AFRL-AFOSR-UK-TR-2014-0019

Computational Performance of Intel MIC, Sandy Bridge, and GPU Architectures: Implementation of a 1D c++/OpenMP Electrostatic Particle-In-Cell Code

Giovanni Lapenta, A. Vapirev, J. Deca, S. Markidis, I. Hur and J.-L. Cambier

KATHOLIEKE UNIVERSITEIT TE
INST. OPENB. NUT
OUDE MARKT 13
LEUVEN 3000 BELGIUM

EOARD Grant 12-2048

Report Date: May 2014

Final Report for 15 March 2012 to 15 March 2014

Distribution Statement A: Approved for public release distribution is unlimited.

Air Force Research Laboratory
Air Force Office of Scientific Research
European Office of Aerospace Research and Development
Unit 4515 Box 14, APO AE 09421

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE ***(DD-MM-YYYY)***	**2. REPORT TYPE**	**3. DATES COVERED** *(From – To)*
05-05-2014	Final Report	15 March 2012 – 15 March 2014

4. TITLE AND SUBTITLE

Computational Performance of Intel MIC, Sandy Bridge, and GPU Architectures: Implementation of a 1D c++/OpenMP Electrostatic Particle-In-Cell Code

5a. CONTRACT NUMBER

FA8655-12-1-2048

5b. GRANT NUMBER

Grant 12-2048

5c. PROGRAM ELEMENT NUMBER

61102F

6. AUTHOR(S)

PI and Corresponding Author: Giovanni Lapenta
Other Authors: A. Vapirev, J. Deca, S. Markidis, I. Hur and J.-L. Cambier

5d. PROJECT NUMBER

5d. TASK NUMBER

5e. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

KATHOLIEKE UNIVERSITEIT TE
INST. OPENB. NUT
OUDE MARKT 13
LEUVEN 3000 BELGIUM

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

EOARD
Unit 4515
APO AE 09421-4515

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/AFOSR/IOE (EOARD)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

AFRL-AFOSR-UK-TR-2014-0019

12. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution A: Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

We present initial comparison performance results for Intel MIC, Sandy Bridge (SB), and GPU. A 1D explicit electrostatic particle-in-cell (PIC) code is used to simulate two-stream instability in plasma. We compare the computation times for various number of cores/threads and compiler options. The parallelization is implemented via OpenMP with maximum thread number of 128. Parallelization and vectorization on the GPU is achieved with modifying the code syntax for compatibility with CUDA. We assess the speedup due to various auto-vectorization and optimization level compiler options. Our results show that the MIC is several times slower than SB for a single thread and it becomes faster than SB when the number of cores increases with vectorization switched on. The compute times for the GPU are consistently about 6 to 7 times faster than the ones for MIC. Compared to SB, the GPU is about 2 times faster for a single thread and about an order of magnitude faster for 128 threads. The net speedup, however, for MIC and GPU are almost the same. An initial attempt to offload parts of the code to the MIC co-processor shows that there is an optimal number of threads where the speedup reaches a maximum.

15. SUBJECT TERMS

EOARD, co-processor, many integrated cores, particle-in-cell; heterogeneous computing

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18, NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Kevin Bollino
UNCLAS	UNCLAS	UNCLAS	SAR	12	**19b. TELEPHONE NUMBER** *(Include area code)* +44 (0)1895 616163

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39-18

Computational Performance of Intel MIC, Sandy Bridge, and GPU Architectures: Implementation of a 1D c++/OpenMP Electrostatic Particle-In-Cell Code

A. Vapirev[1,2], J. Deca[1], G. Lapenta[1*], S. Markidis[3], I. Hur[2] and J.-L. Cambier[4]

Abstract
We present initial comparison performance results for Intel MIC, Sandy Bridge (SB), and GPU. A 1D explicit electrostatic particle-in-cell (PIC) code is used to simulate a two-stream instability in plasma. We compare the computation times for various number of cores/threads and compiler options. The parallelization is implemented via OpenMP with maximum thread number of 128. Parallelization and vectorization on the GPU is achieved with modifying the code syntax for compatibility with CUDA. We assess the speedup due to various auto-vectorization and optimization level compiler options. Our results show that the MIC is several times slower than SB for a single thread and it becomes faster than SB when the number of cores increases with vectorization switched on. The compute times for the GPU are consistently about $6-7$ times faster than the ones for MIC. Compared to SB, the GPU is about 2 times faster for a single thread and about an order of magnitude faster for 128 threads. The net speedup, however, for MIC and GPU are almost the same. An initial attempt to offload parts of the code to the MIC co-processor shows that there is an optimal number of threads where the speedup reaches a maximum.

Keywords
co-processor; many integrated cores; particle-in-cell; heterogeneous computing

[1] *Department of Mathematics, KU Leuven, Celestijnenlaan 200b - bus 2400, Heverlee 3001, Belgium*
[2] *Intel ExaScience Lab, Kapeldreef 75, B-3001 Leuven, Belgium*
[3] *PDC Centre, KTH Royal Institute of Technology, Stockholm, Sweden*
[4] *AFRL/PRSA, Edwards AFB, CA 93524*
*__Corresponding author__: giovanni.lapenta@wis.kuleuven.be

Contents

1 **Introduction** 1
2 **Electrostatic Explicit PIC Algorithm** 2
3 **Overview of the Test Architectures** 3
3.1 Sandy Bridge 3
3.2 GPU 3
3.3 Many Integrated Core (MIC) 3
4 **Two-Stream Instability in Plasma: a test case** 3
5 **Code Parallelization and Vectorization** 4
6 **Performance Results** 6
6.1 MIC vs Sandy Bridge vs GPU execution times . . 6
6.2 Offloading to the MIC co-processor 7
7 **Conclusion** 7
Acknowledgments 8
References 8

1. Introduction

Simulations of physical plasma systems are quite challenging because they require extensive use of computing resources and sophisticated numerical methods. Therefore they are very good for assessing the capabilities of high-performance computers (HPC) [1]. A plasma is a collection of particles (electrons and ions) with certain properties and with well-defined interaction functions between the particles. The description of plasmas can be approached by either a kinetic or a fluid dynamic point of view [2]. The electromagnetic fields involved, however, require in addition a solution of full or approximate Maxwell equations.

