

Optimizing BIT1, a Particle-in-Cell Monte Carlo Code, with OpenMP/OpenACC and GPU Acceleration

Jeremy J. Williams¹, Felix Liu¹, David Tskhakaya², Stefan Costea³, Ales Podolnik², and Stefano Markidis¹

¹ KTH Royal Institute of Technology (KTH), Stockholm, Sweden

² Institute of Plasma Physics of the CAS (IPP CAS), Prague, Czech Republic

³ LeCAD, University of Ljubljana (UL), Ljubljana, Slovenia

Abstract. On the path toward developing the first fusion energy devices, plasma simulations have become indispensable tools for supporting the design and development of fusion machines. Among these critical simulation tools, BIT1 is an advanced Particle-in-Cell code with Monte Carlo collisions, specifically designed for modeling plasma-material interaction and, in particular, analyzing the power load distribution on tokamak divertors. The current implementation of BIT1 relies exclusively on MPI for parallel communication and lacks support for GPUs. In this work, we address these limitations by designing and implementing a hybrid, shared-memory version of BIT1 capable of utilizing GPUs. For shared-memory parallelization, we rely on OpenMP and OpenACC, using a task-based approach to mitigate load-imbalance issues in the particle mover. On an HPE Cray EX computing node, we observe an initial performance improvement of approximately 42%, with scalable performance showing an enhancement of about 38% when using 8 MPI ranks. Still relying on OpenMP and OpenACC, we introduce the first version of BIT1 capable of using GPUs. We investigate two different data movement strategies: unified memory and explicit data movement. Overall, we report BIT1 data transfer findings during each PIC cycle. Among BIT1 GPU implementations, we demonstrate performance improvement through concurrent GPU utilization, especially when MPI ranks are assigned to dedicated GPUs. Finally, we analyze the performance of the first BIT1 GPU porting with the NVIDIA Nsight tools to further our understanding of BIT1’s computational efficiency for large-scale plasma simulations, capable of exploiting current supercomputer infrastructures.

Keywords: OpenMP · Task-Based Parallelism · OpenACC · Hybrid Programming · GPU Offloading · Large-Scale PIC Simulations

1 Introduction

Plasma simulations are vital for understanding complex interactions between plasma and wall materials, which present significant modeling challenges, including the need of resolving different simulation time and spatial scales or modeling accurately atomic and collision processes.

Particularly, these challenges are notable when modeling plasma-loaded divertors in fusion devices, such as the ITER tokamak, a major nuclear fusion project. In summary, the divertor manages the heat and particle fluxes that occur during the operation of the tokamak. In fact, during a fusion reaction, high energy neutrons are produced, and these can cause damage to the first wall of the tokamak. Additionally, impurities from the plasma, need to be efficiently removed to maintain optimal conditions for the fusion process. The divertor accomplishes these tasks by diverting the flow of plasma to a specific region of the tokamak. This region, known as the divertor region, is usually located at the bottom of the toroidal chamber (see Fig. 1).

Among the tools used to address these challenges, BIT1 is a specialized plasma simulation tool, focusing on describing accurately atomic processes and collisions in plasmas during plasma-wall interactions. In particular, BIT1 is widely used to analyze how power is distributed on divertors in these devices. BIT1 plays a critical role as a massively parallel PIC code for studying complex plasma systems and their interactions with various materials.

Initially introduced by D. Tskhakaya and collaborators [8,9], BIT1 has distinctive capabilities. It models plasmas confined between two conducting walls and includes collision modeling to capture complex plasma dynamics. What makes BIT1 unique is its capability of modeling accurately processes occurring at the interface of plasma and a wall, such as sputtering from the wall, emissions, and collisions. While it has shown that BIT1 is scalable for thousands MPI processes [13], it has two major limitations. The first one is that BIT1 relies only on MPI for parallel communication, even for on-node communication, where a shared-memory computing approach is more convenient and can decrease the memory usage and allow for task-based approaches. The second limitation is the lack of support for running on GPU-accelerated supercomputers. Given the fact that most of the top supercomputers in the world, such as Frontier, Aurora, Eagle and LUMI, the lack of support for GPUs is a major limitation that hinders the usage of BIT1 in the largest supercomputers available. This work addresses these BIT1 limitations by designing and implementing hybrid MPI and OpenMP/OpenACC version that can exploit shared-memory and GPUs.

Recently, findings presented in [13] have led to a detailed investigation of the BIT1 code performance, pointing out performance bottlenecks, and iden-

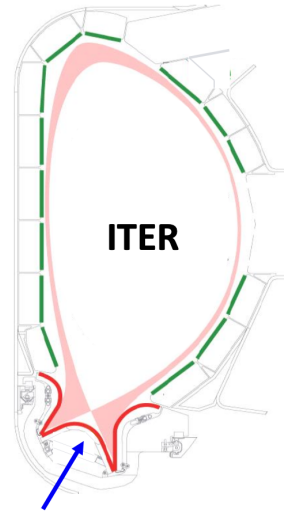


Fig. 1. BIT1 simulates plasma behavior in the tokamak divertor region (blue arrow), such as in the ITER fusion device.

