVIDIA Nsight Systems [4]. Figure 11 shows the overlaps for the Species Streaming, and Figure 12 shows the overlaps for the Particle Streaming. Note that for the former, there is an HtoD and DtoH overlap for both kernels,

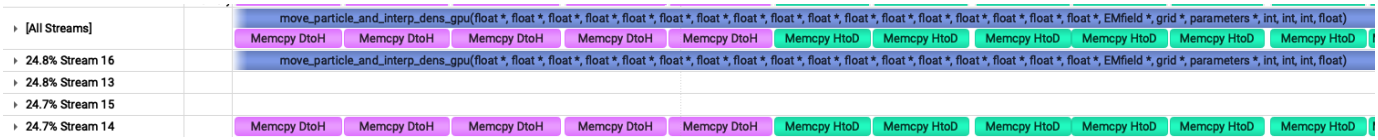


Fig. 4: Overlap for Fused Kernel with Particle Streaming

in addition to a `memset` overlap because of the use of `cudaMemsetAsync` for initializing the densities. However, for the latter, there is no DtoH overlap for the interpolation kernel (and neither for the memset) as that is done synchronously – as discussed earlier.

D. Kernel Fusion

The final optimization we worked on built on top of streaming and took the idea of parallel compute one step further. In both the streaming approaches mentioned earlier, we first compute the new locations of the particles for all species through `moverPC_gpu`, and only after that do we compute the interpolated densities from these movements through `interpP2G_gpu`. However, for a given particle of a given species, the movement and interpolation of densities from said movement is independent of the movement and interpolation of *any* other particle, regardless of species. Thus, both the movement and the interpolation steps for a given particle can happen at the same time as the movement and the interpolation steps of any other particle.

We exploited this observation by *fusing* the `move_particle_gpu` and the `interp_particle_gpu` kernels in to one big kernel: the `move_particle_and_interp_dens_gpu` kernel. Now, instead of 2 separate loops for movement and then for interpolation for each species, the main simulation loop consists of just one for-loop which calls the launcher of this kernel for all species. The function responsible for setting up and launching this kernel: `mover_and_interp_gpu` in the `src/Particles.cu` file, handles the initialization and transfer of data on the GPU before launching the fused kernel. This kernel then does all the steps performed by the original mover kernel, followed by the steps done by the original interpolation kernel for each particle.

This fused kernel builds on top of the Particle Streaming approach from Section II-C2. Thus, the particle array is divided in to smaller segments and is copied asynchronously over to the GPU. After the computation, the new locations of the particles are copied back to the CPU asynchronously. The interpolated densities are initialized on the GPU and copied over to the CPU using synchronous versions of `cudaMemset` and `cudaMemcpy` respectively, for reasons discussed in Section II-C2. The overlap between the HtoD transfer of particles, the execution of the joint kernel, and the DtoH transfer of particles can be seen in Figure 4

The main benefit of fusion is to reduce the overhead of launching 2 separate kernels which perform computations in succession (as seen in Figure 1.

E. Correctness Verification

To verify that the optimised versions are ”reasonably” correct we compare the outputs of the optimised versions with that of the original CPU simulation. This was done through the ‘check.py‘ script present in the root directory of the project folder(s). To run the script one needs to provide the `data` folder paths which contain the `.vtk` files generated by a reference implementation and the implementation to be checked. To be precise: we compared *rho net*, *rhoe* and *rhoi*. We simulated the CPU and GPU version for 100 iterations. For every point within the mesh, we checked that the absolute difference between CPU and GPU version is smaller than $1e-4$. This test was passed by all implementations presented in this report. An example of the test ran for the fused kernel implementation from section II-D is shown in Figure 13

F. Runtime Measurements

To measure the runtime of various implementations, we executed the simulation on various grid sizes, ranging from 64×32 up to 1024×512. Per grid cell we used 125 particles per species. Each simulation was run for 10 cycles using 4 species and no sub-cycling. We ran all simulation 5 times and provide the mean runtime.

For all implementations except for the Fused Kernel, we tracked the total runtime, and the runtime of the mover call and the interpolation call separately. The total runtime here includes work done on the CPU even for the GPU implementation, since we only focused on porting the interpolation and the mover step on the GPU The total runtime is presented to show an impact of how the speedup would feel in an *actual* simulation scenario, and the individual mover and interpolation provide for a more localized evaluation.

For the Fused Kernel, since both these calls are now replaced by a single call, we track a *combined* runtime of the mover and interpolation step instead of separate runtimes. This runtime is also tracked for other GPU implementations to get a better reflection of GPU performance. This is differentiated from the runtime of the entire simulation, which we refer to as the *total* runtime.

For investigating our utilization of streams, we use *nsys*. We use `nsys profile` followed by the invocation of the binary for the simulator to generate an `*.nsys-rep` file, which we then open using NVIDIA Nsight Systems to investigate.