A kinetic description is often necessary for collisionless plasmas. Typically, two methods are most commonly used: (1) smoothed distribution function over the space and velocity grids, or (2) Particle-In-Cell - information, based on following the trajectories of many individual particles [2]. Kinetic models are usually very computationally intensive. They are very good in describing the plasma on smaller scales, wave-particle interactions, charge exchange, etc... [3, 4]. Fluid models describe plasmas in terms of smoothed quantities on

larger scales and can not resolve small particle-scale physics phenomena [5]. They are often accurate when the number of collisions is sufficiently high to keep the plasma velocity distribution close to a Maxwell-Boltzmann distribution; time scale of collisions is shorter than the other characteristic times in the system. This is usually not the case in fusion, space and astrophysical plasmas, but still the general picture can be presented quite well with the fluid approach [6, 7].

The microscopic behavior of collisionless plasmas in the presence of an electromagnetic field is kinetically well described by the Vlasov-Maxwell system of equations [8]. If the magnetic field is removed, then the problem becomes electrostatic and the equation set boils down to a Vlasov-Ampere problem [9]. The Vlasov equation describes the evolution of the species probability distribution function, while the electric field behaves according to Ampere's Law.

To solve numerically such a system of equations is rather challenging both algorithmically and computationally. The PIC approach has proven to be very successful in handling the physics of plasmas while maintaining high numerical accuracy [10]. One of the most widely used numerical schemes is the explicit PIC method [11]. However, the standard explicit PIC approach has some numerical stability constraints (the CFL condition) which require that the Debye length must be resolved so that instabilities arising from the finite grid scheme can be avoided. As a result, explicit PIC can be immensely computationally demanding if used to study a realistic three-dimensional plasma system. However, if a proper numerical scheme is chosen as it will be presented later in this study, the computational cost can be significantly reduced for certain problems.

The goal of this work is to report on initial code porting efforts and to assess the performance between Intel Xeon (Sandy Bridge) and Intel Xeon-Phi (MIC) architectures (installed at the Intel ExaScience Lab in Leuven, Belgium) and compare it with the performance of a GPU unit running CUDA. We implement a test case of a 1D two-stream instability problem in plasma and we advance the solution in time using a Vlasov-Ampere (VA) explicit PIC method. The authors would like to stress to the fact that in the present work no attempt has been made to explicitly vectorize or optimize the code according to the specifications of each one of the different architectures which would be the case for the majority of the existing codes and models in any field. Minor changes have been made regarding the programming language syntax only in the GPU/CUDA version of the code and these changes do not have any significant impact on the final performance.

2. Electrostatic Explicit PIC Algorithm

In this paper we use a 1D Vlasov-Ampere (VA) explicit PIC method to study the evolution of a simple 1D two-stream instability problem in an electrostatic collisionless fully ionized plasma [12]. In this case the governing Vlasov-Ampere equations are:

$$\frac{\partial f_\alpha}{\partial t} + \mathbf{v}\cdot\nabla f_\alpha + \frac{q_\alpha}{m_\alpha}\mathbf{E}\cdot\nabla_v f_\alpha = 0 \qquad \varepsilon_0\frac{\partial \mathbf{E}}{\partial t} + \mathbf{J} = 0 \tag{1}$$

where $f_\alpha(\mathbf{r},\mathbf{v})$ is the particle distribution function for species α, q_α and m_α are the species particle charge and mass, $\mathbf{E}$ is the self-consistent electric field, ε_0 is the electric permittivity in vacuum.

The evolution of the system is determined by calculating the position and velocity $(\mathbf{x}_p, \mathbf{v}_p)$ for each particle p by solving the equations of motion:

$$\frac{d\mathbf{x}_p}{dt} = \mathbf{v}_p \qquad m_p\frac{d\mathbf{v}_p}{dt} = q_p\mathbf{E}_p \tag{2}$$

where $\mathbf{E}_p$ is the electric field acting on the particle. The electric field is obtained from the Ampere's law which for a system with no initial currents in it is written as:

$$\frac{\partial \mathbf{E}}{\partial t} = -\frac{\mathbf{J}}{\varepsilon_0} \tag{3}$$

To solve numerically the equations of motion 2 we use a simple second-order accurate leapfrog scheme [13]:

$$\mathbf{x}_p^{n+1} = \mathbf{x}_p^n + \mathbf{v}_p^{n+1/2}\Delta t \tag{4}$$

$$\mathbf{v}_p^{n+3/2} = \mathbf{v}_p^{n+1/2} + \frac{q_p}{m_p}\mathbf{E}_p^{n+1}(\mathbf{x}_p^{n+1})\Delta t \tag{5}$$

where n is the loop number in time. Here the particle positions are updated on integer grid points and the velocities on half grid points. The electric field is calculated directly from Ampere's law by discretizing in time:

$$\mathbf{E}^{n+1} = \mathbf{E}^n - \mathbf{J}^{n+1/2}\Delta t/\varepsilon_0 \tag{6}$$

