COMPARISON OF PYTHON 3 SINGLE-GPU PARALLELIZATION TECHNOLOGIES ON THE EXAMPLE OF A CHARGED PARTICLES DYNAMICS SIMULATION PROBLEM

A. Boytsov [1,a], I. Kadochnikov [1,4,b], M. Zuev [1,c], A. Bulychev [2,d], Ya. Zolotuhin [3,e], I. Getmanov [3,f]

[1] *Joint Institute for Nuclear Research, 6 Joliot-Curie, Dubna, Moscow region, 141980, Russia*

[2] *Independent researcher, Moscow, Russia*

[3] *The First Electrotechnical University "LETI", 5 Professora Popova, St. Petersburg, 197376 Russia*

[4] *Plekhanov Russian University of Economics, 36 Stremyanny per., Moscow, 117997, Russia*

E-mail: [a] boytsov@jinr.ru, [b] kadivas@jinr.ru, [c] zuevmax@jinr.ru, [d] a.a.letterbox@gmail.com, [e] yaroslav.zolotukhin@bk.ru, [f] igor.getmanov1@gmail.com

Low energy ion and electron beams, produced by ion sources and electron guns, find their use in surface modification, nuclear medicine and injection into high-energy accelerators. Simulation of particle dynamics is a necessary step for optimization of beam parameters. Since such simulations require significant computational resources, parallelization is highly desirable to be able to accomplish them in a reasonable amount of time. From the implementation standpoint, dynamically typed interpreted languages, such as Python 3, allow high development speed that comes at cost of performance. It is tempting to transfer all computationally heavy tasks on a GPU to alleviate this drawback. Using the example of a charged particles dynamics simulation problem, various GPU-parallelization technologies available in Python 3 are compared in terms of ease of use and computational speed. The reported study was funded by RFBR according to the research project № 18-32-00239\18. Computations were in part held on the basis of the heterogeneous computing cluster HybriLIT (LIT, JINR)

Keywords: Python3, GPU computation, particle dynamics, parallelization, Numba, CUDA, OpenCL.

1. Introduction

The problem of charged particle dynamics simulation has been a challenge for scientists and engineers in many fields, such as plasma physics, accelerator engineering, beam transport, ion sources, high current facilities etc. Simulation allows to predict conditions and results of experiments, facilitating experiment facilities design. In the case of an ion source, it is necessary to simulate an order of 1012 particles [1]. But simulation on this scale can take weeks of CPU time. There are many approaches to drastically speed up computation by parallelization onto distributed clusters or hardware accelerators. Using these techniques complicates development and typically there is a compromise between ease of implementation and computation speed. Nowadays, GPUs have become de-facto standard for parallel computation acceleration: most of the highest-performant supercomputers in the world are GPU-based [2]. Therefore it is essential to utilize GPU capabilities for charge particle simulation.

A program utilizing a GPU for computing has components running on the CPU and GPU. The GPU component performs the compute-intensive and parallelizable operations, and the CPU component performs all the other tasks, such as managing the computation, configuring the GPU and data I/O. It is necessary to explicitly manage data transfer between the CPU and GPU components of the program. To manage the communication between the GPU and the program there are proprietary and open APIs: Nvidia CUDA[3], AMD ROCm, OpenCL, SYCL, vulcan. They can be utilized from many programming languages using libraries native to the language.

We tested 3 popular libraries for GPU computing available for Python 3: Numba[4], PyCUDA[5], PyOpenCL[6]. Numba translates Python functions to optimized machine code at runtime using the industry-standard LLVM compiler library. Numba-compiled numerical algorithms in Python can approach the speeds of native compiled code written in C or FORTRAN. Moreover, Numba supports using Nvidia CUDA and AMD ROCm to translate and run python functions on the GPU. PyCUDA gives access to Nvidia's CUDA parallel computation API and PyOpenCL gives access to OpenCL API. We want to compare all three available solutions on the example of a model problem. In the next section we describe the example problem and then describe the obtained results.