III. Experimental Setup

All runtime tests were performed on the EECS school cluster. This provides access to a MIG partition of a H100 GPU, where the partition is one eigth of the overall device.

TABLE I: Speedup vs CPU (baseline) - Mover and Interpolation Combined

Grid Size	Naïve GPU	Consolidation GPU	Particle Streams GPU	Species Streams GPU	Fused Kernel GPU
64×32	2.55×	4.48×	7.45×	8.83×	6.89×
128×64	5.18×	6.89×	12.28×	12.99×	12.74×
256×128	8.02×	8.91×	15.39×	16.24×	17.39×
512×256	9.10×	10.55×	20.62×	20.75×	24.42×
1024×512	11.00×	10.75×	21.50×	21.88×	24.55×

H100 GPUs have 144 SMs with 80GB of VRAM. The partition then provides 16 SMs with 10GB of RAM.

IV. Results

We first present a summary of our findings, after which we will explore the individual implementations in their own sections. The benchmarking reveals a big overall speedup for all GPU implementations. As the overall best performing implementation, the fused kernel achieved up to 10.40x speedup compared to our CPU baseline implementation at the largest grid size (1024×512) for the total simulation runtime. Compared to just the runtime of the mover and interpolation step, the fused kernel achieved a 24.55x speedup compared the CPU for the largest grid size. The biggest performance improvements were found in the *moverPC* method. Due to its embarrassingly parallel structure it achieves over 100x speedup (see table V) for the streamed GPU versions compared to the CPU version for that part of the simulation specifically. In comparison, the interpolation part "only" performed up to 10x better (table VI). This phenomenon is also seen in the GPU optimizations compared to the Naïve GPU implementation, where mover speedups seem to be larger than the interpolation speedups. We believe that this is largely due to the use of atomic operations, which are needed to handle race conditions. This becomes a bottleneck, especially for smaller grids with many particles. In addition, Figure 3 also suggests that the `moverPC_gpu` is memory bound (HtoD copies) while the `interpP2G_gpu` seems kernel bound. The optimizations to the naïve GPU implementation were not aimed at addressing the logic of the kernels, and thus it is likely that despite the better memory transfer because of the optimizations, the kernel logic of the interpolation function (or the interpolation section of the code for the fused kernel) dominates runtime.

We also want to emphasize that we only run the simulation for 10 cycles (due to feasibility reasons). This means that the initialization contributes to a large part to the total runtimes, which explains the modest speedups factors compared to the particle mover speedups.

With regards to the performance of the GPU optimizations compared to the Naïve GPU implementation, we observe that the streamed-particles, streamed-species, and the fused kernel versions provide noticeable speedup, offering up to 1.95x, 1.99x, and 2.23x quicker runtimes for the mover + interpolation step for the largest grid size. While giving some speedups for smaller grid sizes, the consolidated GPU optimization seems to not offer much advantage, being very similar in performance to the naïve version for the largest grid size.

Now, we will first focus on the Naïve GPU implementation, comparing its runtime to the base CPU version. Afterwards, we focus on the 4 optimizations to the Naïve GPU implementation (consolidation, streams with particles, streams with species, and fused kernel) and compare their performance to it.

A. Analyzing Scaling Behaviour

We can clearly a big difference in how well the GPU outperforms the CPU with regards to the problem size. This is visible in Figure 5, where we see considerable speedup in even the Naïve GPU version compared to the CPU version for all grid sizes for both the functions, and thus the total runtime. This is to be expected, due to the fact that larger problems better amortize GPU overhead costs. Specifically for the moverPC kernel, we can see that the speedup itself is around 8x more for the biggest grid size (1024×512) compared to the smallest one (64×32) for the Naïve GPU implementation.

B. Consolidation

Moving memory life-cycle handling and copying outside the simulation loop for the Grid, Field, and Parameters provides marginal improvement compared to the Naïve GPU implementation for smaller grids, as can be seen in Figure 8.

For larger grid sizes, the speed-up is negligible (around 0.98 for the largest grid). This indicates that any speed-up gained by removing the transfer of grid, field, and parameters from the simulation loop was nullified by the growth in the number of particles with an increase in grid size. In this implementation, there is still synchronous HtoD transfer of particles occurring in each simulation loop in each call to either kernels (in addition to the HtoD movement of interpolated densities for the interpolation kernel). For smaller grid sizes, the number of particles being transferred might have been *small enough* to be marginally comparable to the grid, field, and parameters.

However, for larger grid sizes, the growth in the number of particles likely keeps the HtoD transfer times as high as those seen in Figure 3.