In plasma simulations the electric field acting on a single particle can be computed by directly summing up the contributions due to all other particles in the system. This approach, however, is rather cumbersome when the number of particles becomes big. Instead of computing Equation 6 for each particle we apply the approach used in the so-called Particle Mesh (PM) schemes [14]. We define a mesh of cells and an interpolating function $W(\mathbf{x}_g - \mathbf{x}_p)$ for each particle p in cell g. Then the total current in each cell at half-time level is computed by summing up the currents due to all particles in that cell:

$$\mathbf{J}_g^{n+1/2} = \sum q_p\mathbf{v}_p^{n+1/2}W(\mathbf{x}_g - \mathbf{x}_p)/V_g \tag{7}$$

where V_g is the volume of cell g. From here the electric field in each cell $\mathbf{E}_g$ is obtained by applying the Ampere's law:

$$\mathbf{E}_g^{n+1} = \mathbf{E}_g^n - \mathbf{J}_g^{n+1/2}\Delta t/\varepsilon_0 \tag{8}$$

Finally, in order to advance the particles in time (Equation 5), the electric field acting on a single particle is calculated by interpolating the fields on the mesh back to the particles:

$$\mathbf{E}_p = \sum \mathbf{E}_g W(\mathbf{x}_g - \mathbf{x}_p) \tag{9}$$

Distribution A: Approved for public release; distribution is unlimited.

3. Overview of the Test Architectures

3.1 Sandy Bridge

Sandy Bridge is the codename for a processor architecture developed by Intel for central processing units (CPU) in computers. Intel showcased a SB processor in 2009 and the first consumer-targeted products were released in early 2011 [15]. The first products implemented 32nm technology and later the production was based on a 22nm process [16]. Some of the upgrade features, compared to the previous generation Nehalem cores, include 64-byte cache, improved performance for mathematical operations, and up to 8 physical cores. The SB in this study runs Red Hat 4.4.6-4 with Intel compiler icc-13.0.0 20120731. A single processor runs at 2.60GHz and 20480KB cache size.

3.2 GPU

The GPU (Graphical Processing Unit) handles the computations necessary only for computer graphics. It is specifically designed for fast manipulation and processing of images in a frame buffer. GPUs are widely used in everyday life devices, e.g., mobile phones, personal computers, and game consoles. Although GPUs are extremely efficient at processing graphics, they are also used as a general purpose computing CPU for algorithms where processing of large blocks of data is done in parallel. The reason for that is the GPU's highly effective parallel structure. Most of the image and video processing computations involve heavy matrix and vector operations over large amounts of data and thus GPUs have also been extensively used for non-graphical calculations where parallelism and code vectorization are needed. In this work we use one Tesla M2070 device with 448 CUDA cores and 6GB of GDDR5 memory, reaching a theoretical peak double precision floating point performance of 515 Gigaflops. The NVIDIA Cuda 3.2.16 compiler version is used.

3.3 Many Integrated Core (MIC)

The Intel MIC is a multicore computer architecture based on the earlier Intel Larrabee many core architecture, the Teraflops Research Chip multicore chip project, and the Intel Single-chip Cloud Computer multicore microprocessor. The first commercially released product codenamed *Knights Corner* (KNC) features a 22 nm size technology and utilizes Intel's Tri-gate technology. The Xeon Phi 5110P launched on 28 January with 60 cores runs at 1GHz supported by 8GB of GDDR5 memory and, according to Intel, offers 1 Teraflop of peak double-precision floating point computational capability [17]. The MIC used in this study is a KNC unit with the SB described above acting as a host.

The Xeon Phi processor supports hyperthreading in addition to some new x86 instructions created for the wide vector unit. In order to achieve high computational performance, code developers must utilize both parallelism and vector processing. The current generation of Xeon Phi co-processor cores support up to four concurrent execution threads via hyperthreading. However, while the collective computational performance is high for Intel Xeon Phi, each core is slow and has limited floating-point performance when compared with a modern SB processor. As it will be shown later, good performance can be achieved when a large number of parallel threads are utilized, and they issue instructions to the wide vector units quickly enough to keep the vector pipeline full. The key to boosting the Intel Xeon Phi floating-point performance is the efficient use of the per core vector unit. That, however, may not be the case with many legacy codes and numerical models which utilize algorithms relying on larger cache (L2) rather than vectorization when running in parallel.

4. Two-Stream Instability in Plasma: a test case

To test the computational performance of the different architectures we use the above described numerical method to study a test case of a two-stream instability in plasma in a simple 1D explicit electrostatic PIC code. The code for SB, MIC, and CUDA is written in *c++* with double precision for SB and MIC and single precision for CUDA. The reason to use single precision with CUDA is that the atomic add which is used to update the particles works with *float* only [18]. The *double* version as suggested by Nvidia is very slow and we doubt the corresponding results. On the other hand, *double* is used on both SB and MIC due to the instability of the code when using single precision. In the scope of the present study, we consider that the results in terms of scalability and parallelism are the part which should be considered of real importance. The fact that the difference between using double and single precision could be simply a factor of 2 in terms of speed performance is believed by the authors to have no significant impact on the general picture presented here. Performance comparison for double and single precision on a single core for MIC will be briefly discussed later in the text. The parallelization is achieved by using OpenMP *#pragma* calls. Built-in compiler vectorization flags are used to vectorize the code. The basic code algorithm is presented in Figure 1. First, the system is initialized. Then the particle positions are calculated and the periodic boundary conditions applied. Then we calculate the current in each cell (Equation 7), interpolate it to the mesh, and compute the respective field (Equation 8). After interpolating the field back to the particles, their velocities are computed (Equation 5).

The system size is $l = 10d_i$, where d_i is the ion skin depth. The number of cells is $ncells = 256$, the total number of particles is $nparticles = 2^{17}$ which gives 512 particles per cell. The algorithm runs for $n = 8000$ time cycles. The rest of the system parameters are as follows in relative code units. The plasma frequency is $w_p = 1.0$, time step is $dt = 0.1\omega_p^{-1}$, the base velocity is $v_0 = 0.1$, the thermal velocity is $v_t = 0.002$, and the charge-to-mass ratio is $qm = -1$. The two-stream instability is achieved by initially randomizing the velocity distribution with half of the particles having a negative sign

Distribution A: Approved for public release; distribution is unlimited.

