*Bull. Nov. Comp. Center*, Num. Anal., 18 (2016), 17–28


# The portable and scalable kinetic plasma simulation code for hybrid supercomputers*

E.A. Genrikh, A.V. Snytnikov

**Abstract.** Today the method for the development of portable numerical simulation programs is very important because of the two main reasons. The first reason is the diversity of supercomputer architectures in Top500 and the second one is a demand for using the most powerful computers to simulate, for example, plasma. The method presented is based on minimizing the number of code fragments that depend on the architecture and, also, on the use of function variables. The use of the method is shown on the example of the program for plasma simulation by the Particle-In-Cell method. The program was ported from the GPU to Intel Xeon Phi. The porting was done just by re-compilation because of the method described in the paper.

**Keywords:** portability, hybrid supercomputers, GPU, function variables, computational plasma physics.

## 1. A kinetic plasma model

The objective of this section is to display the equations that are solved in all plasma physics problems. They are the equations that the template implementation of the PIC method must be suitable for.

The mathematical model employed for the solution of the problem of beam relaxation in plasma consists of the Vlasov equations for ion and electron components of plasma and of Maxwell's equation system. These equations in the usual notation have the following form:

$$\frac{\partial f_{i,e}}{\partial t} + \vec{v}\frac{\partial f_{i,e}}{\partial \vec{r}} + \vec{F}_{i,e}\frac{\partial f_{i,e}}{\partial \vec{p}} = 0, \quad \vec{F}_{i,e} = q_{i,e}\Big(\vec{E} + \frac{1}{c}[\vec{v}, \vec{B}]\Big),$$

$$\operatorname{rot}\vec{B} = \frac{4\pi}{c}\vec{j} + \frac{1}{c}\frac{\partial \vec{E}}{\partial t}, \quad \operatorname{div}\vec{B} = 0,$$

$$\operatorname{rot}\vec{E} = -\frac{1}{c}\frac{\partial \vec{B}}{\partial t}, \quad \operatorname{div}\vec{E} = 4\pi\rho.$$

In the present paper this equation system is solved by the method described in [2]. All the equations will further be given in the dimensionless form.

*Supported by the Russian Science Foundation Project 16-11-10028. Development of the code was supported by the RFBR under Grants 14-07-00241,16-01-00209,16-07-00434, and 14-01-31088). Simulations are performed at the Siberian Supercomputer Center SB RAS.

The following basic quantities are used for the transition to the dimensionless form:

- characteristic velocity is the velocity of light $\tilde{v} = c = 3 \cdot 10^{10}$ cm/s;
- characteristic plasma density $\tilde{n} = 10^{14}$ cm$^{-3}$;
- characteristic time $\tilde{t}$ is the plasma period (a value inverse to the electron plasma frequency) $\tilde{t} = \omega_p^{-1} = \left(\frac{4\pi n_0 e^2}{m_e}\right)^{-0.5} = 5.3 \cdot 10^{-12}$ s.

The Vlasov equations are solved by the PIC method. This method implies the solution of the equation of motion for model particles:

$$\frac{\partial \vec{p}_e}{\partial t} = -(\vec{E} + [\vec{v}_e, \vec{B}]), \quad \frac{\partial \vec{p}_i}{\partial t} = \kappa(\vec{E} + [\vec{v}_i, \vec{B}]), \quad \frac{\partial \vec{r}_{i,e}}{\partial t} = \vec{v}_{i,e},$$
$$\kappa = \frac{m_e}{m_i}, \quad \vec{p}_{i,e} = \gamma \vec{v}_{i,e}, \quad \gamma^{-1} = \sqrt{1 - v^2}.$$

The quantities with subscripts $i$ and $e$ are related to ions and electrons, respectively.

The leapfrog scheme is employed to solve these equations:

$$\frac{\vec{p}_{i,e}^{\,m+1/2} - \vec{p}_{i,e}^{\,m-1/2}}{\tau} = q_i\left(\vec{E}^m + \left[\frac{\vec{v}_{i,e}^{\,m+1/2} - \vec{v}_{i,e}^{\,m-1/2}}{2}, \vec{B}^m\right]\right),$$
$$\frac{\vec{r}_{i,e}^{\,m+1} - \vec{r}_{i,e}^{\,m}}{\tau} = \vec{v}_{i,e}^{\,m+1/2},$$

where $\tau$ is the timestep.

