

OpenMP offloading and OpenACC programming model approach for object-oriented plasma device algorithms

Ezhilmathi Krishnasamy*, Ivona Vasileska[†], Leon Kos[†], Pascal Bouvry*,

* University of Luxembourg/PCOG, Belval, Luxembourg

[†] University of Ljubljana/LECAD, Ljubljana, Slovenia

ezhilmathi.krishnasamy@uni.lu, ivona.vasileska@lecad.fs.uni-lj.si, leon.kos@lecad.fs.uni-lj.si, pascal.bouvry@uni.lu

Abstract—Plasma physics is becoming more important due to its applications in clean energy production (using fusion technology) and other fields, such as chemical and material science. Even recently, Lawrence Livermore National Laboratory (LLNL) has demonstrated the capability of producing more energy through fusion compared to laser energy. Therefore, in the future, we might need to do more computational simulations for further understanding and explore the advancement in plasma physics. Furthermore, this could be possible with the help of supercomputers. In this work, we parallelise a one-dimensional object-oriented plasma device algorithm, Object Oriented Plasma Device 1d (oopd1), on a multicore CPU and GPU. We use the OpenMP programming model for the CPU version, and for the GPU, we use OpenMP offloading and OpenACC offloading. All of these approaches are compared to each other. Thus, it provides further suitable programming models with parallel capabilities for the existing oopd1 to explore the available parallel architectures.

Keywords—GPU, OpenACC, OpenMP Offloading, Plasma Physics

I. INTRODUCTION

Plasma simulation is the computational modelling of the interaction of charged particles with electric and magnetic fields. The plasma may exhibit complex nonlinear behaviour, and the fields and particles may interact with time-dependent boundary conditions. Fluid codes model the plasma using moments of a distribution function at discrete grid points. In contrast, particle-in-cell (PIC) codes to model the plasma using discrete particles, each representing many charged particles. The particle-in-cell Monte Carlo (PIC-MC) method is a self-consistent kinetic approach capable of predicting the electron and ion energy distribution function (EEDF and IEDF), respectively. Essentially, the PIC simulation employs thousands of simulated particles, known as superparticles, to represent a significantly larger number of real particles (10^{14} - $10^{18} m^{-3}$). In a PIC simulation, the motion of each particle is simulated, and macro-quantities (such as particle density, particle flux, current density, etc.) are calculated from the position and velocity of these particles. The macro force acting on the particles is calculated from the field equations.

The object oriented plasma device 1d (oopd1) is a one-dimensional object-oriented PIC program that pushes particle positions and velocity to advance. The main idea of

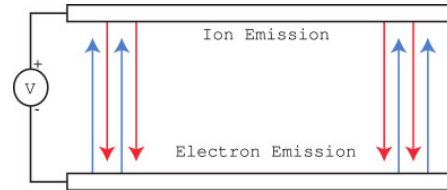


Fig. 1: Plasma one-diode system.

using particles is to represent space collocation, which is much computationally cheaper than the Vlasov/Boltzmann methods [1]. Furthermore, at the same time, Vlasov/Boltzmann represents six-dimensional space and enables arbitrary plasma systems, similar to magnetohydrodynamic methods. It contains a model for planar geometry and includes models for cylindrical and spherical geometries. The oopd1 also allows for different weights of simulation particles and relativistic treatment of electrons. The object-oriented methodology is used for plasma simulations in a physical device with field-emitted electrons in the presence of ion current in a one-dimensional diode. The configuration for a one-dimensional bounded plasma system is shown in Fig. 1.

The algorithm of the oopd1 is common with the other PIC codes [2]–[4]. It contains a particle mover, which updates the position and velocities of the simulated particles according to the famous Newton’s laws of motion, and a field solver, which calculates the fields inside the simulated spatial region at some grid points.