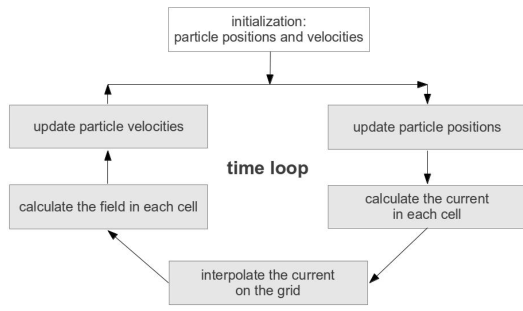

Figure 1. Algorithm of the numerical scheme implemented in the explicit 1D electrostatic PIC code. The main time loop is repeated as many times as desired to advance the evolution of the system.

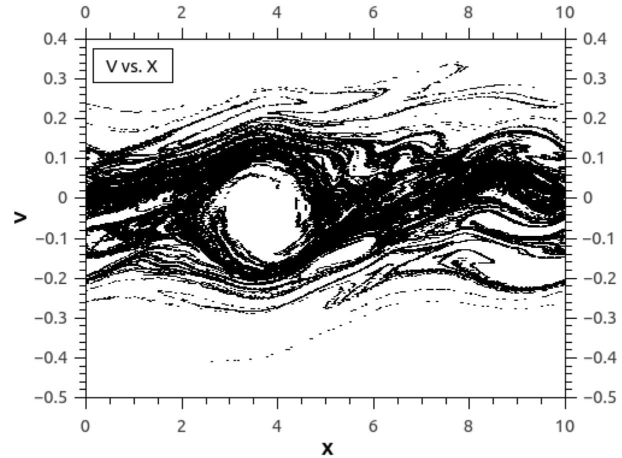

Figure 2. Phase space plot of the evolution of the 1D two-stream instability after 8000 time cycles. Each dot represents one of the 2^{17} particles involved in the calculation with given velocity and position. The instability leads to the formation of a characteristic electron hole in the phase space.

velocity:

$$v_p = (-1)^{ip}(v_0 + rand(v_t)) \tag{10}$$

where ip is the particle counter and goes from 1 to 2^{17}. This creates two particle streams moving in opposite directions. The evolution of the system after 8000 time cycles is presented in Figure 2. In the beginning the particles are distributed more or less evenly throughout the whole simulation domain. After enough time has elapsed, an electron hole is formed in the phase space which is a characteristic of a two-stream plasma instability. Although we discuss specific parts of the code further in the text, here we would like to explain in a little more detail why we have chosen the two-stream instability as our test case. As seen in Figure 2, particles tend to cluster in certain areas of the computational domain, while leaving others practically empty. This poses an extra computational challenge for the threads/processors operating on domains with densely clustered particles. These particular processors will then require more memory directly accessible per processor. In the case of OpenMP implementation it will also force threads performing heavy computations to constantly compete for the same cache which will cause a lot of thread migration and increase the context-switching cost among cores, bad data locality, and increased cache-coherency traffic among the processors [19, 20, 21, 22]. If exactly the same PIC problem is studied but without the initialization of a two-stream instability, i.e., all initial velocities have the same sign in Equation 10, then the computation will be much faster. In that case some small clusterization is possible but that would be only due to the initial randomization and will not pose a significant computational challenge.

Since this is a one-dimensional test with the sole purpose of testing computational performance, no work has been done to assess any possible errors arising from the second-order numerical scheme used in this study. Naturally, for similar problems in two and three dimensions, if more realistic physics results are pursued, such tests are necessary.

5. Code Parallelization and Vectorization

The parallelization of the code in the current study, as mentioned, is done by utilizing OpenMP *#pragma* calls both in the initialization block and in the main computational loop [23, 24]. However, the presented time performance here is only for the main time loop. The code is vectorized using the compiler auto-vectorization options. If using the default *-O2* or the higher *-O3* level of optimization for the *icc* compiler the vectorization is automatically assumed. For *icc* it can be turned off with the flag *-no-vec*. For the GNU *c++* compiler when using the default *-O2* optimization level the vectorization is not automatically assumed and must be turned on with the *-ftree-vectorize* compiler flag. If using *-O3* level with GNU *c++* compiler then the vectorization is automatically assumed and there is no flag to switch it off if wanted (at least to our knowledge). With the help of the *-vec-report[0..5] icc* flag one could see whether a loop is vectorized or not. Due to the simplicity of the code and the small number of cores used in this study, no special precautions have been taken to tackle possible overheads due to synchronization and loop scheduling [25, 26]. A typical OpenMP pragma call used in the code for parallelizing a loop is *#pragma omp parallel for*. For example the loop initializing the particle positions *xp* and the two-stream velocity distribution *vp*:

```
#pragma omp parallel for
for (ip=0;ip<nparticles;ip++) {
    xp[ip] = ip * lbox_constant;
    vp[ip] = pow(-1,ip)*(v0 + vt*( rand() % 1 ));
}
```

is not vectorizable. The reason is that this loop contains the intrinsic function *rand* for which the compiler runtime library does not contain a vectorized version [27]. If that function is removed, the loop will become vectorizable. This is in contrast with the next example loop where intrinsic trigonometric functions are used and vectorization is possible because of the availability of respective vectorized versions. The loop initializing the perturbation:

```
#pragma omp parallel for
for (ip=0;ip<nparticles;ip++) {
    vp[ip] = vp[ip] + v1 * sin(2*pi*xp[ip]/lbox*mode);
    xp[ip] = xp[ip] + xp1 * (lbox/nparticles)
                         * sin(2*pi*xp[ip]/lbox*mode);
}
```

