JOINT INSTITUTE FOR NUCLEAR RESEARCH
Veksler and Baldin laboratory of High Energy Physics

FINAL REPORT ON THE SUMMER STUDENT PROGRAM

Realization of support of calculations on the GPU by a particles-in-cell method for modeling of electron beam ion sources

Supervisor:
Alexey Boytsov

Student:
Yaroslav Zolotukhin, Saint-Petersburg State Electrotechnical University 'LETI'

Participation period:
July 02 – August 15

Dubna, 2018

ABSTRACT

Sources of high-charge electron-beam ions are used for the primary formation and subsequent injection of an ion beam into high-energy accelerators. Modeling of particle dynamics in these devices is often performed by the so-called particle-in-cell method [1,2], which requires considerable computing power. For carrying out calculations for a reasonable time, parallel programming technologies, including graphic accelerators, are required. A review of existing parallel realizations of the particle-in-cell method is carried out. In a freeware program Ef [3], created to simulate the dynamics of low-energy beams, support for calculations on a single graphics accelerator is realized. Some parallel programming technologies are compared: Python3 / Numba and Python3 / PyCUDA - in terms of the optimal relationship between the speed of computation and the ease of support and code development.

INTRODUCTION

Particle-In-Cell (PIC) is a method used to model the motion of charged particles or plasma. This method, applicable to the study of phenomena such as the propagation of the solar wind, or the analysis of the advantages of electric pushers. These discharges are characterized by low plasma density. At low density, the plasma behaves more like a complex of discrete particles than one continuous liquid. Plasma high-density models are modeled using the expansion of computational fluid dynamics in an electromagnet, magnetic hydrodynamics.

Modeling of plasma is complicated by the presence of external and self-induced electromagnetic fields, particle-particle interactions, the presence of solid objects and various characteristic time scales on which ions and electrons are propagated. To speed up the calculation, simplification of assumptions is usually made in accordance with the existing problem. Here we assume that the current generated by the plasma is sufficiently low, so that the self-induced magnetic field can be ignored. This is an admissible assumption for the class of problems that we have here. Thus, the set of underlying Maxwell equations is reduced, and we get an electrostatic PIC code. In addition, we assume that the electrons follow the Boltzmann relation. Then the simulation consists only of heavy particles, ions and neutrals. This simplification has a huge impact on the computational speed, since time integration can be performed in a much larger ion time scale. Finally, we assume that the gas densities are sufficiently small, so that collisions of the particles are negligible [4].

However, the calculation of this algorithm remains time-consuming for direct calculations on the CPU. The purpose of this summer program was to implement this algorithm on the GPU, and compare them for such indicators as:

- The speed of computation and code
- The ease of support code
- The ease of development code

For research, such programming technologies as:

- Python3/Numba
- Python3/PyCUDA

This report describes the method of particles-in-cell and lists the advantages and disadvantages of each programming technology on the GPU and a lively program during the summer program at JINR.

ENTERTAINMENT PROGRAMM IN JINR

In addition to the main task in summer practice, we also had small excursions to the Veksler and Baldin Laboratory of High Energy Physics and Flerov Laboratory of Nuclear Reactions.

The VBLHEP has prepared and is implementing a project to create an accelerator complex, including a modernized Nuclotron-M accelerator, a booster and the heavy nuclei collider NICA. For the experiments, two detector systems are being developed and are being created: MPD (multifunctional detector) and SPD (detector for experiments with polarized beams), and BM@N (baryon matter research at the Nuclotron) for research on extracted Nuclotron beams.

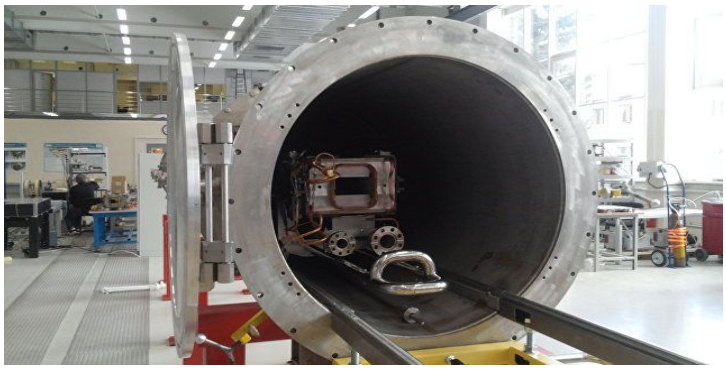

Fig. 1. Excursion at the VBLHEP factory for the production of magnets.

Fig. 2. Excursion on NICA.