C. Streaming

Both streaming approaches, particle-wise and species-wise, provide significant speedups of of 1.95 and 1.99x (respectively) for the largest grid size over the naïve GPU version for the combined runtimes of the mover and interpolation steps. Species-wise streaming slightly outperforms particle-wise streaming, but even then both version seem to perform very similarly, as can be seen by their proximity to each

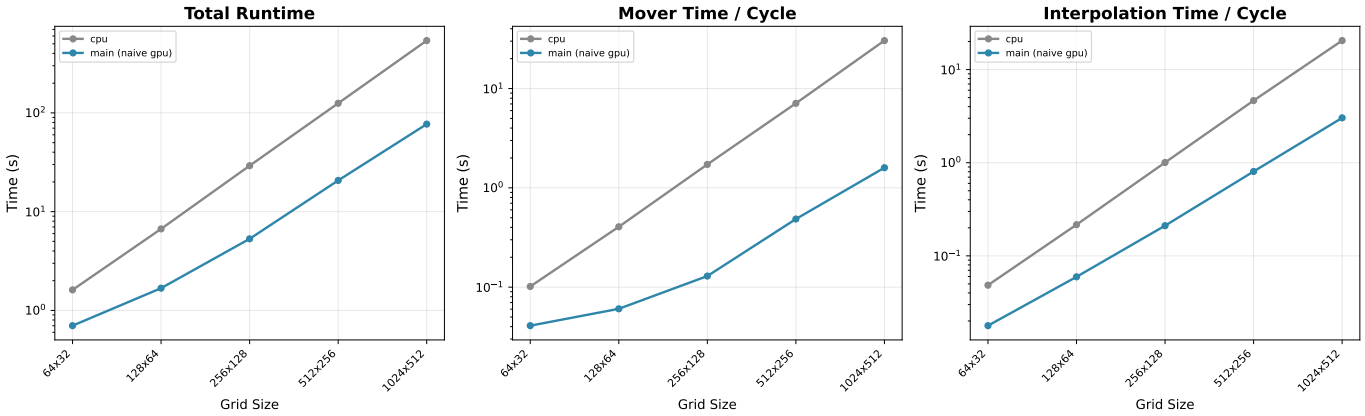


Fig. 5: CPU Implementation vs Naïve GPU Implementation

other in Figure 9. The difference is most noticeable in the performance of the particle movement step. One reason for this could be the *well balanced* problem distribution when it comes to our input files. As mentioned in Section II-F, each species has 125 particles per grid cell, meaning there are the same number of particles for all species. Thus each stream has an even load distribution, which might contribute to slightly better use of GPU resources compared to the Particle-wise streaming. To test this, we experimented with an input file in which one species had 131Mil particles, while the other 3 had 4Mil. We ran this experiment 5 times and recorded the mover and interpolation step runtimes for Particle-streaming and Species-streaming. The results of this can be seen in Figure 6. Here, we see that Particle-wise streaming outperforms species-wise streaming, potentially because species-streaming loses the *well-balanced* advantage here. But in either case, both these optimized streaming versions easily outperform the naïve GPU version. This is attributed to the fact that memory transfers (Host-to-Device (HtoD) and Device-to-Host (DtoH))

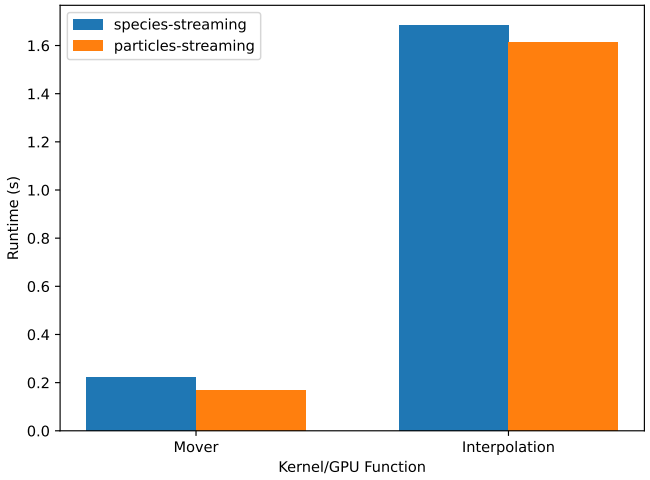


Fig. 6: Stream Comparison

consume a significant portion of the runtime. By overlapping computation with memory transfers, as shown in Figures 11 and 12, we successfully decrease the runtime.

D. Kernel Fusion

Fusing the mover and interpolation kernels into a single kernel launch while doing particle-wise streaming provided the biggest speedup for the combined mover and interpolation step over the naïve GPU version, ranging from 2.23x - 2.70x for the grid sizes considered. A visual representation of this can be seen in Figure 10.

This shows that performing *both* the mover and interpolation step for all particles *at once* in parallel, instead of performing the mover step and then performing the interpolation step was advantageous. In addition to this, the elimination of redundant CUDA overhead of launching more kernels than required was also reduced, which likely played a small role in reducing the runtime as well.

Figure 7 pits the Fused Kernel against all GPU implementations, where for the largest grid sizes it comes out as the victor. For smaller grid sizes, it is slower than the simple Streaming implementations – likely because of a slightly greater overhead of launching a complex kernel. However, this complexity starts paying dividends for larger grid sizes.