is successfully vectorized. Another interesting case is when a loop is forced to be vectorized when the compiler refuses to do so. This can be attempted in cases where we are sure that there is no vector dependence. For example, in the present problem particles move and at each time step we need to find in which mesh cell *g* they are currently located. This raises the need to search for the cell number based on the current particle position so that the electrical current in that cell can be updated accordingly. For example, the computation of the initial current in each cell based on the computed particle positions is performed as follows:

```
#pragma omp parallel for private(icell)
for (ip=0;ip<nparticles;ip++) {
    // find the cell index
    icell = floorf(xp[ip]/dx);
    jx_update = q*vp[ip];
    #ifdef ivdep
    #pragma ivdep
    #endif
    #pragma omp parallel for
    for (ic=0;ic<ncells;ic++) {
        // sum up current in that cell for all particles
        if (ic == icell) {
           jx_old[ic] = jx_old[ic] + jx_update;
        }
    }
}
```

and this part of the code is not vectorized. In general, the compiler will consider outermost loops for parallelization and innermost loops for vectorization [27]. The reason why the compiler is unable to perform vectorization is the *if* statement in the internal loop where we search for the current cell with index *icell* although we try to explicitly force the compiler to vectorize it with the *ifdef-endif* statement. For the inner loop the *-vec-report2* flag gives *remark: loop was not vectorized: vectorization possible but seems inefficient*. In this particular example since the cell number is integer, the *floorf* function is not necessary because it deals with both positive and negative numbers. The grid (and the particle positions) extends from $0d_i$ to the maximum box length $10d_i$ which is a range of positive numbers. That is why *floorf* is actually replaced with *(int)* which does not perform a check over negative numbers (particle positions), thus saving time. Other similar simple modifications to the code are possible but our experience shows that in the present case it would not significantly improve the performance results. We must also ensure that parallelizing the loop above will not lead to a thread race condition, i.e., to make sure that the current *jx* in each cell *icell* is correctly calculated when competing threads try to access and change its value at the same time. The race condition problem is avoided by introducing the *for private(icell)* pragma call, thus ensuring that the code yields the correct result. On the other hand, *if* statements do not pose a loop vectorization problem when they can be implemented as masked assignments: the calculation is performed for all data elements but the result is only stored for those for which the mask evaluation is true. For example the compiler vectorizes the loops containing *if* clauses when enforcing the periodic boundary conditions because in that case the check is against a constant (the box size) and not against a variable which different threads try to access in the memory at the same time (e.g., *icell* as in the previous example above). This is the case in the particle mover code for the SB/MIC where:

```
#pragma omp parallel for private(icell)
for (ip=0;ip<nparticles;ip++) {
    // update particle positions
    xp[ip] = xp[ip] + vp[ip]*dt;
    // boundary conditions - periodic
    if (xp[ip] >= lbox ) xp[ip]=xp[ip]-lbox;
    else if (xp[ip] < 0) xp[ip]=xp[ip]+lbox;
    // find the cell with index icell
    //in which the particle is
    icell = (int) (xp[ip]/dx);
    ...
}
```

The respective code for CUDA is:

```
// update the particle positions
__global__ void move_part(float *xp, float *vp) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < nparticles) {
        xp[tid] = xp[tid] + vp[tid]*float(dt);
        if (xp[tid] < float(0.0)) {
            xp[tid] = xp[tid] + float(lbox);
        }
        if (xp[tid]>= float(lbox)) {
            xp[tid] = xp[tid] - float(lbox);
        }
        tid += blockDim.x * gridDim.x;
    }
}
```

The rest of the particle mover loop for SB/MIC however, i.e., finding the cell number *icell*, is not vectorizable as shown before in the initialization part. This poses a problem (bottle neck) because the current version of the mover code takes most of the computational time necessary to complete one time loop on both the Intel and GPU architectures exactly because of this required step for calculating the cell number. Thus, the particle mover relies mostly on the cache size rather than on vectorization.

To test that OpenMP correctly parallelizes the model we also experimented with few extra pragma calls in the above mentioned code sections in order to make sure that the threads are indeed properly distributed. For example, forcing *private(icell,ip,jx_update)* for the outer loop and also *private(ic)* for the inner loop results in no change in the model output. We also tried even a stronger condition by setting *firstprivate(jx_old)* before the outer loop. It can be also set before the inner loop. We also introduced *#pragma omp critical* before

the inner loop. By declaring *firstprivate* each threads gets its own copy. This could be considered as a workaround for a reduction clause (currently OpenMP only defines array reduction for Fortran but not for C/C++ because arrays are handled differently depending on whether they are passed or they are locally declared). Thus we do not care about what element the array addresses because it is private to the thread. Note that using *critical* pragma does not allow *omp parallel* for that loop. That is even a stronger condition and we know for sure that the threads have been correctly handled in parallel. None of these extra pragma calls yielded any change in the model output.

6. Performance Results

6.1 MIC vs Sandy Bridge vs GPU execution times

In this section we present the performance results for the three different architectures. The comparison between MIC and SB is done with the Intel *icc* compiler using double precision, where as on the GPU we implement CUDA3.2 using single precision. Although *icc* is not used on the GPU, as it will be shown later, the GPU results differ a lot from the results for the two Intel architectures. Therefore we assume that any possible gain/loss in the GPU computational performance due to difference in the compilers does not bring dramatic changes in the final comparison times. The runs on the SB host are done on 32 cores with 4 threads per core. On the MIC 32 cores with 4 threads per core are used. On the GPU the number of threads per block is varied from 4 to 128 (1024 is the maximum number of threads per block allowed by the device). The number of blocks launched per kernel is chosen so that one thread handles only one cell/particle, with the only limitation being the maximum grid dimensions (65535 blocks). Each presented time is an average of five runs. The number of threads used when running the code is increased as powers of 2. There is a slight difference between the results with *-O2* and *-O3* optimization in favor of the latter case. However, we do not consider that to be of any significance (order of $1-5\%$) and the respective results with *-O2* are omitted hereafter. The standard execution time errors for different number of threads and compiler options with *-O3* optimization are presented in Table 1. For SB the maximum error is around 0.3%. For MIC without vectorization generally the error is less than 1% except for the cases with 2 and 4 threads where it is about 2.4% and 1.6% respectively. For MIC with vectorization the error is smaller than 0.5% except for runs with 2 and 8 threads where it is around 1.4% and 1%. The maximum error for the GPU runs is 0.007% for 4 threads and it consistently decreases for higher thread numbers. All these errors are too small to be responsible for any significant kinks observed in the presented data plots. The error values are also too small to be plotted and therefore are omitted from the figures. Table 1 also contains errors for cases where part of the code has been offloaded to the co-processor and also speedup errors and that will be discussed further in the text.