tifying a roadmap for BIT1 performance optimization. The work highlighted areas for optimizing BIT1 to enhance the performance, suggesting a focus on the particle mover function, particularly the particle pusher, as an initial step, given the challenges posed by arranging particles into cells and MPI ranks. This work’s primary focus is on the enhancement of BIT1’s performance with a goal to optimize the particle mover function, one of the most computationally intensive parts of the code. This optimization utilizes OpenMP and OpenACC for multicore CPUs and GPUs, enabling researchers to harness the full potential of modern hardware for plasma simulations, ultimately improving our understanding of plasma dynamics and advancing research in fusion and plasma science. The contributions of this work are the following:

- We design and implement hybrid MPI+OpenMP and MPI+OpenACC versions of the BIT1 code to improve on-node performance. This implementation uses a task-based approach to address potential issues with load-imbalance.
- We develop the first GPU porting of the BIT1 code to NVIDIA GPUs with OpenMP and OpenACC in the particle mover stage of the BIT1 code.
- We critically analyze and discuss the performance of the newly ported BIT1 code, showing major performance improvements, and identify the next performance optimization steps.

2 Background

Particle-in-Cell (PIC) methods are among the most crucial tools for plasma simulations. They find applications in a diverse range of plasma environments, from space and astrophysical plasma to laboratory settings, industrial processes, and fusion devices. BIT1, in particular, is a PIC code specifically tailored for plasma-material simulations. What sets BIT1 apart is its handling of collisions and interactions with material boundaries, including models for phenomena like sputtering at material interfaces.

BIT1 is a 1D3V PIC code, allowing simulations in one dimension while considering particles with three-dimensional velocities. The foundation of BIT1’s computational approach is the PIC method, widely adopted in plasma physics. This method involves tracking the trajectories of millions of particles within a field consistent with density and current distributions, while abiding to Maxwell’s and Poisson equations. Fig. 2 provides a visual representation of BIT1’s explicit PIC method. To initiate the PIC simulations, BIT1 configures the computational grid and sets up particle positions and velocities for various species, including electrons and ions. Subsequently, a computational cycle iteratively updates the electric field, particle positions, and velocities, accurately representing the dynamic interactions within the plasma. BIT1 employs advanced Monte Carlo techniques to simulate collisions and ionization processes. It is important to note that one of the most computationally intensive stages in BIT1 is the particle mover, which is responsible for calculating the trajectories of millions of particles [9].

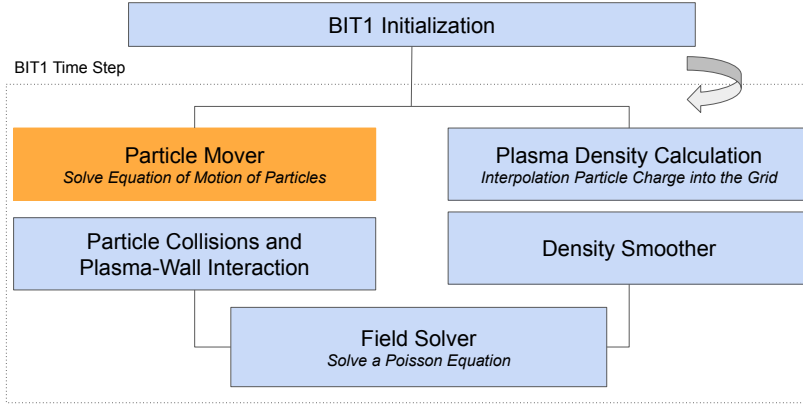


Fig. 2. A diagram representing the algorithm used in BIT1. After the initialization the PIC algorithm cycle is repeated at each time step [13]. In orange, we highlight the particle mover step that we parallelize with OpenMP and OpenACC.

In the current implementation, BIT1 employs domain decomposition for parallelization, utilizing MPI for efficient parallel communication. MPI point-to-point communication is essential for managing information exchange at domain boundaries, crucial for tasks like the smoother, Poisson solver, and handling particles exiting the computational domain. However, the existing BIT1 implementation relies solely on MPI, lacking support for hybrid parallel computing capabilities, such as MPI+OpenMP, MPI+OpenACC, GPU offloading or acceleration.

One of the main and distinctive features of the BIT1 code is the data layout particle information, such as positions and velocities, are stored in memory. BIT1 associates the particles with the cells they are located in, e.g. each grid cell has an associated list with particle information. As soon a particle move from a cell to a neighboring one, then a particle information is removed from one list and added to another one. In plasma simulations, there may be regions of space where plasma particles concentrate, resulting in situations where certain cells have a large number of particles while others have only a few. This results in work imbalance in the particle mover.

3 Methodology & Experimental Setup

In this work, our focus is on investigating the porting of the BIT1 particle mover to leverage OpenMP and OpenACC for shared memory programming and GPU acceleration.