One drawback of this approach is that it increases the complexity of the kernel and could lead to code that is less readable. This can be circumvented by using `__device__` functions if desired.

V. Conclusion

Our GPU acceleration of the simplified iPIC3D simulation achieves significant performance gains through systematic optimizations. A massive computational improvement was achieved in going from the CPU implementation to a naïve GPU implementation by exploiting the embarrassingly parallel structure of the particle mover and density interpolation. Then, eliminating the memory management and movement overhead of constant entities from each simulation loop provided a

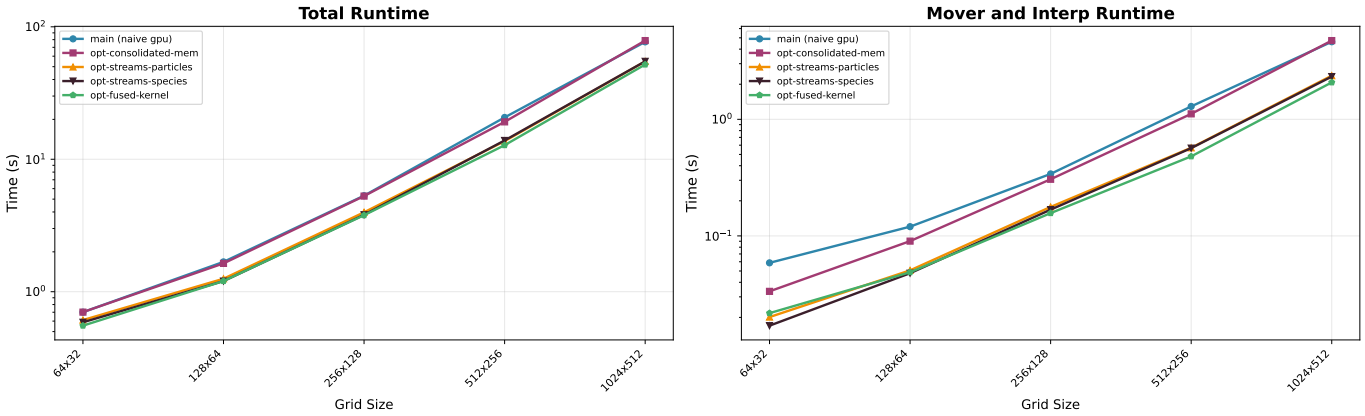


Fig. 7: Naïve GPU vs All GPU Optimizations

slight performance boost for smaller grid sizes through consolidation. However, this performance benefit was quick to fade as the grid sizes increased, likely to the explosion in the number of particles which were still being transferred back and forth every simulation This paved the way for us to explore overlapping transfers with computation through streaming, where species-wise streaming and particle-wise streaming were investigated. Both particle-wise and species-wise streaming performed comparably well Finally, fusion of the mover and interpolation step (on top of particle-streaming) provided the best runtimes out of all of our implementations.

The interpolation kernel did not optimize as well likely due to atomic operations which are necessary due to the fact that particles contribute to overlapping grid cells. A secondary reason could be the fact that the optimizations were not aimed at addressing or altering the logic of the kernels too much, and since the interpolation seems to be kernel bound according to Figure 3, any potential improvements because of better memory management were dwarfed by the kernel runtime. Numerical differences between CPU and GPU implementations (within 10^{-4} tolerance) arise from different floating-point ordering and potential fused multiply-add (FMA) operations on the GPU. We verified this using a comparison script to catch bugs during development.

In conclusion, we achieved a speedup of 24.55x on the biggest input with the fused kernel approach over the CPU version for the combined mover and interpolation steps through systematic GPU optimization. This approach also achieved a speedup of 2.23x with the biggest input compared to the naïve GPU implementation from which we started. The combination of memory consolidation, CUDA streams for overlapping transfers with computation, and kernel fusion provides a solid foundation for further optimization of more complete PIC simulations.

A. Limitations

Our simulation does not divide the particle computation in batches. This means that during computation, all particles for a given species are on the GPU simultaneously. This limits the maximum size of grid we can simulate: once the number of particles in the grid exceeds the maximum allocated memory on the GPU, we cannot proceed with computation.

This implementation doesn't involve the field solver. Without the field solver, there is no need to transfer density information back and forth. We simply set the densities to zero at the start of each cycle rather than transferring the previous cycles particle and density information (as this would require the field solver step). This is a specious optimization: for realistic simulations this would not be possible as the reevaluated density information would be essential for simulation accuracy.

B. Future Work

Adding batching of particle computation would allow us to bypass the upper limit on grid size imposed by the available GPU memory. This would involve batching the particle information in the intialization, transfer to the GPU in chunks, and only performing the kernel computation on the individual batches. Additionally, we would also like to explore the impact of fused kernel with the species-wise streaming approach.