The execution times in seconds for MIC, Sandy Bridge, and GPU (CUDA) as a function of thread number are presented in Figure 3. On MIC and SB *icc* compiler is used with *-O3* optimization. The red and green lines show the MIC execution times without and with compiler auto-vectorization flag. The respective Sandy Bridge results are shown with blue and purple lines. The results for the GPU with CUDA are plotted in light blue.

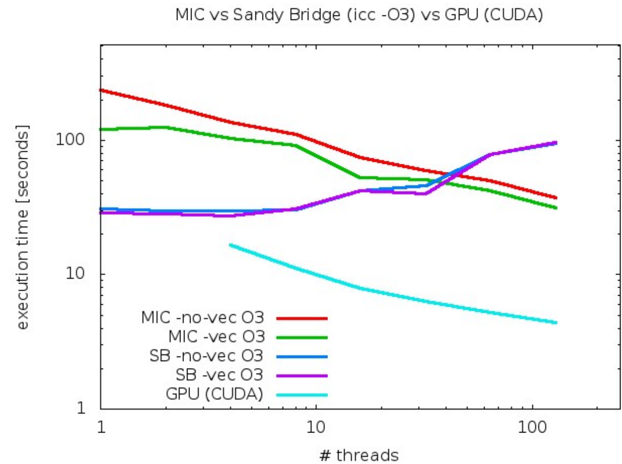

Figure 3. Run times in seconds for MIC, Sandy Bridge, and GPU (CUDA) as a function of thread number. On MIC and SB *icc* compiler is used with *-O3* optimization. The red and green lines show the MIC execution times without and with compiler auto-vectorization. The respective Sandy Bridge results are shown with blue and purple lines. The results for CUDA are in light blue. The code for SB and MIC uses double precision. Single precision is implemented on the GPU.

The SB does not show any significant difference with and without vectorization. The compute times actually increase with increase of the number of threads although at thread number 64 it seems to perform slightly faster with vectorization switched on. Going up to 128 threads however, the two SB times are very similar. This fact implies that probably SB has optimal performance with no more than 2 threads per core, i.e., hyperthreading is optimal when the threads are less than or equal to 64. Once the thread number goes above 2 per core the hyperthreading becomes less efficient and the computational performance goes down.

On the other hand, the MIC is about twice as fast with vectorization on a single thread than without vectorization. However, the difference in performance between vectorized and not vectorized code decreases when increasing the thread number with the vectorized runs still remaining consistently faster for thread number greater than 8. This fact implies that the MIC's Xeon Phi seems to handle very well the hyperthreading (4 threads per core) without a significant performance slow-down. As mentioned earlier, our test code has not been particularly made for scalability as we focus on the performance in general. An interesting fact about the

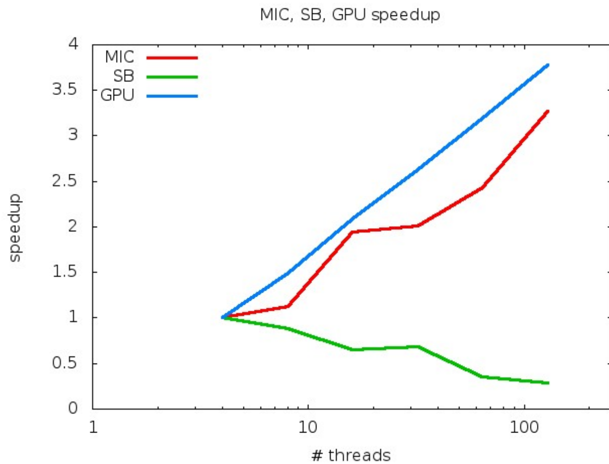

Figure 4. Speedup for MIC (red), Sandy Bridge (green), and CUDA (blue) as a function of the number of threads. MIC and SB use *icc* compiler with *-O3* optimization and double precision. Single precision is implemented on the GPU.

MIC result is that while the non-vectorized code run times steadily decrease from the very beginning, the vectorized code initially experiences a plateau and then shows a significant performance speedup after 4 threads.

The GPU performance is visibly faster than both MIC and SB. At 4 threads the GPU gives 16.615s as compared to 102.805s for MIC and 27.579s for SB (both with auto-vectorization). With increasing the number of threads the GPU is consistently around $6-7$ times faster than the MIC. Increasing the number of threads per block will decrease the total execution time as expected. At this point each multiprocessor (MP) holds 6 blocks simultaneously. This is the closest to the maximum number of resident blocks allowed per MP (8) while at the same time reaching full occupancy (1536 resident threads per MP). Increasing the number of threads/block will not speed up the code any further, because full occupancy per MP has already been reached.

As mentioned above, at this point the current version of the code requires double precision on the Intel architectures (especially MIC) and using *float* sometimes makes the code unstable and hence may yield unreliable results especially if multiple cores are used. Here we show the difference between the results on MIC using *float* and *double* on a single core with *-O3* and auto-vectorization. The MIC performs at 119.367 seconds with *double* vs. 67.936 seconds with *float* which is roughly two times faster. This however, cannot assure that a single precision code will always run twice as fast if using multiple threads/cores.