3.1 Hybrid MPI and OpenMP/OpenACC BIT1

OpenMP Tasks Particle Mover Parallelization. OpenMP is one of the most widely used programming model designed to facilitate shared-memory parallel programming in high-performance computing (HPC) environments. The OpenMP standard is supported by major compiler suites, including GCC and LLVM, making it accessible to a broad range of developers.

```

1  #pragma omp parallel shared(chsp, sn2d, dinj, nstep, np, x, yp, vx, vy)
2      private(isp, i, j) firstprivate(nsp, nc)
3  {
4      #pragma omp single
5      {
6          for (isp = 0; isp < nsp; isp++) {
7              ...
8              #pragma omp taskloop grainsize(500) nogroup
9              for (j = 0; j < nc; j++) {
10                 #pragma omp simd
11                 for (i = 0; i < np[isp][j]; i++)
12                     x[isp][j][i] += nstep[isp] * vx[isp][j][i];
13             }
14             ...
15             #pragma omp taskloop grainsize(500) nogroup
16             for (j = 0; j < nc; j++) {
17                 #pragma omp simd
18                 for (i = 0; i < np[isp][j]; i++)
19                     x[isp][j][i] += nstep[isp] * vx[isp][j][i];
20             }
21         }
22     }
23 }

```

Listing 1.1. Simplified C code snippet illustrating the OpenMP parallelization for CPU in the particle mover.

Listing 1.1 showcases our OpenMP port of the core computational function for the particle mover. In the code, `x[][][]` and `vx[][][]` represent particle position and velocity in one dimension. `nsp` is the number of plasma species, and `nc` is the number of cells in the one-dimensional grid. `np[][]` denotes the number of particles per species per cell.

Our parallelization approach uses the OpenMP `taskloop` construct. The outermost loop, with a small number of iterations (species present in the simulation), is deemed unsuitable for traditional parallelization methods. The `taskloop` construct dynamically distributes loop iterations among available threads, ensuring effective load balancing and optimizing the parallelization strategy for enhanced performance on multicore CPUs.

When examining the code, the `#pragma omp parallel` pragma initiates a parallel region with shared variables (`chsp`, `dinj`, `sn2d`, `nstep`, `np`, `x`, `yp`, `vx`, `vy`), while `isp` and `i` are private to each thread. The `firstprivate` clause ensures private and initialized values for `nsp` and `nc` for each parallel thread.

Within the parallel region, the `single` construct ensures that the subsequent block of code is executed by a single thread, crucial for the initialization section. The `taskloop grainsize(500) nogroup` pragma parallelizes the subsequent loop, dividing iterations into tasks, while optimizing task granularity for efficient parallel execution based on empirical testing and adjustments to achieve optimal performance. The `nogroup` clause allows for dynamic scheduling.

Finally, the `simd` pragma within the innermost loop exploits SIMD parallelism, improving vectorization and the efficiency of particle movement calculations.

OpenACC Multicore Particle Mover Parallelization. OpenACC, akin to OpenMP, is a directive-based programming model primarily designed for GPU accelerators. However, the directive-based approach for GPU offloading can also be advantageous for CPUs, offering a straightforward method for porting codes to CPUs with minimal code changes.

```

1  #pragma acc parallel loop present(chsp[:lenA], sn2d[:lenA], dinj[:lenA], nstep[:lenA],
2  np[:lenA][:lenB], x[:lenA][:lenB][:lenC], yp[:lenA][:lenB][:lenC],
3  vx[:lenA][:lenB][:lenC], vy[:lenA][:lenB][:lenC])
4  {
5      for (isp = 0; isp < nsp; isp++) {
6          ...
7          #pragma acc loop gang vector
8          for (j = 0; j < nc; j++) {
9              #pragma acc loop vector
10             for (i = 0; i < np[isp][j]; i++)
11                 x[isp][j][i] += nstep[isp] * vx[isp][j][i];
12             }
13             ...
14             #pragma acc loop gang vector
15             for (j = 0; j < nc; j++) {
16                 #pragma acc loop vector
17                 for (i = 0; i < np[isp][j]; i++)
18                     x[isp][j][i] += nstep[isp] * vx[isp][j][i];
19             }
20         }
21     }

```

Listing 1.2. Simplified C code snippet illustrating the OpenACC Multicore CPU parallelization in the particle mover.

Listing 1.2 shows our optimized OpenACC parallelization for the particle mover on multicore CPUs. The `#pragma acc parallel loop` directive initiates concurrent execution, specifying essential data arrays. Utilizing `gang` and `vector` directives enhances parallel processing in a nested loop structure (`#pragma acc loop gang vector`) for particle and grid index iterations. Particle positions are updated based on `nstep` and `vx`.

In the else clause, a similar nested loop (`#pragma acc loop vector`) optimally executes particle indices (`i`). Further efficiency is achieved with a conditional statement triggering an additional nested loop to update `yp` and `x`.

This version strategically uses OpenACC directives (`#pragma acc loop gang vector` and `#pragma acc loop vector`) to optimize multicore CPUs for the particle mover function, enhancing parallel performance in nested loops.