References

[1] S. Markidis, G. Lapenta, and Rizwan-uddin, "Multi-scale simulations of plasma with ipic3d," *Mathematics and Computers in Simulation*, vol. 80, no. 7, pp. 1509 – 1519, 2010, multiscale modeling of moving interfaces in materials. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378475409002444

[2] I. Peng, "Dd2360-ht25 simplified ipic3d repository," https://github.com/KTH-ScaLab/DD2360-HT25, 2025.

[3] ——, "Introduction of spunitgpu.pdf," https://canvas.kth.se/courses/57317/files/9711370?wrap=1, 2025.

[4] NVIDIA Corporation, "Nvidia nsight systems," 2025, accessed: 2026-01-07. [Online]. Available: https://docs.nvidia.com/nsight-systems/index.html

APPENDIX

A. Plots

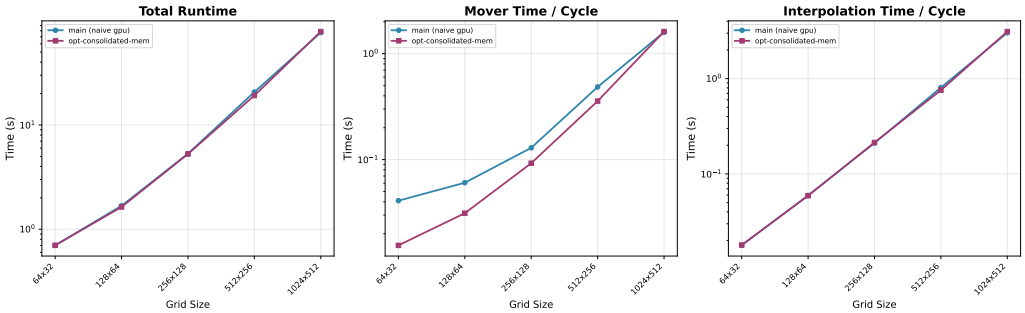


Fig. 8: Naïve GPU vs Consolidation Optimization

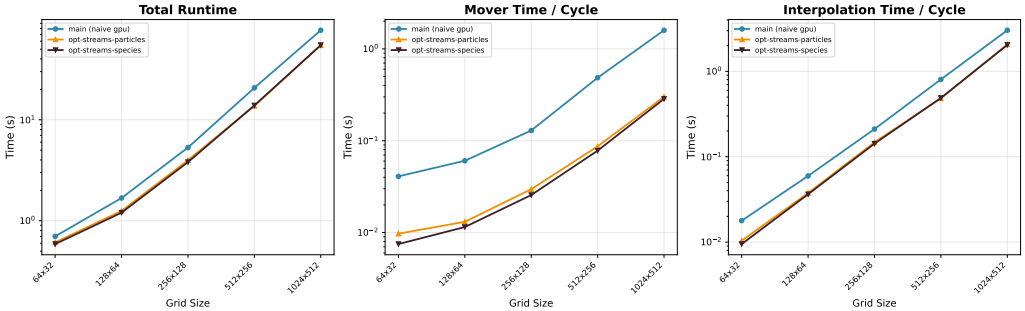


Fig. 9: Naïve GPU vs Streams

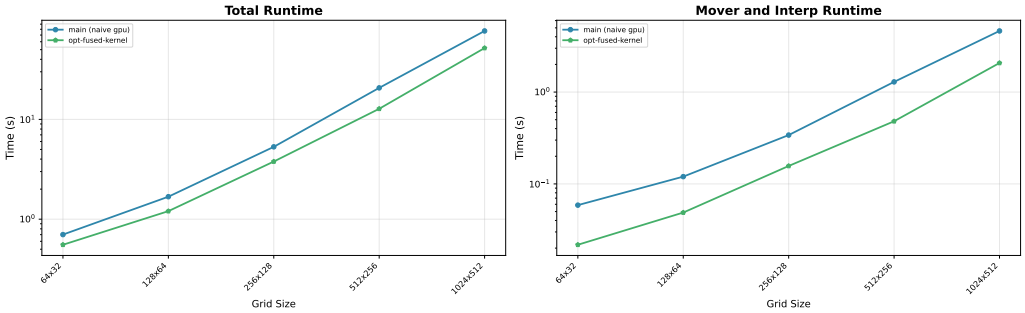
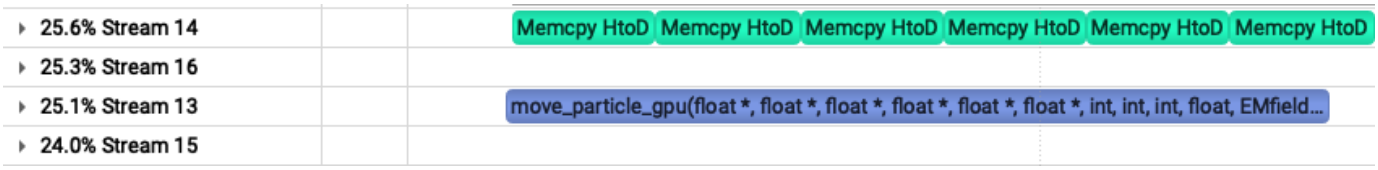
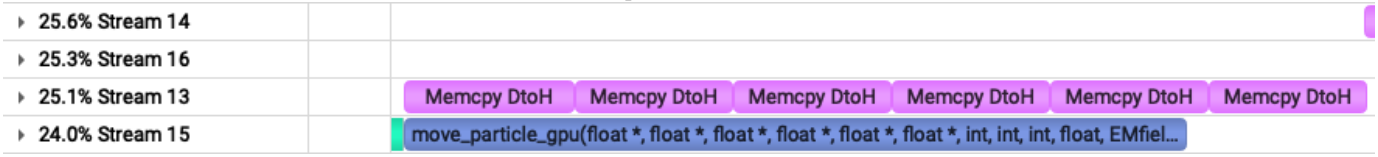