The scheme proposed by Langdon and Lasinski is used to obtain the values of electric and magnetic fields. The scheme employs the finite-difference form of the Faradey and Ampere laws. A detailed description of the scheme can be found in [2]. The scheme gives the second order of approximation with respect to space and time.

## 2. Scalability

### 2.1. Parallel implementation

The program was parallelized by the domain decomposition method. The computational domain is divided into parts along the direction orthogonal to the direction of the beam (along the axis $Y$, the beam moving along the axis $X$). The computational grid in the whole domain is divided into equal parts (subdomains) along the axis $Y$. Each subdomain is assigned to a group of processors (in the case of a multicore system a single core is called a processor, since no hybrid parallelization like MPI+OpenMP is employed, just a mere MPI). Furthermore, the superparticles of each subdomain are

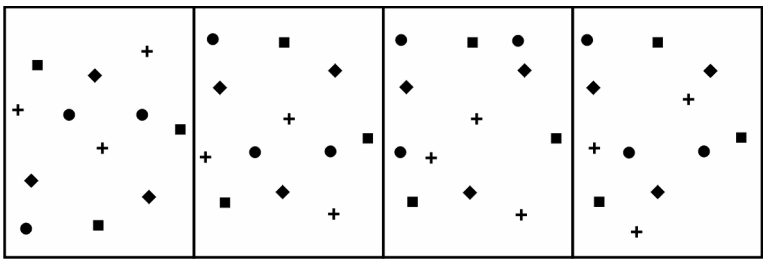

**Figure 1.** The scheme of domain decomposition. The computational domain is divided into 4 subdomains. The superparticles of each subdomain are distributed among four processors uniformly with no regard to their position. Different symbols (a circle, a square, a diamond, a star) denote superparticles belonging to different processors in the same subdomain

uniformly distributed among processors of the group with no regard to their position, as is shown in Figure 1.

Every processor in the group solves the Maxwell's equations in the whole subdomain, and exchanges boundary values of the fields with processors assigned to the adjacent subdomains. Then the the equations of motion for the superparticles are solved, and the 3D matrix of the current density and the charge density are evaluated by each processor. However, since the processor has only a part of the superparticles located inside the subdomain, it is necessary to sum the matrices through all the processors of the group to obtain the whole current density matrix in the subdomain. The interprocessor data exchange is performed by the MPI subroutines.

### 2.2. Parallelization efficiency

The parallel program was primarily developed for the simulation of the beam interaction with plasma on large computational grids and with large numbers of superparticles. That is why the parallelization efficiency was computed in the following way:

$$k = \frac{T_2}{T_1} \cdot \frac{N_1}{N_2} \cdot \frac{S_2}{S_1} \cdot 100\ \%. \tag{1}$$

Here $T_1$ is the computation time with $N_1$ processors, $T_2$ is the computation time with $N_2$ processors, $S$ is the characteristic size of the problem in each case. Here the characteristic size is the grid size along the axis $X$. In this section, the characteristic size $S$ is proportional to the number of processors $N$. This means that the workload of a single processor is constant. The purpose of such a definition of efficiency is to find out what the communication overhead is when the number of processors is increased with a constant workload for each processor. In the ideal case, the computation

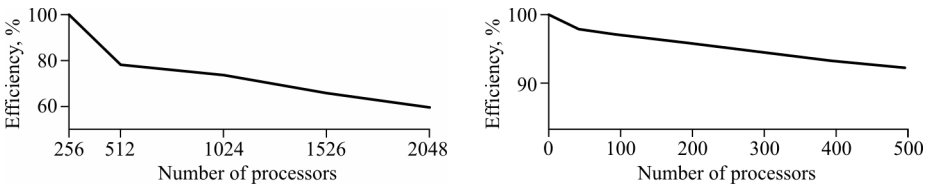


**Figure 2.** Parallelization efficiency measured with MVS-100K cluster (left) and the "Lomonosov" supercomputer at the Moscow State University (right). The grid size along the axes $Y$ and $Z$ is 64 nodes, the grid size along the axis $X$ is equal to the number of processors, 150 superparticles per cell are used in all the cases

time must remain the same (the ideal $k = 100$ %). In the computations dealing with the efficiency evaluation only the grid size of the axis $X$ was increased, all the other parameters remaining unchanged. The results are shown in Figure 2.