A. Contribution

For the oopd1 code, we have implemented the following approaches:

- OpenMP implementation that uses the available threads in the CPU
- OpenMP offloading directives targeting GPU
- OpenACC directive programming model for GPU enabled parallelisation on GPU
- Performance analysis comparison between all of these implementations

B. previous study

As the oopd1 is based on the PIC code, many studies have been conducted to parallelise similar code on the CPU and GPU [5]–[7]. Moreover, the simplified version of the oopd1 is called SIMPIC, which is also well studied with various parallel implementation approaches [2]–[4]. However, oopd1 involves complex physics and many options to simulate, such as argon, hydrogen, oxygen and chlorine. As far as we know, no study has been explored on parallel implementation of the oopd1 code on GPU.

II. PARALLEL PROGRAMMING

Parallel programming exists to achieve parallel computation on parallel computers. Parallel architecture exists in the form of distributed memory, shared memory and hybrid memory (compute node is connected with accelerators, e.g., Graphics Processing Unit (GPU) and Field Programmable Gate Array (FPGA)). For the past few decades, parallel programming exists, notably Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) programming models. For example, MPI is for parallelisation on distributed memory, and OpenMP is for parallelisation on shared memory parallel architecture.

In particular, MPI and OpenMP have version standards which are being updated regularly; the initial MPI standard was created in 1992 [8]. Similarly, the OpenMP version standard was set in 1997 for the FORTRAN language [9]; since then, OpenMP standards have been regularly updated, and it supports C/C++ as well. But recently, many parallel programming models have been emerging, e.g., Compute Unified Device Architecture (CUDA), Heterogeneous Interface for Portability (HIP), and Open Computing Language (OpenCL). These are being used for explicitly targeting the accelerators, such as GPUs and FPGAs.

As the MPI and OpenMP have existed for many decades, most scientific applications have been written using those programming models. However, to make use of the new parallel heterogeneous architecture (compute nodes with accelerators), scientific codes or programmers have to adapt to vendor-specific (e.g., CUDA and HIP) or low-level programming (OpenCL) to target these parallel heterogeneous architectures.

Although these vendor-specific or low-level programming languages might be employed to port the existing code to accelerators, it might need more effort in terms of time spent on implementing and understanding these programming languages. In order to avoid those obstacles, one could use OpenACC and OpenMP offloading. Both of this OpenACC and OpenMP offloading are very easy to implement with minor changes in the existing code and are directive-based programming languages. This is what we address in this work.

III. TESTING PLATFORM

Presently supercomputers have achieved exascale computing with the help of heterogeneous architecture, e.g.,

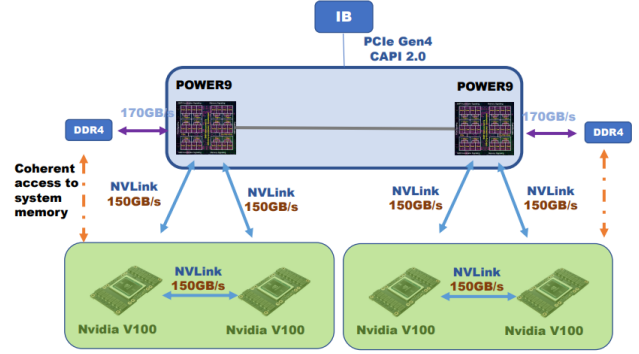


Fig. 2: IBM: Power9 (CPU) with Nvidia Volta V100;source [13].

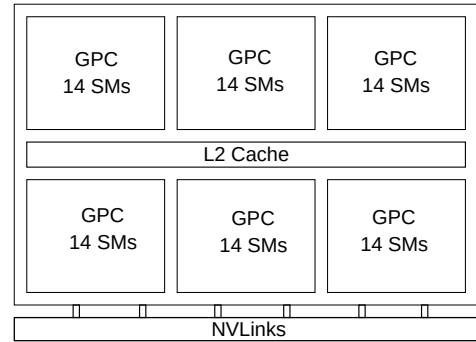


Fig. 3: Nvidia: Volta V100 GPU with 84 streaming multiprocessors.

