3D3V plasma kinetics code DiamondPIC for modeling of substantially multiscale processes on heterogenous computers

A.Yu. Perepelkina, V.D. Levchenko, I.A. Goryachev

Keldysh Institute of Applied Mathematics, Moscow, Russia

Modern problems in plasma physics put many requirements on particle-in-cell (PIC) codes. It is often preferable to conduct numerical experiments with full 3D3V geometry, self-consistent fields, fine mesh, high order of accuracy, and be able to simulate large systems. These requirements strain the limits of computer capabilities. General purpose graphical processing units (GPGPU) with CUDA technology offer new computing power, and it is crucial to use it with maximum efficiency.

A new PIC code DiamondPIC is now in development. We discuss the features of the code and show its advantages over existing software.

Model

A 3D3V self-consistent Vlasov-Maxwell system of equations is the mathematical model for the code. For Maxwell equations FDTD method of 2nd, 4th and higher order is used. Yee cell with electric field components on cell faces and magnetic field components on cell edges is used here. Particle-in cell method is used to solve Vlasov equation. To account for high order schemes we consider form-factors of the following form.

$$\Lambda^{\ell}(x-x') = \int \Lambda^{\ell-1}(x-x'')\Lambda^{0}(x'-x'')dx'',$$

With $\ell = 0$ $\Lambda^0(x)$ is a rectangle function and its support covers exactly one cell. Here ℓ corresponds to the order of scheme approximation. Form-factors with ℓ not less than 2 will be used, so that the order of approximation of the whole calculation would not be less than 2. In this paper, $\ell = 2$ is assumed.

Based on a given problem, the limitations on time step for field evolution dt_{fld} and for particle movement dt_{PIC} may differ. The ratio dt_{PIC}/dt_{fld} of about $\sim 1 \div 20$ will be considered. In the terms of the algorithm it means that one particle push and current deposition step takes place after $1 \div 20$ field evolution steps.

DiamondTorre algorithm for FDTD method

Since FDTD method has cross shaped stencil DiamondTorre 2D algorithm may be used. In outline, the algorithm with its CUDA-based implementation can be described as follows. We consider a mesh of $Nx \times Ny \times Nz$ size. The field data on mesh is contained in 2D array of size

$Nx \times Ny$ in which 9 (3 components of electric and magnetic fields and current densities) 1D arrays with length Nz are stored. 2D DiamondTile at a given (x,y) coordinate is defined as the set of field values which fall into the diamond shape on the mesh on two (E and B) time layers (fig. 1). The upper layer is shifted by half the stencil size (half the cell for 2nd order FDTD; 1.5 cell for 4th oder) to the positive direction of x axis (see fig. 1). Whole arrays of fields in z direction corresponding to the specific (ix,iy) index are assigned to each tile.

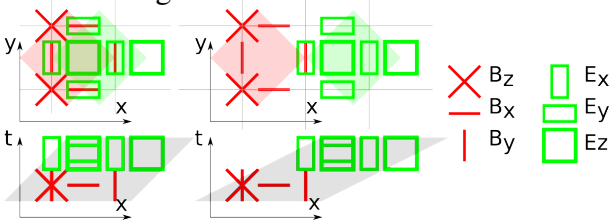

Figure 1: *One DiamondTile for 2nd (left) and 4th (right) order FDTD*

Algorithm is processed in Nx stages. At each odd stage ($I = 1,3,5,...$) $Ny/2$ tiles with (Nx-I,iy) for *even* iy ($iy = 0,2,..,Ny-2$) indexes are computed asynchronously, with each tile assigned to separate CUDA block. For diamond tiles at Nx-1 some part happens to fall outside the domain; these values are not computed. In each tile calculations are distributed between CUDA threads based on z coordinate. To compute finite differences in z direction, the values are saved to shared memory, CUDA threads are synchronized, and then the differences of values saved to shared memory are calculated. In each tile electric field values are computed after magnetic fields.

At each even stage diamond tiles with (Nx-I,iy) for *odd* iy values are calculated in the same way. After all stages are done the whole simulation domain data progresses by 1 time step.

If one chooses $dt_{PIC}/dt_{fld} > 1$, the tiles are stacked onto each other to form tilted towers (called 'Torre' in Italy, hence the name of the algorithm). This way, at each stage, each CUDA block processes consequently $Nt = dt_{PIC}/dt_{fld}$ tiles at coordinates $(ix + (NO-1)it, iy)$ for ix corresponding to the current stage, with odd or even iy, NO equal to scheme order, $it = 0..Nt-1$.

It should be noted that the until the whole simulation domain is covered the field data at different cells correspond to different time instants, from 0 to Nt. Nevertheless the algorithm provides correct dependencies and DiamondTorres at each stage are *asynchronous* so they may be processed concurrently by CUDA blocks without superfluous synchronizations .

The method is advantageous since it provides vectorized access to mesh data and mitigates the misbalance of memory bandwidth to calculation efficiency.

At present state the achieved calculation rate of FDTD calculation is $\sim 1.6 \cdot 10^9$ cells per second for 4th order and $5 \cdot 10^9$ for 2nd order on one nVidia GeForce GTX TITAN.