2. Example Problem Description

We chose a simple problem as our example: simulating the classical dynamics of a set of N identical charged particles in a 2-dimensional volume of space. Namely, each particle is represented as its 2 cartesian coordinates, 2 velocity components, charge and mass. Dynamics equation are

$$dx = v \cdot dt, \qquad mdv = F \cdot dt$$

The integration was done using the basic Euler method with a constant time step.

$$x_{t+1} = x_t + v_t \cdot h, \qquad v_{t+1} = v_t + a_t \cdot h$$

The force acting on each particle simply as sum of Coulomb forces from the other particles.

$$\overrightarrow{F_i} = \sum_{i \neq j} \frac{q_i q_j \overrightarrow{r_{ij}}}{{r_{ij}}^3}$$

Coulomb's constant is 1 here, as we use arbitrary units for simulation. The particles are contained in a two-dimensional square volume. On reaching the boundary, a particle gets perfectly elastically reflected. Initially the particle positions are evenly non-randomly distributed on a spatial grid, while the velocity components are uniformly distributed in the interval ($-V_{max}$, V_{max}). Observing the coordinate and velocity distribution of the particles over time allows to debug the simulation implementations.

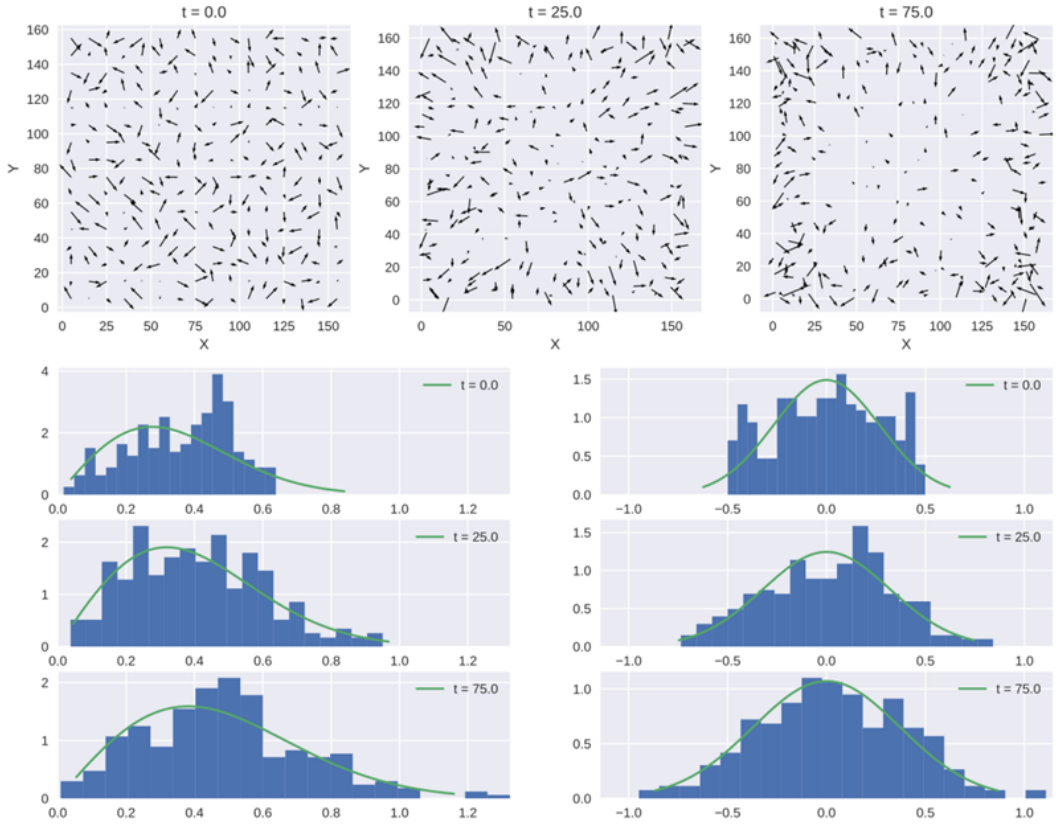

Figure 1. Evolution of coordinate and velocity distributions of particles over time in the CUDA simulation

A sample evolution of 256 particles as simulated by CUDA time is shown on Figure 1. As expected, the velocity x-component fits a normal distribution, while magnitude fits a Rayleigh (2-degree chi) distribution.