3.2 Accelerating BIT1 with OpenMP and OpenACC

Accelerating BIT1 with OpenMP Target. Listing 1.3 illustrates our use of the OpenMP target construct to parallelize BIT1’s particle mover function for GPU offloading. The code strategically employs OpenMP target directives to optimize particle movement computations by offloading them to GPUs.

```

1  #pragma omp target enter data map(to: chsp[:lenA], sn2d[:lenA], dinj[:lenA], nstep[:
   lenA],
2  np[:lenA][:lenB], x[:lenA][:lenB][:lenC], yp[:lenA][:lenB][:lenC],
3  vx[:lenA][:lenB][:lenC], vy[:lenA][:lenB][:lenC])
4  {
5      for (isp = 0; isp < nsp; isp++) {
6          ...
7          #pragma omp target teams distribute parallel for thread_limit(256) num_teams
   (391)
8              for (j = 0; j < nc; j++) {
9                  #pragma omp simd
10                 for (i = 0; i < np[isp][j]; i++)
11                     x[isp][j][i] += nstep[isp] * vx[isp][j][i];
12             }
13             ...
14             #pragma omp target teams distribute parallel for thread_limit(256) num_teams
   (391)
15                 for (j = 0; j < nc; j++) {
16                     #pragma omp simd
17                     for (i = 0; i < np[isp][j]; i++)
18                         x[isp][j][i] += nstep[isp] * vx[isp][j][i];
19                 }
20             #pragma omp target exit data map(from: x[:lenA][:lenB][:lenC]...)
21         }
22     }

```

Listing 1.3. Simplified C code snippet illustrating the OpenMP (OMP target) parallelization with data clauses and array "shape" for GPU porting in the particle mover.

The pragma `#pragma omp target enter data` initiates the data transfer from the host to the GPU, encompassing crucial arrays such as `chsp`, `sn2d`, `dinj`, and the arrays for particle position and velocity (`x`, `yp`, `vx`, `vy`).

Within the unstructured data mapping region, the `#pragma omp target teams distribute parallel for` directive initiates worksharing across multiple levels of parallelism using combined constructs, thereby enabling the parallel execution of the nested loops for particle movement calculations on the GPU. To fine-tune parallelism, directives `thread_limit(256)` and `num_teams(391)` are used to set thread and team limits based on our experimental setup's system specifications and workload requirements.

In the nested loops, the `#pragma omp simd` directive provides a hint to the compiler for potential vectorizations of the inner loops, optimizing SIMD parallelism. The calculations, updating particle positions based on velocities and time steps, are concurrently distributed across GPU threads.

Finally, the `#pragma omp target exit data` directive ensures seamless transfer of modified data, specifically particle positions, from the GPU back to the host for efficient GPU offloading and parallelization of particle mover computations.

Accelerating BIT1 with OpenACC Parallel. OpenACC, designed for GPU acceleration, simplifies the task of offloading functions to GPUs, offering an accessible solution without complex GPU programming. Supported by platforms like NVIDIA and GCC, OpenACC empowers developers to harness GPU parallel processing efficiently.

In Listing 1.4, we showcase our OpenACC parallelization of the particle mover function for GPU acceleration. Data movement between CPU and GPU is facilitated using `#pragma acc enter data` and `#pragma acc exit data` di-

```

1 #pragma acc enter data copyin(chsp[:lenA], sn2d[:lenA], dinj[:lenA], nstep[:lenA],
2   np[:lenA][:lenB], x[:lenA][:lenB][:lenC], yp[:lenA][:lenB][:lenC],
3   vx[:lenA][:lenB][:lenC], vy[:lenA][:lenB][:lenC])
4 {
5     for (isp = 0; isp < nsp; isp++) {
6         ...
7         #pragma acc parallel loop gang worker vector vector_length(128)
8         present(np[:lenA][:lenB], nstep[:lenA], x[:lenA][:lenB][:lenC],
9         vx[:lenA][:lenB][:lenC]) firstprivate(nc,isp,nsp) private(i)
10        for (j = 0; j < nc; j++) {
11            #pragma acc loop
12            for (i = 0; i < np[isp][j]; i++)
13                x[isp][j][i] += nstep[isp] * vx[isp][j][i];
14        }
15        ...
16        #pragma acc parallel loop gang worker vector vector_length(128)
17        present(np[:lenA][:lenB], nstep[:lenA], x[:lenA][:lenB][:lenC],
18        vx[:lenA][:lenB][:lenC]) firstprivate(nc,isp,nsp) private(i)
19        for (j = 0; j < nc; j++) {
20            #pragma acc loop
21            for (i = 0; i < np[isp][j]; i++)
22                x[isp][j][i] += nstep[isp] * vx[isp][j][i];
23        }
24        #pragma acc exit data copyout(x[:lenA][:lenB][:lenC]...)
25    }
26 }

```

Listing 1.4. Simplified C code snippet illustrating the OpenACC (ACC parallel) parallelization with data clauses and array "shape" for GPU porting in the particle mover.

rectives with `copyin` and `copyout` clauses, managing the transfer of relevant arrays (`chsp`, `sn2d`, `dinj`, `nstep`, `np`, `x`, `yp`, `vx`, `vy`).

In the GPU parallel unstructured data region, the `#pragma acc parallel` loop directive is employed to parallelize the outer loop over `j` for components (`nc`). The `present` clause ensures availability of specified arrays, while the `firstprivate` and `private` clauses handle variables `nc`, `isp`, `nsp`, and `i` appropriately.