## 3. The GPU implementation

The implementation of the above PIC algorithm for the GPUs is quite standard. The field evaluation method is ported to the GPU almost without any change. The computation speed is high enough even without optimization. The field arrays are stored in the GPU global memory.

The bottleneck of the PIC codes is the particle push. With the CPUs it takes up to 90 % of runtime. So, first the particles are distributed among cells. This step only reduces the push time half as much as the CPUs. With the GPUs it is even more important since this enables the use of texture memory (the texture memory is limited and the whole particle array will never fit). The second step is keeping the field values related to a cell (and, also, to the adjacent cells) in a cell itself. This is important since each particle needs 6 field values and writes 12 current values into the grid nodes, and now this all is done within a small amount of memory (a cell) without addressing the global field or the current arrays that contain the whole domain. Then the evaluated currents from all cells are added to the global current array.

This gives the speedup of about 40 for Kepler K40 as compared to 4 Xeon cores.

## 4. Portability

### 4.1. Statement of the problem

It is necessary to design a tool for porting the code among the most common types of supercomputers:

1. Clusters based on the Nvidia Kepler computation accelerators.

2. Clusters based on Intel Xeon Phi computation accelerators.
3. Clusters based on Intel Xeon processors as well as AMD Opteron, Fujitsu SPARC64, etc.

In the present paper we consider porting the code from the GPU to Intel Xeon Phi (not in the opposite direction!). Also, the optimization for both architectures is not considered.

The main difficulties of the code porting from CUDA [9, 10, 11] to the MIC architecture are the following:

1. Compilation of the CUDA kernels and, particularly, the CUDA kernel calls without Nvidia compiler.
2. Skipping the function calls used to copy the data among different memory types in the CUDA.
3. Definition of data types and keywords used in the CUDA extension of C/C++ language.

In order to provide the portability of the CUDA-specific code spots (CUDA kernels, CUDA API calls), such code spots should be:

- reduced to a minimum,
- wrapped into functions,
- brought out into a standalone external library.

The CUDA-specific code spots are reduced to a minimum in the following way within the simulation code under consideration (Particle-In-Cell plasma simulation [14, 15]). The code has from 15 to 20 small functions used to process grid nodes, model particles, and boundaries of the computational domain.

All these functions are implemented as the CUDA kernels. This means that these functions cannot be compiled with a compiler provided by Intel, etc. The main idea of the method proposed — there must be only one such a non-portable spot in the code.

To conclude of the problem statement it should be explained why the multi-architecture tools like OpenCL are not used to develop portable programs. There are two reasons: first, the OpenCL support for MIC was declared but has not been implemented yet, and second, CUDA provides better performance as compared to OpenCL or OpenACC.

### 4.2. Description of the method

**The universal launcher.** In order to implement the above-mentioned main idea of the method (to provide one non-portable spot in the code), the universal launcher function is proposed:

```
int Kernel_Launcher(
    Cell<Particle> **cells, KernelParams *params,
    unsigned int grid_size_x, unsigned int grid_size_y,
    unsigned int grid_size_z,
    unsigned int block_size_x, unsigned int block_size_y,
    unsigned int block_size_z,
    int shmem_size, SingleNodeFunctionType h_snf)
{
    struct timeval tv1,tv2;
#ifdef __CUDACC__
    dim3 blocks(grid_size_x, grid_size_y, grid_size_z),
    threads(block_size_x, block_size_y, block_size_z);

    gettimeofday(&tv1, NULL);
    GPU_Universal_Kernel<<<blocks, threads, shmem_size>>>
        (cells, params, h_snf);

    DeviceSynchronize();
    gettimeofday(&tv2, NULL);
#else
    char hostname[1000];
    gethostname(hostname, 1000);

    gettimeofday(&tv1, NULL);

    omp_set_num_threads(OMP_NUM_THREADS);

#pragma omp parallel for
    for(int i = 0; i < grid_size_x; i++)
    {
        // ....
            h_snf(cells, params, i, j, k, i1, j1, k1);
        // ....
```