For all the simulations the parameters are set as follows: m = 2.0, q = 1.0, V_{max} = 1.0, h = 0.005 With increasing number of particles, the volume dimensions L_x=L_y were increased accordingly to keep particle density $P = \frac{n}{L_x L_y}$ at 0.01 over different scales of simulation. This allowed to keep simulation stable, as the numerical integration requires forces between particles to be sufficiently small.

3. Performance metrics

To compare different GPU acceleration libraries for simulation a one-step function was implemented with each library using analogous code. Then the performance of simulation for each library was measured depending on the scale of simulation. At each scale the number of particles and volume dimensions were set, then the initial particle positions and velocities were generated. The data was copied to GPU memory, and one batch of simulation steps was performed and timed, each step corresponding to one step of numerical integration. The batch size was varied with scale, to keep the batch processing time at a consistent level. The batch needed to be long enough to allow measuring performance accurately, but short enough to measure and plot at many scales.

For each batch, the total simulation time was measured using the standard python time.perf_counter() function. This was then converted into steps per second, particle updates per second and particle interactions per second, which are more comparable across scales.

4. Test results

The development and performance testing were carried out using Google Colaboratory on a Nvidia Tesla K80 GPU, Driver Version 396.44. Library versions are given in Table 1.

Table 1. Versions of libraries tested

Numba	pycuda	CUDA	Pyopencl	OpenCL	Numpy
0.40.1	2018.1.1	9.2.148	2016.1	1.2	1.14.6