The story, however, looks quite different if we compare the speedup for the different architectures. The speedup as a function of the thread number for MIC (red line), Sandy Bridge (green line), and CUDA (blue line) as a function of the number of threads is presented in Figure 4. The speedup is computed with respect to the execution time for 4 threads for all three architectures since the GPU cannot go lower than 4 threads. The compiler used on MIC and SB is again *icc* with *-O3* level of optimization. The respective standard errors are presented in Table 1. For SB and GPU/CUDA the errors are below 0.51% and 0.014% respectively while for MIC they are less than 1% except for the runs with 8 threads where the error is around 1.5%. The SB speedup shows a significant decrease in performance which is also seen in Figure 3. For the MIC and GPU the speedup results are quite similar. The GPU speedup steadily increases whereas for the MIC the speedup fluctuates. At thread number 128 it has about the same value as the result for the GPU.

6.2 Offloading to the MIC co-processor

In this section a simple performance test result is presented by offloading to the co-processor (MIC) part of the code in the main computational loop. The parts of the code which require more cache are left on the host (SB), i.e., moving the particles in the cells. The parts in the main loop which are reported as vectorized by the *-vec-report* flag are offloaded via OpenMP pragma calls "*#pragma offload target(mic:0) in(...) out(...)*" before the respective "*#pragma omp parallel for*". The results are presented in Figure 5. The standard errors are presented in Table 1 and the maximum value does not exceed 2.5%. Our tests show that for the current simulation problem there is an optimal number of threads/cores where the speedup is maximal - a little over 1.4 at 4 threads. With the increase of the thread number, however, the speedup performance significantly decreases and results in much slower execution times as compared to the single thread performance. There is practically no difference between the code compiled with *-O2* or *-O3* flags and whether it is vectorized or not. The fastest time for *-O3* vectorized executable at 4 threads is 58.368s: for comparison the respective run times are 27.579s for SB and 102.805s for MIC. There are a number of reasons that could explain such behavior. Due to the nature of our particular plasma physics problem, it is very likely that a major drawback is the constant data exchange between the host and the co-processor when moving particles and updating fields in combination with an insufficient numerical arithmetic intensity on both the host and the co-processor. We consider that in order to achieve better performance it is necessary (especially for large clusters) that each core keeps the data in its own local cache and remote cache access and off-chip memory access should be avoided [28, 29].

7. Conclusion

In this study we used a one-dimensional explicit electrostatic PIC code to simulate a two-stream instability in plasma as an initial performance test for Intel MIC and SB architectures and compare the results with a GPU. The code was run as it is, without any architecture specific optimization techniques on both MIC and SB with double precision. Small changes were made only in the GPU version to fit the CUDA coding standard (using single precision). We compared the compute

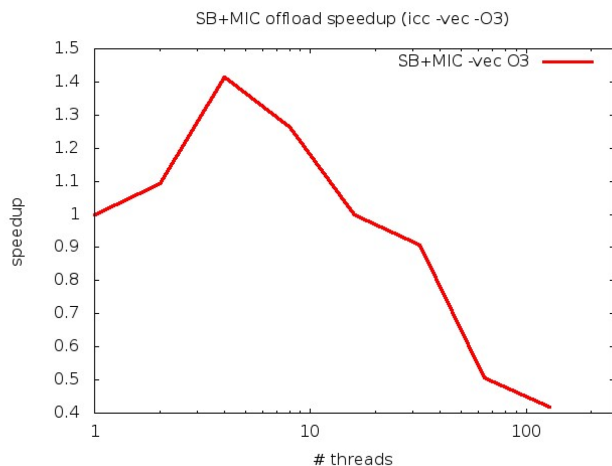

Figure 5. Offload speedup for Sandy Bridge + MIC co-processor. There is an optimal number of threads where the speedup has a maximum.

times for various cases of number of cores/threads and compiler flags. The code was parallelized with OpenMP on MIC and SB and with CUDA on the GPU. The tests were done to a maximum thread number of 128 which was the current limit on the host processor (SB). Our results showed that for a simple test case without explicitly vectorizing the code, for a single thread with *-O2* the MIC was about 8.3 times slower than SB without auto-vectorization and respectively 4.8 times slower with auto-vectorization. When using *-O3* optimization level, the corresponding ratios for a single thread were about 8 and 4.3 times. The MIC, however, became faster than SB when the number of threads went beyond 100 with compiler auto-vectorization switched on. The SB execution times actually became worse with the increase of the number of threads. The GPU showed shorter compute times than the MIC, being consistently about $6-7$ times faster than the respective times for the vectorized code on MIC. In comparison with SB, the GPU was about 2 times faster when using a single thread and about an order of magnitude faster for 128 threads. The reason why MIC showed slower times was that in the current test case the particle mover required large cache to update the cell index and respectively to calculate the updated quantities. Moreover, the current version of plasma mover code did not allow for auto-vectorization to be fully implemented which impeded any possible advantage stemming from the MIC co-processor architecture. The computing performance did not scale well with the increase of the number of cores while keeping the system size constant on either of the Intel architectures and this trend persisted regardless of the compiler optimization level and the vectorization on/off switch. However, when comparing the speedup performance, MIC and GPU showed almost equal result despite the slight "oscillation" of the MIC performance as function of the thread number. Both performance speedups were practically the same for 128 threads, being around $3.7-3.8$ times faster than the performance for 4 threads. A basic effort was made to offload to the co-processor part of the code in the main computational loop. Our tests showed that for the current simulation problem there was an optimal number of threads/cores where the speedup reached its maximum. With the increase of the thread number, however, the speedup performance significantly decreased which was most probably due to the extensive exchange of data between the host and the co-processor. The authors consider that future test cases on MICs should implement architecture specific code optimization and explicit vectorization techniques which would most probably lead to a significant performance speedup, which at least at this point we were unable to demonstrate. At this stage a GPU seems to out-perform a MIC in terms of speed. Extensive studies are necessary for clusters with large number of MICs and GPUs where the communication between the units may play an important role for both speed performance and code scalability.