In recent years, six new chemical elements with nuclear numbers 113-118 and about 50 new isotopes of transaktinoid elements have been synthesized at FLNR. The direct experimental proof of the existence of an "island of stability" for superheavy elements with a center near Z = 114 and N = 184 is obtained for the first time.

Applied work of FLNR is related to research in the field of nanotechnology, radiation resistance of materials, modification of surfaces. Their further development presupposes the creation of a specialized building equipped with modern analytical and testing equipment (a joint project of JINR and GC Rosnano).

A special place in the applied research of FLNR was the creation of accelerator heavy ion complexes for the industrial production of track membranes: the DC-60 cyclotron for the Eurasian State University (Astana, Kazakhstan); cyclotron DC-110 for the company NANOCASCAD of the special economic zone "Dubna". In recent years, the volume of experiments on testing of electronic components has significantly increased in the interests of the Federal Space Agency (Roskosmos).

Fig. 3. FLNR excursion

PARTICLE-IN-CELL METHOD

The Particle-In-Cell method consists of the initial configuration, the basic settings, and the output of the results. All calculations occur in a cycle. The cycle consists of the following steps:

- Calculate the charge density: the positions of the particles are scattered in the grid
- Calculate the electrical potential: perform by solving the Poisson equation
- Calculate the electric field: from the potential gradient
- Move Particles: update the speed and position according to Newton's second law.
- Generate particles: sources of samples to add new particles
- Output: optional, save the simulation status information
- Repeat: the cycle repeats until the maximum number of time steps has been reached or until the simulation reaches a steady state

The charge density is a scalar variable spatially varying in space. It indicates the number of units of charge per unit volume. We calculate it by distributing the charge of all particles to the nodes of the computational cells, and then dividing by the corresponding volume of the node.

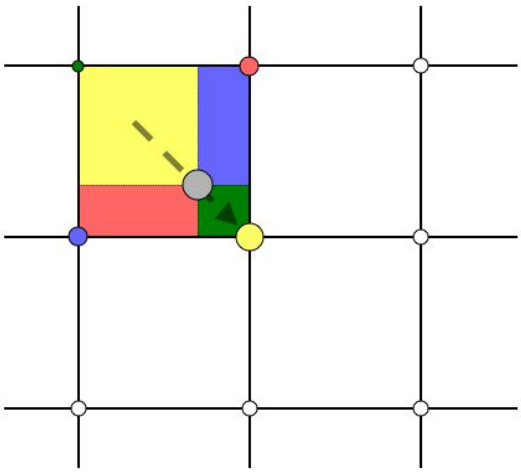

Fig. 4. Schematic of the scattering operation. The charge of the particle (in gray color) is distributed between the surrounding nodes. The charge introduced into each node is based on the proximity of the particle to this node.

The operation of the first order (linear) scattering is schematically shown in Figure 2. This figure shows a fragment of the computational grid and a simulating particle shown in gray. The charge of the particle is distributed between the nodes of the cell

in which the particle is located. Hence the name of this method, the particle in the cell. The yellow node receives the largest fraction because of the close proximity of the particle to this node. The smallest amount is entered in the green node. The weight coefficients are determined by four fractions,

$$\omega_1 = (1-h_x)(1-h_y)$$
$$\omega_2 = (h_x)(1-h_y)$$
$$\omega_3 = (1-h_x)(h_y)$$
$$\omega_4 = (h_x)(h_y)$$

where 1, 2, 3 and 4 correspond respectively to the blue, yellow, green and pink nodes. h_x is the fractional distance of the particle from the origin of the cell in the x direction. The volumes of nodes are equal to the volumes of cells, $\Delta x \Delta y$. Exceptions exist along the boundaries of the domain, since only half (or a quarter for the corners) of the cells participate here. More complex boundary conditions require additional processing. For example, periodic boundaries require summing nodes on the plus (+) and negative (-) side of the domain. This interpolation technology can also be used to scatter data with non-rectangular cells by comparing physical coordinates with a regular logical region.

Next, the electric potential is calculated by calling an elliptic equation solver. The Poisson equation can be discretized as:

$$\frac{\phi_{i-1,j} - 2\phi_{i,j} + \phi_{i+1,j}}{\Delta^2 x} + \frac{\phi_{i,j-1} - 2\phi_{i,j} + \phi_{i,j+1}}{\Delta^2 y} = -\frac{e}{e_0}\left[n_i - n_0 \exp(\frac{\phi - \phi_0}{kT_e}) \right]$$

where we used a finite difference with a central decomposition. This method is modified along the boundaries of the grid. An elliptic equation describes a boundary value problem and therefore such boundary conditions must be prescribed over all boundaries. There are two types of boundaries: Dirichlet and Neumann. The first, Dirichlet, indicates the meaning of the potential. The second boundary specifies the value of the derivative or electric field. Plasma problems usually contain a combination of both types of boundaries.