The computational functions are given as the input parameter of the universal launcher function. If the code is compiled by the CUDA C/C++ compiler and is being executed by a GPU-equipped computer with the GPU, then the function parameter is passed to `GPU_Universal_Kernel` and is launched by a GPU inside a kernel. On the other hand, if the code is being compiled by a non-CUDA compiler (GNU C/C++ compiler, Intel C compiler etc.) and is being executed by the Intel Xeon Phi in the native mode (or, simply, by a multicore processor with openMP multithreading), then the function passed as parameter is just called inside a loop. The loop is a 6-times nested loop corresponding to the dimensionality of the CUDA grid of the thread blocks (three-dimension as for the grid and three more for the block of threads).

It is necessary to use function variables (function parameters) since the support of the object-oriented tools of the C++ is limited with CUDA. Due to this reason it is not possible to make computational functions be virtual class members and call a proper function depending on the architecture.

**The unified data type for computational functions.** All the computational functions that where previously launched as Cuda kernels must be wrapped into functions with unified return value type and unified parameters:

```
typedef void (*SingleNodeFunctionType)(GPUCell<Particle> **cells,
    KernelParams *params,
    unsigned int bk_nx, unsigned int bk_ny, unsigned int bk_nz,
    unsigned int nx, unsigned int ny, unsigned int nz);
```

Furthermore, it is necessary to deal with the CUDA C extensions, e.g. special data types like `int3`, `double3`, `dim3`, etc. They could be either redefined, or if possible, they can be defined by the direct inclusion of the cuda.h header. The special keywords like `__global__`, `__device__`, etc. could be masked with a preprocessor.

The only difficulty left is the CUDA API functions — copying from one memory type to another, error handling, etc. These functions must not be called directly from CUDA API, but through the wrapper functions from a standalone header.

The use of the unified data type is shown with an example of the function used to evaluate the electric field from the GPUPlasma class. The listing below shows the original CUDA kernel `GPU_eme` used to evaluate the field for a single node of the computational mesh. One should notice that the real computation is done by the `emeElement` function. The kernel just defines the node by means of the CUDA inner indices, passes the parameters (mesh steps, time steps, electric current values according to the numerical scheme described in [15]) and calls `emeElement`:

```
template <template <class Particle> class Cell>
__global__ void GPU_eme(Cell<Particle> **cells,
    int i_s, int l_s, int k_s,
    double *E, double *H1, double *H2, double *J,
    double c1, double c2, double tau,
    int dx1, int dy1, int dz1,
    int dx2 ,int dy2, int dz2)
{
    unsigned int nx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int ny = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int nz = blockIdx.z * blockDim.z + threadIdx.z;
    Cell<Particle> *c0 = cells[0];
```

```
    emeElement(c0, i_s+nx, l_s+ny, k_s+nz, E, H1, H2,
        J, c1, c2, tau, dx1, dy1, dz1, dx2, dy2, dz2);
}
```

**Passing parameters of computational functions.** The `emeElement` function has parameters. Moreover, it is clear that all the computational functions may have different parameter sets. In order to call all these functions in the same way, and to pass all the functions through a parameter of `SingleNodeFunctionType` type, the special `KernelParams` data type structure was introduced. It includes all the possible parameter sets for all the computational functions. The resulting structure is not so big because within the implemented computational algorithm many functions have overlapping parameter sets.