The loop parallelization strategy uses `gang`, `worker`, and `vector` directives. The `gang` directive divides loop iterations into gangs, potentially assigned to different cores. Within each gang, `worker` directive enables concurrent execution, and `vector` directive subdivides each worker, specifying simultaneous processing with `vector_length(128)` determining vector size.

The innermost loop over `i` is parallelized with `#pragma acc loop` directive, optimizing GPU capacity for parallel computations on inner loops while minimizing data transfer overhead.

The `#pragma acc exit data copyout` directive ensures copied back modified data to the CPU after GPU computations are complete.

3.3 Experimental Setup

In this work, we use the following two systems:

- **Dardel**, an HPE Cray EX supercomputer, features a robust **CPU** partition with 1,270 compute nodes. Each node is equipped with two AMD EPYC™ Zen2 2.25 GHz 64-core processors, 256GB DRAM, and interconnected using an HPE Slingshot network with Dragonfly topology providing 200 GB/s bandwidth. The Lustre file system has a 12 PB capacity, and the operating system is SUSE Linux Enterprise Server 15 SP3. We load GNU compiler suite

option "PrgEnv-gnu" for compiler "gcc v11.2.0" and MPI library, "cray-mpich v8.1.17".

- **NJ**, an HPC system, has an AMD EPYC 7302P 16-Core Processor with 32 CPU cores. It operates on the x86_64 platform, 2 threads per core, and 16 cores per socket, running at a 3.0 GHz base clock. **NJ** also hosts two NVIDIA A100 GPUs with 40 GB HBM2e memory, 6912 Shading Units, 432 Tensor Cores, and 108 SM Count. GPUs have 192 KB L1 Cache per SM, 40 MB L2 Cache, and offer double-precision matrix (FP64) performance of approximately 9.746 TFLOPs. We load CUDA Driver v11.0, NVIDIA HPC SDK v23.7 with compiler version "gcc v12.2.0," and MPI "openmpi v4.1.5".

In this work, we focus on optimizing the particle mover function in BIT1, with a particular emphasis on closely monitoring and analyzing BIT1 performance. As a test case, we simulate neutral particle ionization resulting from interactions with electrons in upcoming magnetic confinement fusion devices like ITER and DEMO. The scenario involves an unbounded unmagnetized plasma consisting of electrons, D^+ ions and D neutrals. Due to ionization, neutral concentration decreases with time according to $\partial n / \partial t = nn_e R$, where n , n_e and R are neutral particles, plasma densities and ionization rate coefficient, respectively. We use a one-dimensional geometry with 100K cells, three plasma species (e electrons, D^+ ions and D neutrals), and 10M particles per cell per species. The total number of particles in the system is 30M. Unless differently specified, we simulate up to 1K time steps. An important point of this test is that it does not use the Field solver and smoother phases (shown in the diagram of Fig. 2).

4 Performance Results

4.1 Hybrid MPI and OpenMP/OpenACC BIT1

Focusing on intra-node testing, an in-depth investigation into how BIT1 performs in terms of "execution time" has been conducted, to explore the advantages of utilizing hybrid approaches on the two HPC systems. The aim was to determine if using both MPI and OpenMP/OpenACC, instead of just MPI, would make a significant difference. Fig. 3 shows executions of hybrid BIT1 total execution vs. optimized mover function using 2 and 16 ranks per node for 1000 times steps on *NJ*. For both 2 ranks and 16 ranks, our hybrid MPI+OpenMP version of BIT1 shows a reduction in both total simulation and mover function time. This suggests that parallelizing BIT1 with OpenMP threads improves performance by enabling multiple threads to work on the problem concurrently. Similar to OpenMP, our hybrid MPI+OpenACC version for multicore CPUs also results in a reduction in both total simulation and mover function time. This demonstrates that BIT1 benefits when used with multicore CPUs, due to better utilization of CPU cores through parallelization.

Investigating the scalability of hybrid BIT1 on CPUs and Fig. 3, it is easy to see that with an increase in MPI ranks from 2 to 16, there is a significant improvement in performance for both total simulation and optimized mover

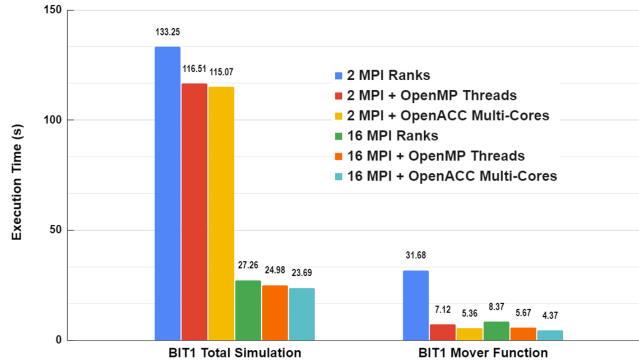


Fig. 3. Hybrid BIT1 total simulation and optimized mover function using 2 and 16 ranks per node on *Dardel* for 1000 times steps.

function execution times. This shows that our hybrid BIT1 scales well. However, to confirm these findings, our second system, *Dardel*, was used for further investigation with a focus on the optimized mover function using OpenMP (since OpenACC was not used).