The electric field is calculated by the difference electric potential, $E_{x,i} = -\frac{\phi_{i+1,j} - \phi_{i-1,j}}{2\Delta x}$ or along boundaries $E_{x,i} = -\frac{\phi_{i+1,j} - \phi_{i-1,j}}{2\Delta x}$.

Only the derivative along the x-axis is shown here, and the one-sided boundary equation is applicable to the x-face. The remaining derivatives are obtained in a similar manner.

Then we integrate the motion of the particle through the time step Δt. For time integration, Boris scheme is used:

$$\vec{r}_{i+1} = \vec{r}_i + \vec{v}_{i+1/2}\Delta t$$

where i is a number of time step and

$$\vec{v}_{i+1} = \vec{v}_+ + \frac{q\Delta t}{2m}\vec{E}(\vec{r}_i)$$

$$\vec{v}_+ = \vec{v}_- + \frac{q\Delta t}{2mc}(\vec{v}_+ + \vec{v}_-) \times \vec{B}(\vec{r}_i)$$

$$\vec{v}_- = \vec{v}_{i-1/2} + \frac{q\Delta t}{2m}\vec{E}(\vec{r}_i)$$

v_+ can be expressed in term of v_- - from the last two equations using algebraic manipulations or geometric considerations:

$$\vec{v}_+ = \vec{v}_- + [\vec{v}_- + [\vec{v}_-, \vec{h}], \vec{s}]$$

$$\vec{h} = \frac{q\Delta t}{2mc}\vec{B}(\vec{r}_i); \vec{s} = \frac{2\vec{h}}{1+h^2}$$

Due to its excellent long-term accuracy, Boris's algorithm is the de-facto standard for moving a charged particle. It was realized that the excellent long-term accuracy of the non-relativistic algorithm of Boris is due to the fact that it preserves the volume of the phase space, although it is not symplectic. The global estimate of the energy error, usually associated with symplectic algorithms, is still satisfied for Boris's algorithm, which makes it an effective algorithm for multiscale plasma dynamics. It has also been shown [5] that it is possible to improve Boris's relativistic momentum to make it both a conservation of volume and to have a solution with a constant speed in the crossed fields E and B.

New particles are generated by sampling sources. Sources include spaceship engines or a solar wind. This step is specific to a specific task.

In some cases, all particles will be loaded initially, and the simulation will only calculate their final distribution state. Usually we are interested in the sample of the Maxwell distribution. As indicated in Birdsall, this distribution can be approximated as follows:

$$f_M = \sqrt{\frac{M}{12}}\left[\sum_{i=1}^{M} R_i - \frac{M}{2}\right]$$

where M is a certain number, and Ri is the i-th random number in the range

[0: 1). The value selected for M controls the accuracy of this method. Birdsall used 12 to prevent records exceeding 6 times the thermal speed.

Finally, since the Boris method requires that the speed and position are shifted from each other, it is necessary to update the velocity of the sampled particles to -Δt. This step is often omitted. The size of the numerical error introduced as a result of this omission will depend on several factors: the strength of the electric field near the source and the size of the time step. This step is also omitted in the code sample, for clarity, as this will require duplication of the speed update code.

The output from the PIC codes includes spatial distribution of plasma parameters such as potential, charge density, electron temperature, as well as particle data such as velocities and current densities. In addition, the temporary output of global cumulative diagnostic data, such as total kinetic and potential modeling energy, is useful for diagnosing code efficiency.

The cycle is repeated until a condition is satisfied. Simulation with continuous sources is triggered until a steady state is achieved. The stationary state is characterized by the number of particles in the modeling domain that remain constant between the time steps. In other words, the number of new particles generated by the sources is balanced by the number of particles leaving the domain through the boundaries.

REALIZATION OF SUPPORT OF CALCULATIONS ON THE GPU

The use of GPUs is advisable in the case when greater capacity is required. The capacity is meant the number of instructions / operations performed per unit of time. This is necessary when processing large data sets is underway, the result of which will not change in the case of paralleling operations on matrix elements.

First of all, the analysis was carried out and we explored the operating time of each part of the algorithm (function or method), for this we used the built-in tools in the

PyCharm development environment. As a result, the most time-consuming ones were functions of calculating the Poisson equation and realizing the motion of particles. These parts of the code were transferred to the GPU using different programming technologies. The time presented in Table 1 demonstrates the time for the implementation of the algorithm for realizing the motion of particles under the same conditions.