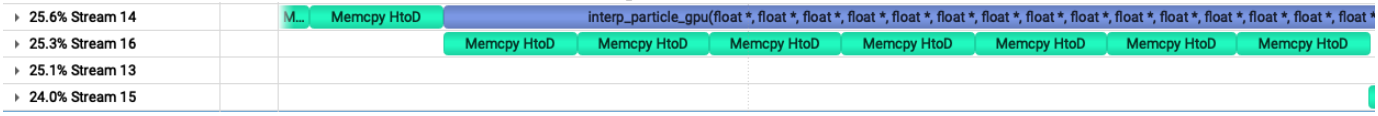
Fig. 10: Naïve GPU vs Fused Kernel



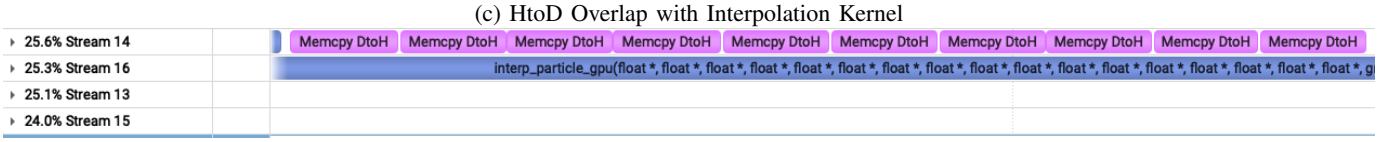
(a) HtoD Overlap with Particle Mover Kernel



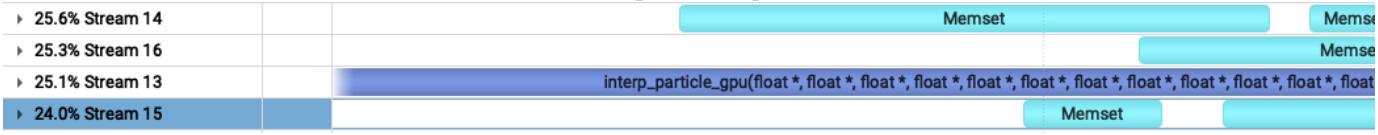
(b) DtoH Overlap with Particle Mover Kernel



(c) HtoD Overlap with Interpolation Kernel

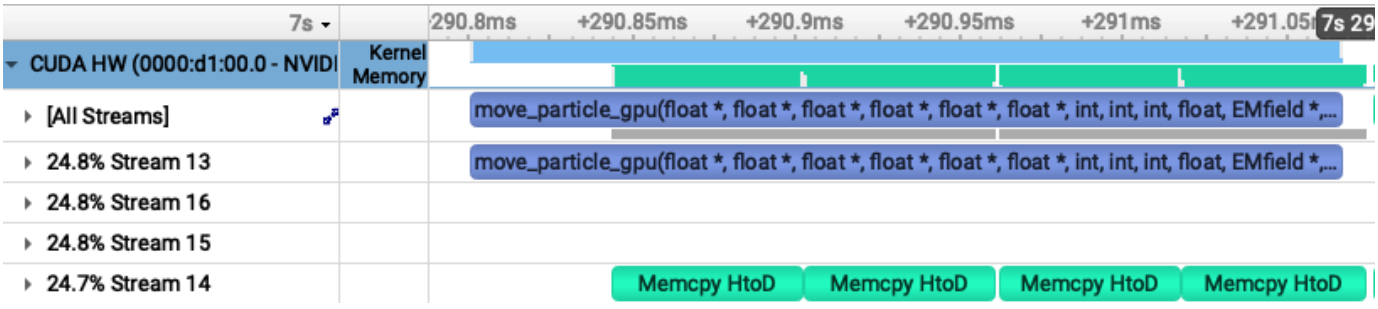


(d) DtoH Overlap with Interpolation Kernel



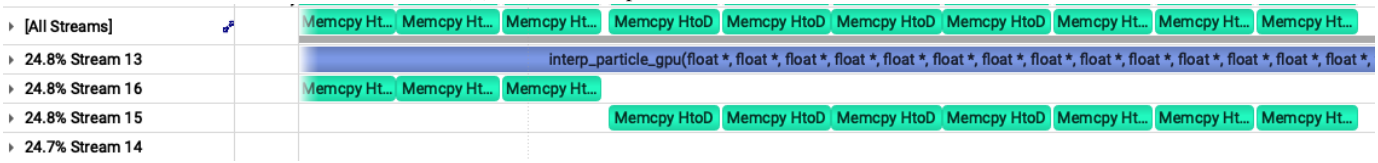
(e) Memset Overlap with Interpolation Kernel