On *Dardel*, as seen in Fig. 4, the execution time significantly decreases as the number of MPI ranks and OpenMP threads increases, showcasing the potential for efficient parallel executions. For instance, with 128 MPI ranks versus 8 MPI ranks with 16 OpenMP threads, the execution time reduces to 10.55 seconds, a notable improvement from the original 12.72 seconds.

4.2 Accelerating BIT1 with GPUs

The challenge of enhancing the performance of the particle mover function in BIT1 by tapping into the computational power of GPUs has been systematically deliberated, revealing valuable insights. In addressing this challenge, our focus shifts to improving the particle mover function’s **execution time** by offloading it to the GPU using OpenACC and OpenMP. In doing this, we are one step closer for BIT1 being ready for Exascale platforms. To achieve this target, two main strategies provided by OpenACC and OpenMP were investigated: the explicit approach, where data regions for GPU offloading are clearly defined using directives, and the unified memory method, which simplifies memory management by sharing a common space between the CPU and GPU.

BIT1 OpenACC GPU Explicit. Initial work began by using OpenACC’s explicit approach on the GPUs on *NJ* for up to 10 time steps for better visualization of profiling results, particularly focusing on the particle mover function. The profiling results, using NVIDIA Nsight Systems, reveal crucial insights. The CUDA kernel statistics showed that the primary kernel responsible for the particle mover function consumes 99.7% of the GPU execution time, emphasizing its significance in the overall computation. In Fig. 5, the memory operation statistics

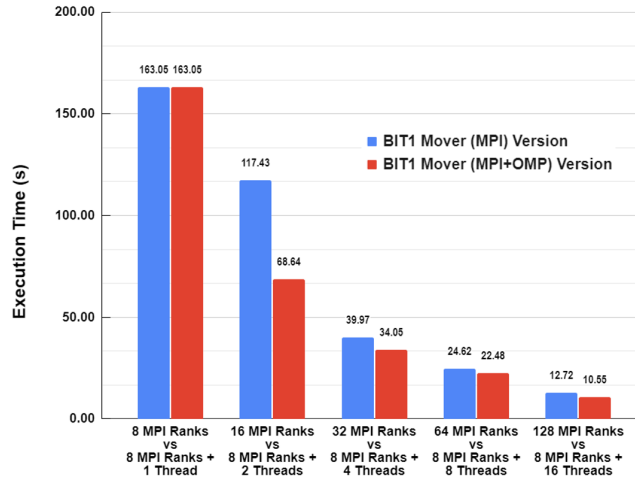


Fig. 4. Optimized mover function - scaling up to 128 MPI ranks on *Dardel* for 1000 times steps.

shed light on the substantial time spent on `memcpy` operations. Specifically, 80% of the GPU time is allocated to copying data from host to device, emphasizing the importance of efficient data transfer strategies.

These findings emphasize the importance of optimizing data movement and kernel execution in the particle mover function. Strategies such as overlapping computation and communication, along with exploring ways to minimize data transfer size, can be instrumental. Additionally, considering the high memory bandwidth of the NVIDIA A100 GPUs on *NJ*, enhancing the efficiency of memory operations becomes pivotal for achieving optimal performance.

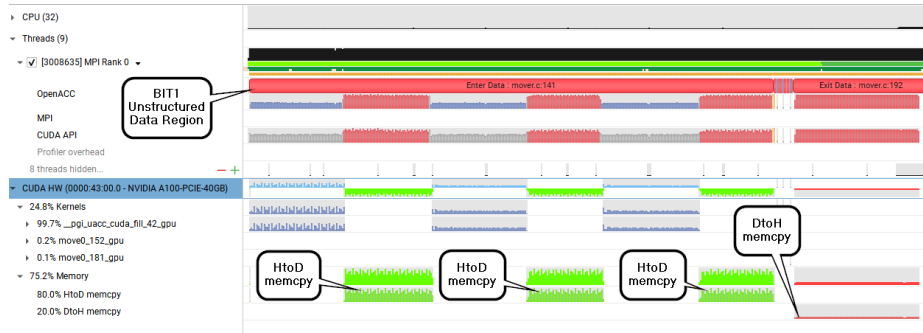


Fig. 5. NVIDIA Nsight Systems OpenACC Explicit View - GPU porting of BIT1 mover function on *NJ* with 1 time step.

BIT1 OpenACC GPU Unified Memory. Next, the particle mover function has been initially investigated on *NJ* using OpenACC Unified Memory. The profiling results revealed critical insights into the dynamics of memory management and kernel execution.

The CUDA kernel statistics showed the primary mover kernel *move0_{152_gpu}* dominated GPU execution time, accounting for 66.7% with a total execution time of 1.623457 seconds. Similarly the second kernel *move0_{181_gpu}* contributed 33.3% with an average execution time of 0.8106292 seconds, highlighting the significance of these kernels in the overall computation.