The results of the study you can see below:

Table 1.

Name of the technology	The speed of computation and code only calculate on GPU(s)	The speed of computation and code with save data on CPU (s)	The ease of support code	The ease of development code
CPU(one core)	264 ± 2(s)	300 ± 2(s)	++	++
Python/NUMBA	0.559 ± 0.005(s)	55.83 ± 0.5(s)	+	+
Python/PyCUDA	0.0115 ± 0.0005(s)(+)	50.47 ± 0.3(s)(+)		+

The Numba library provides the possibility of jit (just-in-time) compiling code on a python into a bytecode that is comparable in performance to C or Fortran code. Numba supports compiling and running python code not only on the CPU, but also on the GPU, while the style and type of the program using the Numba library remains purely Python, which makes this library convenient for use by a person not familiar with programming on the GPU [6]. We note here a few nice features. First, this implementation is much shorter and more visible. Secondly, if in the C implementation we were required to pass all the constants (for example, N) by executing functions like cudaMemcpyToSymbol (dN, & N, sizeof (int)); then we just use global variables, as in the usual python function. Finally, the implementation does not require any knowledge of C language and GPU architecture.

The second of the tested python libraries was the PyCUDA library. Unlike Numba, the developer will need to write the kernel code in C, so you cannot do without the knowledge of this language. On the other hand, except for the actual kernel on C, you do not need to write anything. The supplement shows the implementation of the cores in these programming languages on the GPU to implement particle motion [7].

CONCLUSION

Summarizing, we implemented the algorithm of the particle method in a cell on a graphics processor using various parallel programming technologies and chose the most optimal for us. PyCUDA library for the python programming language was chosen as the fastest, and at the same time simple in support and implementation of the algorithm.

The results submitted in the V Annual All-Russian Scientific Forum. (The forum will be held on November 21-23, 2018. Details on the site https://www.openscience-forum.com/)

ACKNOWLEDGMENT

I am very grateful to the JINR ESIS group for their help and advice regarding this project throughout the summer program. I would especially like to thank my supervisor Alexey Boytsov for consultations in implementation of software algorithms and physical issues. Computations were held on the basis of the heterogeneous computing cluster HybriLIT (LIT, JINR). I would also like to thank the organizers of the student summer program at JINR and AYSS for performing of the wonderful entertainment and educational program.

REFERENCES

1. R.W. Hockney, J.W. Eastwood, Computer Simulation Using Particles (CRC Press, 1988)
2. Y.N. Grigoryev, V.A. Vshivkov, M.P. Fedoruk, Numerical "Particle-in-Cell" Methods: Theory and Applications (VSP, 2002)
3. A.Yu. Boytsov, A.A. Bulychev, Ef: Software for Nonrelativistic Beam Simulation by Particle-in-Cell Algorithm, EPJ Web Conf. 177 (2018)
4. The Electrostatic Particle In Cell (ES-PIC) Method // PIC-C URL: https://www.particleincell.com/2010/es-pic-method/ (дата обращения: 06.08.2018)
5. Qin, Hong, et al. "Why is Boris algorithm so good?." Physics of Plasmas 20.8 (2013): 084503.
6. Numba // NUMBA URL: https://numba.pydata.org/ (дата обращения: 06.08.2018).
7. Welcome to PyCUDA’s documentation! // PyCUDA URL: https://documen.tician.de/pycuda/ (дата обращения: 07.08.2018).