Fig. 11: Overlaps for Species Streaming



(a) HtoD Overlap with Particle Mover Kernel

(b) DtoH Overlap with Particle Mover Kernel



(c) HtoD Overlap with Interpolation Kernel

Fig. 12: Overlaps for Particle Streaming

```
jovyan@jupyter-rishiv:~/DD2360-GPU/project/iPIC3D-mini$ python3 check.py data data_cpu 1e-4
[OK]   rhoe_60.vtk: max difference 5.000e-07
[OK]   rhoe_40.vtk: max difference 2.000e-07
[OK]   rhoe_80.vtk: max difference 1.100e-06
[OK]   rho_net_50.vtk: max difference 2.820e-07
[OK]   rhoi_100.vtk: max difference 1.000e-07
[OK]   rhoe_50.vtk: max difference 3.000e-07
[OK]   rho_net_70.vtk: max difference 1.100e-06
[OK]   rhoi_20.vtk: max difference 1.000e-07
[OK]   rhoi_90.vtk: max difference 1.000e-07
[OK]   rho_net_60.vtk: max difference 5.400e-07
[OK]   rhoe_100.vtk: max difference 1.600e-06
[OK]   rho_net_90.vtk: max difference 1.120e-06
[OK]   rhoe_30.vtk: max difference 1.000e-07
[OK]   rho_net_80.vtk: max difference 1.120e-06
[OK]   rhoe_90.vtk: max difference 1.100e-06
[OK]   rhoe_10.vtk: max difference 1.000e-07
[OK]   rhoi_70.vtk: max difference 1.000e-07
[OK]   rhoi_30.vtk: max difference 1.000e-07
[OK]   rho_net_20.vtk: max difference 1.000e-07
[OK]   rho_net_100.vtk: max difference 1.540e-06
[OK]   rhoi_40.vtk: max difference 1.000e-07
[OK]   rhoe_20.vtk: max difference 1.000e-07
[OK]   rhoi_50.vtk: max difference 1.000e-07
[OK]   rho_net_40.vtk: max difference 1.700e-07
[OK]   rhoe_70.vtk: max difference 1.000e-06
[OK]   rhoi_10.vtk: max difference 1.000e-07
[OK]   rho_net_10.vtk: max difference 7.000e-08
[OK]   rhoi_80.vtk: max difference 1.000e-07
[OK]   rhoi_60.vtk: max difference 1.000e-07
[OK]   rho_net_30.vtk: max difference 1.100e-07

All checked files are within tolerance.
```

Fig. 13: Output Verification with Tolerance $1e-4$

B. Tables

TABLE II: Total Runtime (s) - Mean across 5 runs

Grid Size	CPU	Naïve GPU	Consolidation GPU	Particle Streams GPU	Species Streams GPU	Fused Kernel GPU
64×32	1.614	0.701	0.701	0.611	0.588	0.554
128×64	6.689	1.680	1.636	1.253	1.203	1.204
256×128	29.187	5.308	5.272	3.976	3.808	3.781
512×256	125.105	20.691	19.149	13.707	13.862	12.771
1024×512	539.600	76.954	78.711	54.732	54.856	51.893

TABLE III: Speedup vs CPU (baseline) - Total Runtime

Grid Size	Naïve GPU	Consolidation GPU	Particle Streams GPU	Species Streams GPU	Fused Kernel GPU
64×32	2.30×	2.30×	2.64×	2.75×	2.91×
128×64	3.98×	4.09×	5.34×	5.56×	5.56×
256×128	5.50×	5.54×	7.34×	7.66×	7.72×
512×256	6.05×	6.53×	9.13×	9.02×	9.80×
1024×512	7.01×	6.86×	9.86×	9.84×	10.40×

TABLE IV: Speedup vs main (naïve GPU) - Mover and Interpolation Combined

Grid Size	Consolidation GPU	Particle Streams GPU	Species Streams GPU	Fused Kernel GPU
64×32	1.76×	2.92×	3.46×	2.70×
128×64	1.33×	2.37×	2.51×	2.46×
256×128	1.11×	1.92×	2.03×	2.17×
512×256	1.16×	2.27×	2.28×	2.68×
1024×512	0.98×	1.95×	1.99×	2.23×