Frontier [10] has AMD CPU and GPU. Similarly, most supercomputers worldwide have accelerators connected to compute nodes (CPU) such as Nvidia GPU, Intel GPU and FPGA. This work considers IBM POWER9 AC922 at 3.1 GHz with Nvidia Volta V100 GPU. Each compute node has 2 CPU sockets, which have in total of 32 cores. Each core can have up to 4 threads, bringing a total of 128 [11]. Although both OpenACC and OpenMP offloading can be executed on AMD and Nvidia GPU, we consider only Nvidia GPU here.

Figure 2 shows the technical hardware configuration of IBM POWER AC922 with Nvidia GPU compute node. The important perspective of this architecture is that two GPUs are connected to each CPU socket, thus providing a bandwidth of up to 150 GB/s between CPU and GPU. Furthermore, Figure 3 shows the Nvidia Volta architecture GPU that has 6 GPU Processing Cluster (GPC), and each GPC has 14 Streaming Multiprocessors (SMs) with a total of 80 SMs [12]. Nvidia V100 has 2688 double precision cores running at a maximum speed of 1530 MHz [12].

IV. IMPLEMENTATION

This section focuses on the parallel implementations of using OpenMP, OpenACC and OpenMP offloading simulation for argon gas in oopd1.

A. OpenMP

We have introduced the OpenMP directives wherever it is needed. We have identified the parallelisation blocks, where there is no data dependency and data race. This ensures the correctness of the solution and parallelisation in oopd1. We have also introduced `private` and `shared` to identify the variables and avoid computational errors. Listing 1 shows the simple syntax of the OpenMP implementation that we have used in the oopd1 code. We have used the GNU compiler for the compilation.

Listing 1: OpenMP.

```
#pragma omp parallel for
for (int i=0; i<N; i++)
{
    // computation
}
```

B. OpenACC

OpenACC is very similar to OpenMP, which is based on the directive programming model. Where adding the directive clauses above the loops will parallelise the loop. Furthermore, OpenACC has three concepts; they are incremental (original or existing code can be maintained), single source (some code can be executed on different architecture), and low learning curve (easy to learn). Listing 2 shows the pseudo-code implementation and its output, where we can see that the loop is already parallelised with 128 GPU threads. An `acc` will instruct the compiler to execute the code block on the device, and `parallel` will instruct the code block to be executed in parallel on the device. Since the code has 1000 lines and more than 100 files, we opted to use the unified memory option in OpenACC. This option will take care of the memory transfer wherever it is necessary. Even if we do it with explicit memory allocation, we would not expect to get a large difference in performance. Therefore, we have enabled the unified memory option flag `-ta=tesla:managed` during the compilation.

Listing 2: OpenACC.

```
#pragma acc parallel loop
for (int i=0; i<N; i++)
{
    // computation
}

/** compilation output */
Fields::Fields(SpatialRegion *):
    81, Generating NVIDIA GPU code
    83, #pragma acc loop gang, vector(128)
    /* blockIdx.x threadIdx.x */
```

OpenACC is supported by both commercial and open-source compilers, such as HPE, Nvidia HPC SDK, GNU, and OpenARC. However, we chose the Nvidia HPC SDK compiler, and moreover, this is the only option we got from the testing HPC platform. For compilation, we used flags as follows `nvc++ -fast -Minfo=accel -ta=tesla:managed -acc`.

C. OpenMP offloading

The OpenMP API standard started to support the OpenMP offloading from 4.0, and thereafter the standard included more advanced features to support the GPU [14]. Although OpenMP offloading is supported by many compilers, such as AMD, GNU, HPE, IBM, and Intel, we use the Nvidia HPC SDK compiler in this work [15]. Here in OpenMP offloading, `target` will execute the code on the device, `teams` will create teams of thread blocks, and finally, `distribute parallel` will create a number of threads to be executed in parallel on each thread blocks. Listing 3 shows the pseudocode example used in the oopd1 code and the compilation output of how the device code block is parallelised using the default thread block.