Paritcle-in-cell implementation specifics

The common issues to address here is to organize particle data structures so that data locality is preserved both in field to particle interpolation step, and in current deposition step. Particles

move freely in the mesh, this requirement also assumes that resorting of particles takes place.

The proposed algorithm works as follows. Particles are organized in cubes of $Nc \times Nc \times Nc$ size. For a particle push step, each cube is executed by one CUDA-block, and particle calculations are distributed between threads. To accelerate particles located inside the cube, $2 \cdot 3 \cdot (Nc+2)^3$ field values are needed. Assuming particles may cross m boundaries in one direction away from the original position, particle affect $3(Nc+2+2m)^2(Nc+1+2m)$ current density values on mesh in the worst case.

To accelerate particles in the cube, it appears efficient to get field values through read-only GPGPU cache, since these are not modified during particle push step.

To update current densities, it is proposed to allocate an array in shared memory in each CUDA-block. Initially, the array is filled with zeroes. Using atomic operations to avoid race conditions, particles add some values to the array elements during their movement. Current deposition is an additive operation, so the resulting array is projected by summation on the original array of cells.

For both these considerations, $Nc = 8$ is most appropriate. Required fields occupy $\sim$24KB, which is equal to read-only cache size on modern compute capabilities. Current density array occupies $\sim$ 19KB, 30KB, and 46KB for $m = 1,2$ and 3 correspondingly, which fits into the available shared memory capacity. Both field and particle data amounts to $\sim$ 100 Bytes per particle if the amount of particles in cell is $N_{PIC} = 1$, and $\sim$ 8 Bytes per particle for $N_{PIC} = 10$. If no such clustering is used for field arrays, total required data save/load amount per particle would be $\sim$ 1KB, so the benefit of this approach is evident.

Let us now discuss how particle sorting takes place. Particle data management is organized alike virtual memory of operating systems. Particle data is stored in a 'page', pages are organized in an array. A control structure as a *table of page indexes* provides methods to get particle with specific index; to write a particle to a page; and to free a page.

Each cube has a table to look up particles from the cube in an array. It also contains a 'reservoir' table of indexes. A separate structure exists for the table of free pages in the array. When particles are moved, they are saved to the reservoir of the cube, in which they end their movement on the current step. After the cube is processed, its main table of indexes is cleared completely and pages are freed. When the cube with all its neighbors is processed, its reservoir is copied to main table, then cleared. So particles remain sorted throughout the simulation and no separate sorting step is needed.

Alike 27-color scheme, for 3D case the particle push is completed in 27 consequent calls of CUDA kernel with $Nx/(3Nc) \times Ny/(3Nc) \times Nz/(3Nc)$ asynchronous CUDA blocks to avoid

conflicts between thread writes.

By current estimates and tests in one second $\sim 10^9$ particles may be processed on one nVidia GeForce GTX TITAN.

Particle-in-cell with DiamondTorre

With DiamondTorre algorithm, the usual state is that some part of simulation domain has processed *Nt* steps and is ready for particle push step, while the part of the domain remains on the initial step. The further step may be added on top of DiamondTorre (fig. 2), so the particle push and current update operations are processed where possible. In the area where the current update is ready, DiamondTorre for FDTD can be further built.

Let us limit the height of DiamondTorre is limited by some number. Each stage of the algorithm then generally does not need the data from whole simulation domain. Only this amount of data is needed to be stored on GPU for this case.

This way even full 3D3V simulations may be processed without large amount of GPU.

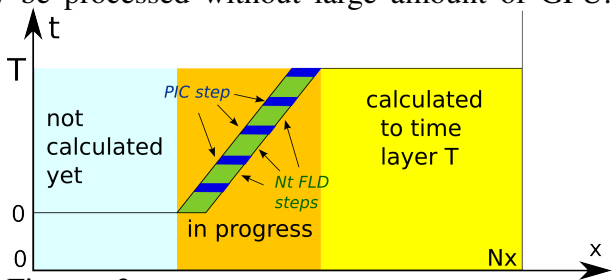

Figure 2: *DiamondTorre. Only data for 'in progress' calculation is located in GPU memory.*

Conclusion

The code is aimed to utilize GPGPU capabilities with maximal efficiency. The advantages of the presented code are the following.

FDTD simulation is vectorized by data and already achieve up to $5 \cdot 10^9$ cells per second calculation rate.

The particles remain localized throughout simulation so that no separate sorting step is needed.

The maximal size of the simulation domain is not limited by available GPU memory capacity.

Aknowledgements

The work is supported by RFBR grants 12-01-00708, 14-01-31483.

References

[1] A. Yu. Perepelkina, V. D. Levchenko and I. A. Goryachev, Journal of Physics: Conference Series **510**, 1, 012042, (2014)

[2] M. Bussman et al, Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, 5:1–5:12, (2013)

[3] X. Kong, M. C. Huang, C. Ren and V. K. Decyk, Journal of Computational Physics, **230**, 4, 1676-1685 (2011)