TABLE V: Speedup vs CPU (baseline) - Mover Time / Cycle

Grid Size	Naïve GPU	Consolidation GPU	Particle Streams GPU	Species Streams GPU
64×32	2.48×	6.55×	10.44×	13.54×
128×64	6.70×	12.99×	30.95×	35.28×
256×128	13.32×	18.62×	58.12×	67.21×
512×256	14.64×	19.90×	81.69×	91.07×
1024×512	19.07×	18.80×	101.04×	106.45×

TABLE VI: Speedup vs CPU (baseline) - Interpolation Time / Cycle

Grid Size	Naïve GPU	Consolidation GPU	Particle Streams GPU	Species Streams GPU
64×32	2.72×	2.70×	4.66×	5.11×
128×64	3.63×	3.66×	5.76×	5.94×
256×128	4.77×	4.71×	6.81×	7.06×
512×256	5.77×	6.14×	9.62×	9.52×
1024×512	6.76×	6.57×	9.91×	10.04×

C. Code Listings

Listing 1: GPU Memory Allocation

```
struct particles *d_part = nullptr;

struct particles l_part = *part;

FPpart *d_part_x = nullptr;
FPpart *d_part_y = nullptr;
FPpart *d_part_z = nullptr;
FPpart *d_part_u = nullptr;
FPpart *d_part_v = nullptr;
FPpart *d_part_w = nullptr;

cudaMalloc(&d_part_x, size_part);
cudaMalloc(&d_part_y, size_part);
cudaMalloc(&d_part_z, size_part);
cudaMalloc(&d_part_u, size_part);
cudaMalloc(&d_part_v, size_part);
cudaMalloc(&d_part_w, size_part);

cudaMemcpy(d_part_x, part->x, size_part, cudaMemcpyHostToDevice);
cudaMemcpy(d_part_y, part->y, size_part, cudaMemcpyHostToDevice);
cudaMemcpy(d_part_z, part->z, size_part, cudaMemcpyHostToDevice);
cudaMemcpy(d_part_u, part->u, size_part, cudaMemcpyHostToDevice);
cudaMemcpy(d_part_v, part->v, size_part, cudaMemcpyHostToDevice);
cudaMemcpy(d_part_w, part->w, size_part, cudaMemcpyHostToDevice);

cudaMalloc(&d_part, sizeof(struct particles));

cudaMemcpy(d_part, &l_part, sizeof(struct particles), cudaMemcpyHostToDevice);
```

Listing 2: Simulation Loop Pseudocode for Species Streaming

```
(..)
for (int i = 0; i < NUM_CUDA_STREAMS; i++) {
  cudaStreamCreate(&streams[i]);
}

(...)

while(performing cycles) {
  for (int is = 0; is < species; is++)  {
    int stream_id = get_stream_for_species(is);
    mover_PC_gpu(..., streams[stream_id]);
  }

  synchronize();

  for (int is = 0; is < species; is++) {
    int stream_id = get_stream_for_species(is);
    interpP2G_gpu(..., streams[stream_id]);
  }

  synchronise();
}

  (...)
}
```

Listing 3: Particle Array Alloc and Free in Pinned Memory

```
void particle_allocate(...) {
  (..)
  cudaHostAlloc((void **) &(part->x), npmax * sizeof(FPpart), cudaHostAllocDefault);
  (..)
}

void particle_deallocate(...) {
(..)
cudFreeHost(part->x);
(..)
}
```

Listing 4: Particle Stream Kernel Setup and Launch for Mover

```
for (i = 0; i < numSegments; i++) {
  int streamId = i \% num_streams
  int segmentStart = ...;
  (..)
  asyncCopy(&d_part->x[segmentStart], &h_part->x[segmentStart], ..., HtoD, stream[streamId]);
  (..)
  TPB = 1024; NBlocks = (elementsInSegment + TPB - 1) / TPB;
  mover_kernel<<<NBlocks, TPB, 0, stream[streamId]>>>(&d_part->x[segmentStart], ...);
  (..)
  asyncCopy(&h_part->x[segmentStart], &d_part->x[segmentStart], ..., DtoH, stream[streamId]);
}

synchronize();
(..)
```

Listing 5: Particle Stream Kernel Setup and Launch for Interpolation

```
// Synchronous memset
cudaMemset(d->rhon_flat, 0, ...);
(..)
for (i = 0; i < numSegments; i++) {
  int streamId = i \% num_streams
  int segmentStart = ...;
  (..)
  asyncCopy(&d_part->x[segmentStart], &h_part->x[segmentStart], ..., HtoD, stream[streamId]);
  (..)
  TPB = 1024; NBlocks = (elementsInSegment + TPB - 1) / TPB;
  interp_kernel<<<NBlocks, TPB, 0, stream[streamId]>>>(&d_part->x[segmentStart], ...)
}
synchronize();

// Synchronous DtoH
copy(h->rhon_flat, d->rhon_flat, ..., DtoH)
}
```