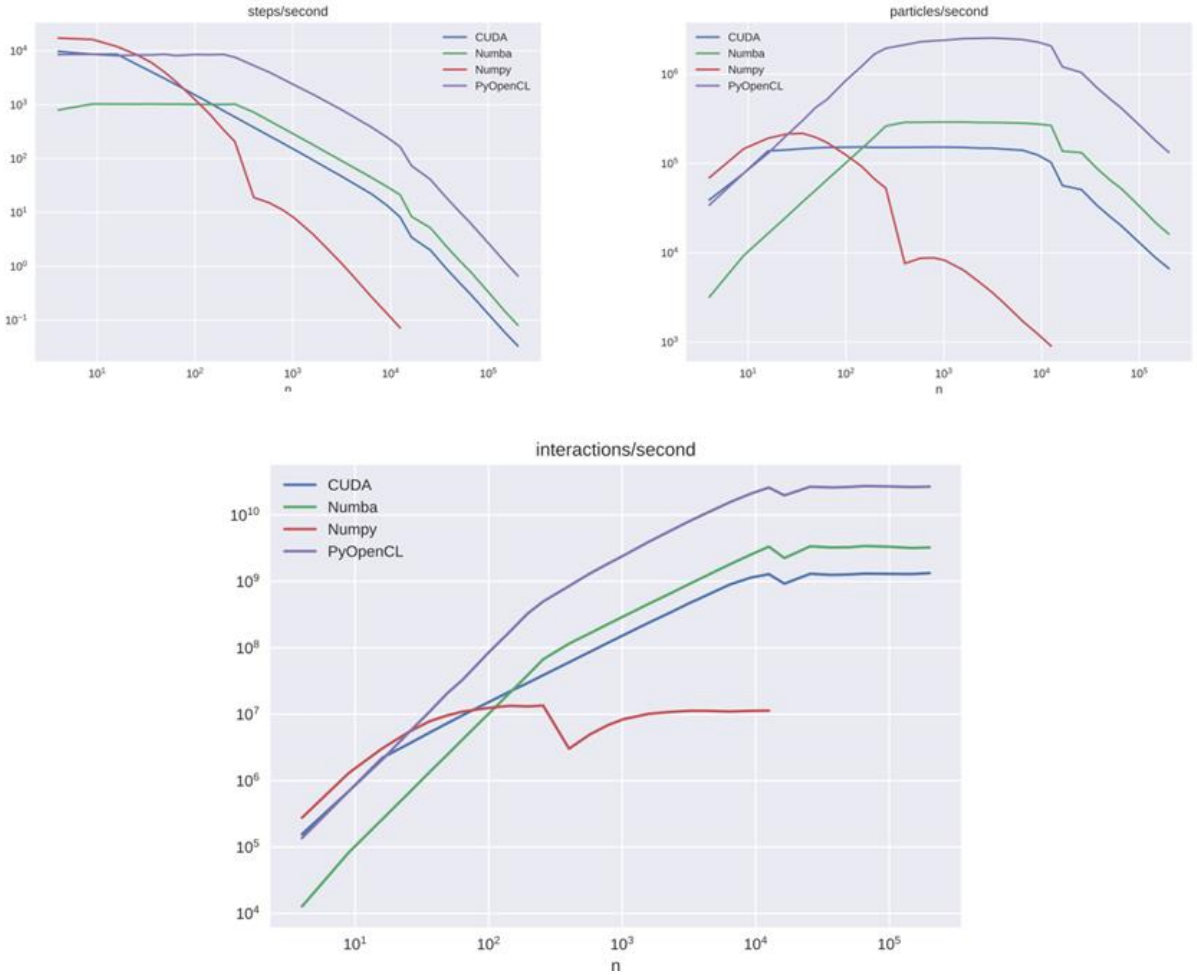

Figure 2 simulation performance metrics depending on number of particles being simulated (n)

In this test both Numba and OpenCL used CUDA as the back-end interface to the GPU. Numpy runs on the CPU and provides the baseline performance to compare GPU speed-up.

The plots of performance results are shown in Figure 2. At small scales ($n < 10^2$) GPU steps per second are constant, while CPU stays ahead. This suggests that start-up overhead dominates per-particle computation. At scales of $10^2 < n < 10^4$ it can be observed that particle updates per second is near-constant, that is to say, computation time grows as O(n). As each particle update consists of O(n) force computations to derive acceleration, we can conclude that at these scales all update streams run perfectly in parallel. As a check, one can compare the update thread count (n) to the number of threads the GPU can execute in parallel. This can be derived from the GPU multiprocessor count and max threads per multiprocessor. For the Tesla K80, this is $13 \times 2048 = 26624$ which agrees with the critical point that can be seen at $\sim 2 \cdot 10^4$. At large scales of $n > 10^4$ the GPU is fully utilized and particle interactions per second tend to a constant value, signifying $O(n^2)$ performance.

5. Conclusion

We compared several GPU parallelization technologies available for Python 3. All provide considerable (~10^3) speed-up compared to CPU computing using Numpy. OpenCL seems to outperform Numpy and CUDA. However, this test did not check all factors that may affect GPU performance. Among those outside of scope of this paper are: data type and shape of arrays being processed, thread block size and shape, using scalar or constant parameters for mass and charge, library version, driver version, GPU model, optimized block caching, optimizing the algorithm.

Acknowledgements

The reported study was funded by RFBR according to the research project № 18-32-00239\18. Computations were held partially on the basis of the heterogeneous computing cluster HybriLIT (LIT, JINR).

References

[1] E. D. Donets, "Review of the JINR Electron Beam Ion Sources," *IEEE Transactions on Nuclear Science*, vol. 23, no. 2, pp. 897–903, Apr. 1976.
[2] "June 2018 | TOP500 Supercomputer Sites." [Online]. Available: https://www.top500.org/lists/2018/06/. [Accessed: 06-Nov-2018].
[3] "CUDA Zone," *NVIDIA Developer*, 18-Jul-2017. [Online]. Available: https://developer.nvidia.com/cuda-zone. [Accessed: 06-Nov-2018].
[4] "Numba: A High Performance Python Compiler." [Online]. Available: https://numba.pydata.org/. [Accessed: 06-Nov-2018].
[5] "PyCUDA." [Online]. Available: https://mathema.tician.de/software/pycuda/. [Accessed: 06-Nov-2018].
[6] "PyOpenCL." [Online]. Available: https://mathema.tician.de/software/pyopencl/. [Accessed: 06-Nov-2018].