Listing 3: OpenMP offloading.

```
#pragma omp target teams distribute parallel for
for (int i=0; i<N; i++)
{
    // computation
}

/** compilation output */
Fields::Fields(SpatialRegion *):
#pragma omp target teams distribute parallel for
    82, Generating "nvkernel__ZN6FieldsC1EP13
    SpatialRegion_F1L82_2" GPU kernel
    85, Loop parallelized across teams and
    threads(128), schedule(static)
```

We used the Nvidia HPC SDK compiler for compilation:
`nvc++ -fast -mp=gpu -gpu=managed
-Minfo=mp, accel.`

V. CODE OPTIMISATION

We have chosen the default thread blocks for both OpenACC and OpenMP directive clauses; however, as a programmer, we can also choose to set the threads blocks in OpenACC and OpenMP to get maximum performance. Table I shows how the CUDA terminologies `thread blocks`, `threads`, and `warps` are interpreted in OpenACC (for both `kernels` and `parallel`) and OpenMP offloading. OpenACC offers two options for creating a parallel code block, using `kernels` and `parallel`; however, we have used `parallel` (explicit and creates just one device kernel) for the entire oopd1 code [16]. According to OpenACC, `kernels` is safer (implicit and also creates more device kernels) to use when a programmer does not know much about data dependency [16], [17]. However, since we knew the data flow in oopd1, we have used `parallel`.

Listing 4 shows how the loop can be optimised to use more thread blocks within the parallel code block. As can be seen in Listing 2, we do not know exactly how many gangs will be created, and for each and every gang, we will have up to 128 threads; this can be seen in Figure 4. However, on the Nvidia device, threads in the thread blocks are executed as warp, and each warp will have up to 32 threads. Each thread block can be executed on the GPU on the SMs; therefore, the equivalent amount of

TABLE I: OpenACC and OpenMP offloading loop mapping in terms of CUDA terminologies [19], [20].

CUDA Terminologies	OpenACC Kernels	OpenACC Parallel	OpenMP Offloading
Thread Blocks	gang	num_gangs	teams
Threads	vector	vector_length	parallel
Warps	worker	num_workers	simd

SMs with threads block will have maximum performance. But this is only possible when we know the problem size; otherwise, it is better to let the compiler choose the default thread blocks for a specific problem. Furthermore, GPU architecture is based on Single Instruction Multiple Threads (SIMT) which execute the threads based on the Single Instruction Multiple Data (SIMD), and it uses a set of 32 threads (warp) [18]. So, warps of 32 threads will be in a queue depending on the SMs resource available. Figure 5 shows these phenomena, where each thread block has just 32 threads and will be executed as one warp on the SM.

Similarly, we have done it in OpenMP offloading as well; that is, setting the thread blocks and the number of threads on each thread block. Listing 5 shows the num_teams and thread_limit; however, we do not see this as a compiler output similar to OpenACC; perhaps using the GNU compiler might give more information for compilation output. We can also see the visualisation phenomena in Figure 5, where threads are grouped as warps and will be executed one by one. Moreover, Nvidia Volta V100 GPU has 32 double-precision floating-point cores (FP64).

Listing 4: OpenACC optimisation.

```
#pragma acc parallel loop
num_gangs(32) vector_length(32)
for (int i=0; i<N; i++)
{
    // computation
}

/** compilation output */
Fields::Fields(SpatialRegion *):
12, Generating NVIDIA GPU code
17, #pragma acc loop gang(32), vector(32)
/* blockIdx.x threadIdx.x */
```

Listing 5: OpenMP offloading optimisation.

```
#pragma omp target teams num_teams(32)
distribute parallel for thread_limit(32)
for (int i=0; i<N; i++)
{
    // computation
}

/** compilation output */
Fields::Fields(SpatialRegion *):
82, #omp target teams distribute parallel for
num_teams(32) thread_limit(32)
82, Generating "nvkernel_ZN6FieldsC1EP13
SpatialRegion_F1L82_2" GPU kernel
85, Loop parallelized across teams and
threads(128), schedule(static)
```

Device Streaming Multiprocessors

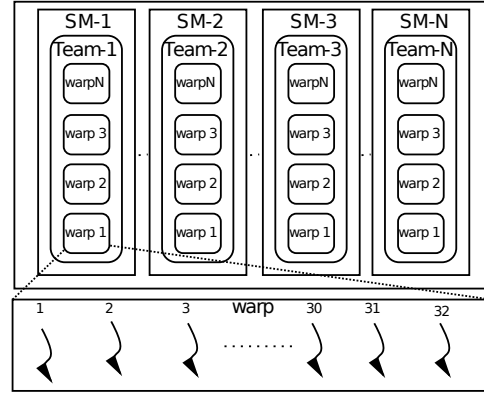


Fig. 4: Schematic workflow of OpenACC and OpenMP offloading threads (default) on the device.