An example of the adjustment of `GPU_eme` function to the unified function data type is shown below. The most important difference from the original function is that the coordinates of the node `nx`, `ny`, `nz` being processed are obtained from the function parameters, but not from the CUDA inner variables `blockIdx`, `blockDim`, `threadIdx`:

```
template <template <class Particle> class Cell>
__device__ void GPU_eme_SingleNode(Cell<Particle>  **cells,
    KernelParams *params,
    unsigned int bk_nx, unsigned int bk_ny, unsigned int bk_nz,
    unsigned int tnx, unsigned int tny, unsigned int tnz)
{
    unsigned int nx = bk_nx * params->blockDim_x + tnx;
    unsigned int ny = bk_ny * params->blockDim_y + tny;
    unsigned int nz = bk_nz * params->blockDim_z + tnz;
    Cell<Particle> *c0 = cells[0];

    emeElement(c0, params->i_s+nx, params->l_s+ny, params->k_s+nz,
        params->E, params->H1, params->H2, params->J,
        params->c1, params->c2, params->tau,
        params->dx1, params->dy1, params->dz1,
        params->dx2, params->dy2, params->dz2);
}
```

Therefore, before each computational function call (that was previously the CUDa kernel launch), the necessary fields of the `KernelParams` structure are filled in, and the structure is copied to the device memory.

The universal kernel launcher is then called and takes the array of all the cells, `KernelParams` structure, the dimensions of the grid and block, and the computational function itself.

Finally, it is important to answer on the questions: what is a difference between the CUDA and the MIC, and what is the difference between the

corresponding optimization strategies? The main issue in both cases is the data locality. For plasma simulation this means that all the model particles in each cell must be placed closely in the memory of a computational device, either it is a general purpose processor or an accelerator of any type. It must be noted that for the Particle-in-Cell method the data locality affects the performance much more that architecture dependent optimizations.

Further, one of the most important tools to improve the performance for the Intel Xeon Phi is vectorization. Preparing the source code for vectorization may also increase the GPU performance.

In addition, the substitution of the CUDA kernel calls by means of the loops with OpenMP pragmas requires complementary tuning (meaning the number of OpenMP threads, the number of loops involved in the OpenMP section, etc.) for attaining a higher performance. This tuning is a priori not clear, but it can be introduced into the code without loss in the CUDA performance.

In brief, optimization is still problematic, but the CUDA and the MIC optimization do not contradict each other.

### 4.3. The main principles of the portability method

How to write the code to make it easily portable?

1. Make all the code fragments processing mesh nodes, particles, basis elements eigenfunctions, etc. as small functions with a unified return value type and a unified parameter set.
2. Call these not from the CUDA kernel or from the OpenMP-parallelized loops, but from the universal launcher function.
3. The API function that depends on the architecture, like memory copying or error handling, should be called through the substitution library (we use archAPI.h).

## 5. The plasma simulation results

In order to simulate the interaction of an electron beam with plasma, the following values of the main physical parameters were set:

- electron temperature of 1 KeV,
- the mass of ion 1836 electron masses (hydrogene ions),
- plasma density of $10^{17}$ cm$^{-3}$,
- the ratio of the beam density to the plasma density of $10^{-3}$,
- beam energy of 1 MeV,

- the size of the domain $L_X = 0.065$ cm and $L_Y = L_Z = 0.008$ cm,
- the grid size of $512 \times 64 \times 64$ nodes, 150 superparticles per cell.

Density modulation was observed in the computational experiments. The amplitude of the modulation is 220 % of the initial value of density. The modulation in this case means the occurrence of regions with a very high or a very low density in the previously uniform-density plasma as is shown in Figure 3a. It is seen that the density becomes non-uniform not only along the direction of the beam (the axis $X$), but also along the axis $Y$. Thus, the density is modulated not along $X$, that seems quite natural, but also along $Y$ and $Z$. This corresponds the physics of the process well, because it is known that the waves propagating in plasma due to the beam relaxation have all the three components of the wave vector as nonzeros.

Moreover, it was found out that the movement of beam electrons becomes eddy as a result of the beam interaction with plasma. At the initial moment of time all the beam electrons have the same velocity strictly along the axis $X$. This results in the eddy structure of the electron heat flux

$$q(x, y) = |T_e(x, y, L_Z/2)\vec{v_e}(x, y, L_Z/2)|$$

The electron heat flux also gains modulations along X and Y. Moreover, there are regions with a very low value of the electron heat flux as is shown in Figure 3b (less than 1 % of the initial electron heat flux). This means that only in small and isolated regions the value of electron heat flux is close to the initial value, but, generally, in the computational domain the electron heat flux is very low. Thus, the domain as a whole has a very low heat conductivity after the beam relaxation has occurred.