Supplement A

Implementation of the nucleus of motion of contaminated particles using the library NUMBA

```
@cuda.jit
def region_update_vels_from_bin_interaction(reg, part_dev_0, part_dev_1,
part_dev_2, vel_dev_0, vel_dev_1, pos_dev_0, pos_dev_1):
    pi = cuda.grid(1)
    if pi < part_dev_1.size:
        for pj in range(part_dev_1.size):
            if part_dev_0[pi] != part_dev_0[pj]:
                # supposed to be 1 / r
                rad = 0.0001
                dist_0 = pos_dev_0[pj] - pos_dev_0[pi]
                dist_1 = pos_dev_1[pj] - pos_dev_1[pi]
                norm = 0
                for x in dist_0, dist_1:
                    norm += x**2
                dist_len = (norm) ** 0.5
                if dist_len < rad:
                    tmp_0 = dist_0 / rad / rad
                    tmp_1 = dist_1 / rad / rad
                else:
                    tmp_0 = dist_0 / dist_len / dist_len
                    tmp_1 = dist_1 / dist_len / dist_len
                f_0 = part_dev_1[pi] * part_dev_1[pj] * tmp_0
                f_1 = part_dev_1[pi] * part_dev_1[pj] * tmp_1
                vel_dev_0[pi] += f_0 / part_dev_2[pi] * reg
                vel_dev_1[pi] += f_1 / part_dev_2[pi] * reg


@cuda.jit
def region_update_pos(reg, pos_dev_0, pos_dev_1, vel_dev_0, vel_dev_1):
    x = cuda.grid(1)
    if x < pos_dev_0.size:
       pos_dev_0[x] = pos_dev_0[x] + vel_dev_0[x] * reg
       pos_dev_1[x] = pos_dev_1[x] + vel_dev_1[x] * reg


@cuda.jit
def region_check_rebound(reg_5, pos_dev_0, pos_dev_1, vel_dev_0, vel_dev_1):
    p = cuda.grid(1)
    if p < pos_dev_0.size:
            if pos_dev_0[p] < 0:
                pos_dev_0[p] = - pos_dev_0[p]
                vel_dev_0[p] = -1 * vel_dev_0[p]
            elif pos_dev_0[p] > reg_5:
                pos_dev_0[p] = reg_5 - (pos_dev_0[p] - reg_5)
                vel_dev_0[p] = -1 * vel_dev_0[p]
            if pos_dev_1[p] < 0:
                pos_dev_1[p] = - pos_dev_1[p]
                vel_dev_1[p] = -1 * vel_dev_1[p]
            elif pos_dev_1[p] > reg_5:
                pos_dev_1[p] = reg_5 - (pos_dev_1[p] - reg_5)
                vel_dev_1[p] = -1 * vel_dev_1[p]
```

Supplement B

Implementation of the nucleus of motion of contaminated particles using the library pyCUDA

```
mod = SourceModule("""
    #include<math.h>
    __global__ void region_update_vels_from_bin_interaction(float reg, float
reg_5, float *part_dev_0, float *part_dev_1, float *part_dev_2, float
*vel dev 0, float *vel dev 1, float *pos dev 0, float *pos dev 1)
    {
        int pi = threadIdx.x + blockIdx.x*blockDim.x;
    if(pi < sizeof(pos_dev_0)/sizeof(pos_dev_0[0])){
        for(int pj = 0; pj<sizeof(pos_dev_0)/sizeof(pos_dev_0[0]); pj++){
            if(part dev 0[pi] != part dev 0[pj]){
                float rad = 0.0001;
                float dist_0 = pos_dev_0[pj] - pos_dev_0[pi];
                float dist_1 = pos_dev_1[pj] - pos_dev_1[pi];
                float norm = 0;
                norm+=dist 0*dist 0;
                norm+=dist_1*dist_1;
                float dist_len = sqrt(norm);
                float tmp 0;
                float tmp_1;
                if(dist_len < rad){
                   tmp_0 = dist_0 / rad / rad;
                   tmp_1 = dist_1 / rad / rad;
                } else{
                    tmp_0 = dist_0 / dist_len / dist_len;
                    tmp_1 = dist_1 / dist_len / dist_len;
                }
                float f_0 = part_dev_1[pi] * part_dev_1[pj] * tmp_0;
               float f_1 = part_dev_1[pi] * part_dev_1[pj] * tmp_1;
               vel_dev_0[pi] += f_0 / part_dev_2[pi] * reg;
               vel_dev_1[pi] += f_1 / part_dev_2[pi] * reg;
            }
           }
          pos_dev_0[pi] = pos_dev_0[pi] + vel_dev_0[pi] * reg;
          pos_dev_1[pi] = pos_dev_1[pi] + vel_dev_1[pi] * reg;
            if(pos_dev_0[pi] < 0){
                pos_dev_0[pi] = - pos_dev_0[pi];
                vel_dev_0[pi] = -1 * vel_dev_0[pi];
            }else if( pos_dev_0[pi] > reg_5){
                pos_dev_0[pi] = reg_5 - (pos_dev_0[pi] - reg_5);
                vel dev 0[pi] = -1 * vel dev 0[pi];
            }if (pos_dev_1[pi] < 0){
                pos_dev_1[pi] = - pos_dev_1[pi];
                vel_dev_1[pi] = -1 * vel_dev_1[pi];
            }else if( pos_dev_1[pi] > reg_5){
                pos dev 1[pi] = reg 5 - (pos dev 1[pi] - reg 5);
                vel_dev_1[pi] = -1 * vel_dev_1[pi];
            }
    }
    }
    """)
```