Device Streaming Multiprocessors

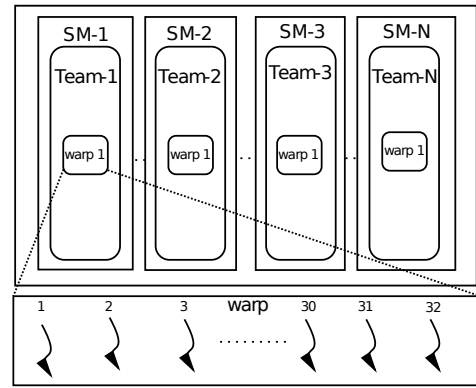


Fig. 5: Schematic workflow of manually setting threads on a device for OpenACC and OpenMP offloading.

VI. DISCUSSION

In oopd1, we have tried the argon gas model for testing with 1000 iterations. Figure 6 shows the time difference between serial and various parallel implementations. Since our main goal is to offload the computational task to GPU, we used all the threads(128) from the compute node for the OpenMP version. As we can notice, OpenACC and OpenMP offloading show similar behaviour and also with optimized thread blocks of these two approaches. This is mainly due to two reasons: the compiler already chooses the optimised thread blocks for the given problem, and the computational task within the device code is not very intensive. However, this phenomenon might look different where the device code needs lots of arithmetic computations, that is, if each warp execution takes more time. In that scenario choosing the thread blocks manually would give little more performance benefit than choosing the default thread block.

At the same time, the OpenMP version with 128 threads does not compete with the other two options, OpenACC and OpenMP offloading; this is mainly because the com-

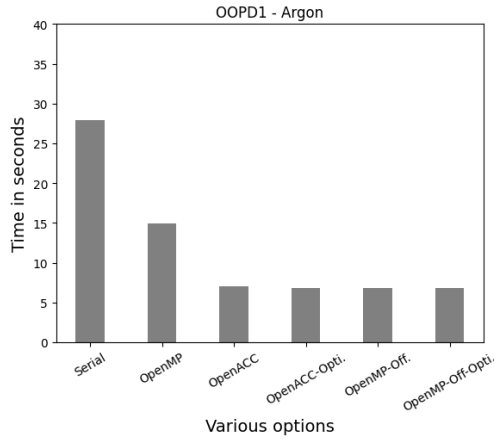


Fig. 6: Schematic workflow of OpenACC threads on the device.

putational domain has more than 128 spatial domains (spatial domain is discretised with 1001). Therefore, the OpenMP version takes a little longer time to complete the task compared to GPU (OpenACC and OpenMP offloading). Because GPU (OpenACC and OpenMP offloading) can have up to 1001 threads, these thread blocks can be executed on the SMs on the GPU. In addition, V100 GPU has up to 84 SMs [12], and each warp can be executed on each SM and adding up to 32 SMs (total, we get 1024 threads (32*32)).

VII. CONCLUSION

We have ported the oopd1 code to GPU using OpenACC and OpenMP offloading. We notice that both OpenACC and OpenMP offloading show good speedup compared to serial and OpenMP versions. Although optimised option thread blocks for both OpenACC and OpenMP offloading do not show any difference due to the problem (argon gas) nature in oopd1; however, it was good to try this option in oopd1 and notice the behaviour.

We would like to continue with MPI implementation with GPUs in the future. At the same time, we also want to study further OpenMP offloading on the AMD GPU and, if possible, with various compilers for OpenMP offloading. Moreover, we do not see any huge difference between OpenACC and OpenMP offloading; therefore, in the future, we would like to focus on OpenMP offloading, considering the code can be easily run on both Nvidia and AMD GPUs. Finally, we would also like to try various physics parameters in oopd1 and check if the code can be further optimised for the GPU (including heterogeneous computing). From this work, we believe that in the future, a directive-based programming model for the device would completely eliminate the rewriting of any existing scientific code for GPU.