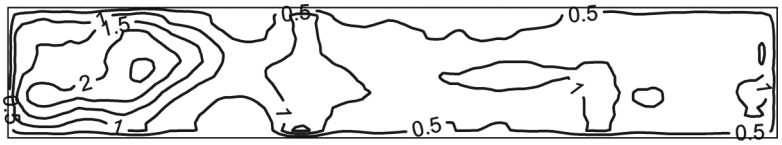


**Figure 3.** Electron density (a) and heat flux (b) contours in the plane $XY$, $z = L_Z/2$, the time instance is $t = 91.7$ in terms of the plasma period. The density and heat flux values are given in terms of their initial values

## References

[1] Annenkov V.V., Timofeev I.V., Volchok E.P. Simulations of electromagnetic emissions produced in a thin plasma by a continuously injected electron beam. — http://arxiv.org/abs/1512.07167.

[2] Vshivkov V.A., Grigoryev Yu.N., Fedoruk M.P. Numerical Particle-in-Cell Methods. Theory and applications. — Utrecht: VSP BV, 2002.

[3] Lotov K.V., Timofeev I.V., Mesyats E.A., Snytnikov A.V., Vshivkov V.A. Note on quantitatively correct simulations of the kinetic beam-plasma instability // Physics of Plasmas. — 2015. — Vol. 22, Iss. 2. — P. 024502. — http://dx.doi.org/10.1063/1.4907223.

[4] Rosales C. Porting to the intel xeon phi: Opportunities and challenges // 2013 Extreme Scaling Workshop. — 2013. — http://ieeexplore.ieee.org/document/6805036/?part=1.

[5] Nakashima H. Manycore challenge in particle-in-cell simulation: How to exploit 1 TFlops peak performance for simulation codes with irregular computation // Computers & Electrical Engineering. — 2015. — Vol. 46. — P. 81–94.

[6] Lyakh D.I. An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and Nvidia Tesla GPU // Computer Physics Communications. — 2015. — Vol. 189. — P. 84–91.

[7] Liu T., Xu X.G., Carothers C.D. Comparison of two accelerators for Monte Carlo radiation transport calculations, Nvidia Tesla M2090 GPU and Intel Xeon Phi 5110p coprocessor: A case study for X-ray CT imaging dose calculation // Annals of Nuclear Energy. — 2015. — Vol. 82. — P. 230–239.

[8] Bernaschi M., Bisson M., Salvadore F. Multi-kepler GPU vs. multi-intel MIC for spin systems simulations // Computer Physics Communications. — 2014. — Vol. 185, Iss. 10. — P. 2495–2503.

[9] Nvidia Cuda home page. — http://www.nvidia.ru/object/cuda-parallel-computing-ru.html.

[10] Kharlamov A.A., Boreskov A.V. The Basis of Using the CUDA-tecnology. — Moscow: DMK-Press, 2010.

[11] Sanders E.K.J. The CUDA-technology on Examples. — Moscow: DMK-Press, 2011.

[12] Morgan T.P. Intel slaps xeon phi brand on mic coprocessors. — http://www.theregister.co.uk/2012/06/18/intel_mic_xeon_phi_cray/.

[13] Fang J., Sips H., Zhang L., et al. Test-driving Intel Xeon Phi // Proc. 5th ACM/SPEC International Conference on Performance Engineering. ICPE'14. — New York, USA, 2014. — P. 137–148.

[14] Glinskiy B., Kulikov I., Snytnikov A., et al. Co-design of parallel numerical methods for plasma physics and astrophysics // Supercomputing Frontiers and Innovations. — 2014. — Vol. 1, No. 3. — P. 88–98. — http://superfri.org/superfri/article/view/26.

[15] Dudnikova G.I., Vshivkov V.A., Vshivkov K.V. Algorithms of solving the problem of interaction of a lase impulse with plasma // Vychislitelnye Teknologii. — 2001. — Vol. 6, No. 2. — P. 47–63 (In Russian).