For hybrid BIT1 data movement and the implementation of a unified memory strategy, the runtime system manages the seamless transfer of data between the host and the device. One key advantage, aside from programmer convenience, is the opportunity for the runtime to automatically detect instances of overlapping computation and communication. Fig. 6 displays NVIDIA Nsight Systems’ view of such overlapping, revealing that the unified memory version exhibited faster overall runtime than explicit copies. This indicates automatic overlapping of computation and communication. However, we observe that BIT1 OpenACC Unified Memory performance is still hindered by substantial data movement between the host and device.

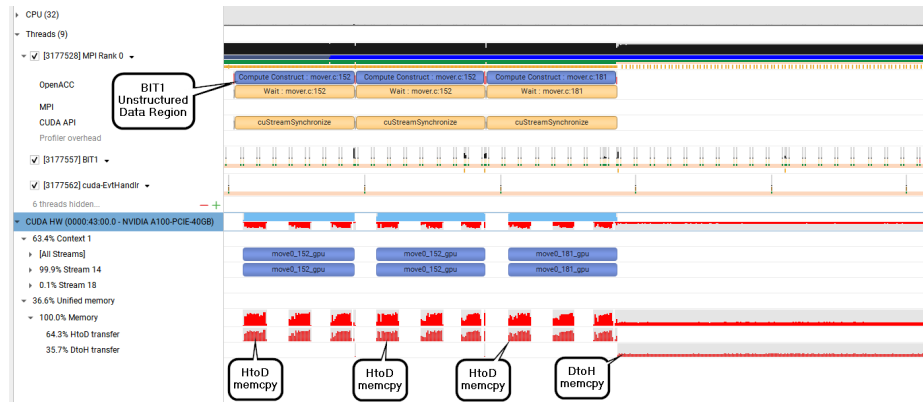


Fig. 6. NVIDIA Nsight Systems OpenACC Unified Memory View - GPU porting of BIT1 mover function on *NJ* with 1 time step.

BIT1 OpenMP GPU Offloading. Due to the observed performance gain with hybrid BIT1 on multicore CPU, an investigation was conducted using the OpenMP target construct for further BIT1 GPU offloading. Fig. 7 and 8 provides performance evaluation insights into GPU utilization compared to the CPU-only baseline with 2 MPI and 16 MPI Ranks. Implementations using OpenMP or OpenACC on GPUs exhibit increased execution times, indicating that parallelization strategies may introduce overhead, potentially outweighing the bene-

fits of parallel processing. Among BIT1 GPU implementations, OpenMP Target with 2 GPUs stands out for delivering a substantial reduction in execution time, demonstrating the performance improvement through concurrent GPU utilization, especially when MPI ranks are assigned to dedicated GPUs.

Yet, with promising results, a critical challenge emerges: data transfer constraints during each PIC cycle. Profiling results from Fig. 6 expose the impact of copying substantial data from the CPU to the GPU at each time step, leading to notable performance bottlenecks. Addressing this challenge involves avoiding large data transfers from the CPU to the GPU at each iteration. The exploration of CUDA streams and particle batch processing with OpenMP Target across Multi-GPUs per node emerges as a promising avenue to streamline data transfer and processing, as also observed in [3].

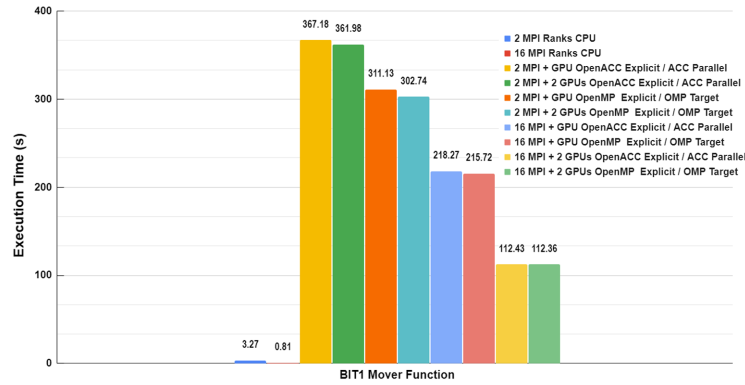


Fig. 7. Optimized mover function execution time(s) on *NJ* for 100 time steps using OpenMP and OpenACC Explicit on NVIDIA GPUs.

5 Related Work

BIT1, an advanced PIC code, is designed to simulate plasma-material interaction [9] and is utilized in various applications, including fusion devices such as tokamaks. BIT1 builds upon the XPDP1 code [11], initially developed by Verboncoeur’s team at Berkeley, and incorporates optimized data layout to efficiently handle collisions [8]. Recently, Williams, Jeremy J., et al. profiled the performance of BIT1, highlighting the particle mover as one of the most computationally intensive parts of the code [13]. Several works have been devoted to hybrid parallelization using MPI and OpenMP for PIC codes, including Smilei [4], iPIC3D [6], and Warp-X [10], to mention a few. In contrast to earlier studies, we employ task-based shared-memory parallelization techniques [1], which are commonly used in linear solvers [5], to specifically address load-imbalance issues

in BIT1’s particle mover [13]. Additionally, BIT1 was ported to NVIDIA GPUs using OpenACC [2]. This approach builds upon previous endeavors, including the successful OpenACC porting of the GTC-P PIC code [12] and iPIC3D [7] to NVIDIA GPUs.