ACKNOWLEDGMENT

We are also grateful to CINECA for letting us to use their GPU cluster computational resources. The first author

is funded by the EuroCC, and the second author is funded by PRACE.

REFERENCES

- [1] T. M. Tyranowski and M. Kraus, "Symplectic model reduction methods for the vlasov equation," *Contributions to Plasma Physics*, p. e202200046, 2022.
- [2] I. Vasileska, P. Tomšič, and L. Kos, "Modernization of the pic codes for exascale plasma simulation," in *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE, 2020, pp. 209–213.
- [3] I. Vasileska, L. Bogdanović, and L. Kos, "Particle-in-cell code for gpu systems," in *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE, 2021, pp. 291–294.
- [4] E. Krishnasamy, I. Vasileska, L. Kos, and P. Bouvry, "Openmp offloading and openacc programming model approach for object-oriented plasma device algorithms," in *2023 The 8th International Conference on Computer and Communication Systems*. IEEE, 2023.
- [5] Z. Juhasz, J. Ďurian, A. Derzsi, Š. Matejčík, Z. Donkó, and P. Hartmann, "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, p. 107913, 2021.
- [6] S. W. Chien, J. Nylund, G. Bengtsson, I. B. Peng, A. Podobas, and S. Markidis, "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)*. IEEE, 2020, pp. 149–156.
- [7] H. Shah, S. Kamaria, R. Markandeya, M. Shah, and B. Chaudhury, "A novel implementation of 2d3v particle-in-cell (pic) algorithm for kepler gpu architecture," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE, 2017, pp. 378–387.
- [8] D. W. Walker and J. J. Dongarra, "Mpi: a standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.
- [9] C. university, "Openmp history," 2022. [Online]. Available: <https://cvw.cac.cornell.edu/openmp/openmp-history-and-evolution>
- [10] Frontier, "Frontier," 2022. [Online]. Available: <https://www.olcf.ornl.gov/frontier/>
- [11] CINECA, "Marconi100 userguide," 2023. [Online]. Available: <https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.2%3A+MARCONI100+UserGuide>
- [12] Nvidia, "Volta gpu white paper," 2023. [Online]. Available: <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>
- [13] Nvidia and IBM, "Ibm ac922," 2023. [Online]. Available: <https://on-demand.gputechconf.com/gtc/2018/presentation/s8309-nvidia-v100-performance-characteristics-on-ibm-power-9-system-and-hpc-application-performance.pdf>
- [14] OpenMP, "Openmp 5.1 api syntax reference guide," 2020. [Online]. Available: <https://www.openmp.org/wp-content/uploads/OpenMPRefCard-5.1-web.pdf>
- [15] OpenMP.org, "Openmp compilers & tools," 2023. [Online]. Available: <https://www.openmp.org/resources/openmp-compilers-tools/>
- [16] S. Chen, "Introduction to openacc," 2017. [Online]. Available: <https://www.bu.edu/tech/files/2017/04/OpenACC-2017Spring.pdf>
- [17] R. Farber, *Parallel programming with OpenACC*. Newnes, 2016.
- [18] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu, "Efficient compilation of cuda kernels for high-performance computing on fpgas," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, pp. 1–26, 2013.
- [19] C. Shen, X. Tian, D. Khaldi, and B. Chapman, "Assessing one-to-one parallelism levels mapping for openmp offloading to gpus," in *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*, 2017, pp. 68–73.
- [20] J. Bispo, J. G. Barbosa, P. F. Silva, C. Morales, M. Mylykoski, P. Ojeda-May, M. Bialczak, M. Uchronski, A. Włodarczyk, P. Wauligmann *et al.*, "Best practice guide: Modern accelerators," Technical report, PRACE aisbl, June 2021. URL: <https://prace-ri.eu/training...>, Tech. Rep., 2021.