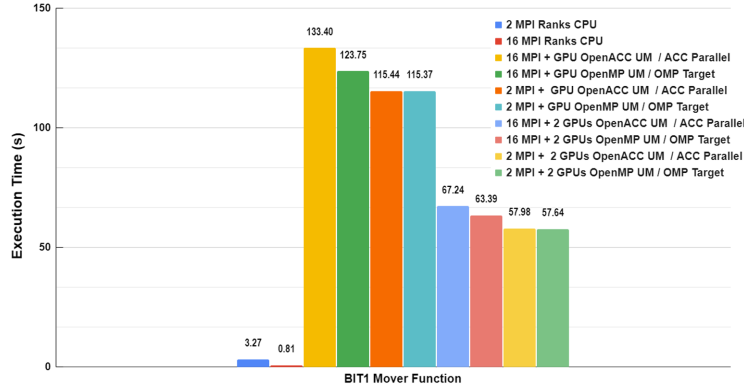


Fig. 8. Optimized mover function execution time(s) on *NJ* for 100 time steps using OpenMP and OpenACC Unified Memory on NVIDIA GPUs.

6 Discussion and Conclusion

Our primary goal was to enhance the particle mover function in BIT1, focusing on node-level efficiency and GPU offloading. Hybrid MPI and OpenMP/OpenACC approaches significantly improved on-node performance, showcasing the potential to leverage multicore CPUs and GPUs efficiently.

Results on multicore CPUs demonstrated the effectiveness of the hybrid approaches. The scalability of the MPI+OpenMP version indicates its potential for large-scale plasma simulations, crucial for efficient use of current supercomputer infrastructure.

GPU porting using OpenMP and OpenACC unveiled challenges and opportunities in tapping into GPU resources for the first time. Emphasizing a balanced hybrid approach for optimal GPU performance, our findings suggest that implementing OpenMP or OpenACC on GPUs may increase execution times, potentially outweighing parallel processing benefits. Notably, the OpenMP Target with 2 GPUs demonstrated a significant reduction in execution time among GPU results, highlighting potential performance improvement through concurrent GPU utilization, especially when dedicated GPUs are assigned to MPI ranks.

Future research can enhance BIT1’s capabilities by fine-tuning GPU optimization and integrating advanced algorithms. Exploring CUDA approaches and

batch processing shows promise in optimizing particle movement, while collaborative efforts with experimental data can bolster simulation reliability.

Hybrid MPI, OpenMP, and OpenACC approaches hold promise for comprehensive parallelization. Exploring synergies between these paradigms ensures BIT1's adaptability to various computing environments.

Acknowledgments. Funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) and Sweden, Finland, Germany, Greece, France, Slovenia, Spain, and Czech Republic under grant agreement No 101093261.

References

1. Ayguadé, E., et al.: The design of openmp tasks. *IEEE Transactions on Parallel and Distributed systems* 20(3), 404–418 (2008)
2. Chandrasekaran, S., et al.: *OpenACC for Programmers: Concepts and Strategies*. Addison-Wesley Professional (2017)
3. Chien, S.W., et al.: sputnic: an implicit particle-in-cell code for multi-gpu systems. In: 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). pp. 149–156. IEEE (2020)
4. Derouillat, J., et al.: Smilei: A collaborative, open-source, multi-purpose particle-in-cell code for plasma simulation. *Computer Physics Communications* 222, 351–373 (2018)
5. Liu, F., et al.: Parallel cholesky factorization for banded matrices using openmp tasks. In: *Euro-Par 2023: Parallel Processing*. pp. 725–739. Springer Nature Switzerland, Cham (2023)
6. Markidis, S., et al.: The epigram project: preparing parallel programming models for exascale. In: *High Performance Computing: ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P³MA, VHPC, WOPSSS, Frankfurt, Germany, June 19–23, 2016, Revised Selected Papers 31*. pp. 56–68. Springer (2016)
7. Peng, I.B., et al.: Acceleration of a particle-in-cell code for space plasma simulations with openacc. In: *EGU General Assembly Conference Abstracts*. p. 1276 (2015)
8. Tskhakaya, D., et al.: Optimization of pic codes by improved memory management. *Journal of Computational Physics* 225(1), 829–839 (2007)
9. Tskhakaya, D., et al.: Pic/mc code bit1 for plasma simulations on hpc. In: 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing. pp. 476–481. IEEE (2010)
10. Vay, J.L., et al.: Warp-x: A new exascale computing platform for beam-plasma simulations. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 909, 476–479 (2018)
11. Verboncoeur, J.P., et al.: Simultaneous potential and circuit solution for 1d bounded plasma particle simulation codes. *Journal of Computational Physics* 104(2), 321–328 (1993)
12. Wei, Y., et al.: Performance and portability studies with openacc accelerated version of gtc-p. In: 2016 17th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT). pp. 13–18. IEEE (2016)
13. Williams, J.J., et al.: Leveraging hpc profiling & tracing tools to understand the performance of particle-in-cell monte carlo simulations. *Euro-Par 2023: Parallel Processing Workshops*, arXiv preprint arXiv:2306.16512 (2023)