This work is funded by Intel, the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT), and USAF EOARD Grant No.FA8655-12-1-2048. Part of the research leading to these results has been done in the frame of the DEEP (Dynamically Exascale Entry Platform) project, which received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under Grant Agreement No.287530. The GPU simulations were performed using the KU Leuven High Performance Computing Cluster VIC3.

References

[1] Markidis S, Lapenta G, VanderHeyden WB, Budimlić Z. Implementation and performance of a particle-in-cell code written in Java: Research Articles. *Concurr. Comput. : Pract. Exper.* Jun 2005; **17**(7-8):821–837, doi:10.1002/cpe.v17:7/8.

[2] Schulz M. Introduction to Plasma Theory. *EOS Transactions* 1986; **67**:79–79, doi: 10.1029/EO067i007p00079-02.

[3] Jordanova VK, Kozyra JU, Nagy AF, Khazanov GV. Kinetic model of the ring current-atmosphere interactions. *jgr* Jul 1997; **102**:14 279–14 292, doi:10.1029/96JA03699.

[4] Markidis S, Lapenta G, Rizwan-uddin. Multi-scale simulations of plasma with iPIC3D. *Mathematics and Computers in Simulation* Oct 2010; **80**(7), doi:10.1016/j.matcom.2009.08.038.

[5] Davidson RC. *An introduction to the physics of nonneutral plasmas*. Addison Wesley Publ., Redwood City, CA (USA), ISBN 0-201-52223-3, 1990.

[6] Raeder J, Berchem J, Ashour-Abdalla M. The Geospace Environment Modeling grand challenge: Results from a Global Geospace Circulation Model. *J. Geophys. Res.* 1998; **103**:14 787.

[7] Lapenta G, Lazarian A. Achieving fast reconnection in resistive MHD models via turbulent means. *Nonlinear Processes in Geophysics* Apr 2012; **19**:251–263, doi:10.5194/npg-19-251-2012.

[8] Davidson RC ((ed.)). *Theory of nonneutral plasmas*, vol. 43, 1974.

[9] Briand C, Mangeney A, Califano F. Coherent electric structures: Vlasov-Ampère simulations and observational consequences. *Journal of Geophysical Research (Space Physics)* Jul 2008; **113**:A07219, doi:10.1029/2007JA012992.

[10] Fehske H, Schneider R, Weiße A ((eds.)). *Computational Many-Particle Physics*, *Lecture Notes in Physics, Berlin Springer Verlag*, vol. 739, 2008, doi:10.1007/978-3-540-74686-7.

[11] Dawson JM. Particle simulation of plasmas. *Rev. Mod. Phys.* Apr 1983; **55**:403–447, doi:10.1103/RevModPhys.55.403.

[12] Califano F, Cecchi T, Chiuderi C. Nonlinear kinetic regime of the Weibel instability in an electron-ion plasma. *Physics of Plasmas* Feb 2002; **9**:451–457, doi:10.1063/1.1435001.

[13] Chung TJ. *Computational Fluid Dynamics*. Cambridge University Press, 2002.

[14] Hockney RW, Eastwood JW. *Computer Simulation Using Particles*. New York: McGraw-Hill, 1981.

[15] Intel Corporation. Products (Formerly Sandy Bridge-EP). `http://goo.gl/Nvriw` 2012. [Online; accessed 15-Apr-2013].

[16] Intel Corporation. Intel 22nm 3-D Tri-Gate Transistor Technology. `http://goo.gl/mbRjq` 2011. [Online; accessed 15-Apr-2013].

[17] Intel Corporation. Intel® Xeon Phi™ Coprocessor 5110P. `http://goo.gl/kViCZ` 2013. [Online; accessed 15-Apr-2013].

[18] Corporation N. Cuda Toolkit Documentation 2013. URL `http://goo.gl/F4Y96P`.

[19] Dagum L, Menon R. OpenMP: an industry standard API for shared-memory programming. *Computational Science Engineering, IEEE* 1998; **5**(1):46–55, doi:10.1109/99.660313.

[20] Marathe J, Mueller F. Source-Code-Correlated Cache Coherence Characterization of OpenMP Benchmarks. *Parallel and Distributed Systems, IEEE Transactions on* 2007; **18**(6):818–834, doi:10.1109/TPDS.2007.1058.

[21] Terboven C, an Mey D, Schmidl D, Jin H, Reichstein T. Data and thread affinity in openmp programs. *Proceedings of the 2008 workshop on Memory access on future processors: a solved problem?*, MAW '08, ACM: New York, NY, USA, 2008; 377–384, doi:10.1145/1366219.1366222.

[22] Molka D, Hackenberg D, Schone R, Muller MS. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on Parallel Architectures and Computation Techniques*, 2009; 261–270, doi:10.1109/PACT.2009.22.

[23] OpenMP®API specification for parallel programming 2013. URL `http://openmp.org/`.

[24] Chapman B, Jost G, van der Pas R. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[25] Bull JM. Measuring synchronisation and scheduling overheads in openmp. *Proceedings of the First European Workshop on OpenMP, Lund, Sweden*, 1999; 99–105.

[26] Min SJ, Basumallik A, Eigenmann R. Optimizing OpenMP Programs on Software Distributed Shared Memory Systems. *International Journal of Parallel Programming* 2003; **31**(3):225–249, doi:10.1023/A:1023090719310.

[27] Intel Corporation. ntel C++ Compiler XE 13.1 User and Reference Guides. `http://goo.gl/9VeHAb`. [Online; accessed 30-09-2013].

[28] Rabenseifner R, Hager G, Jost G. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2009; 427–436, doi:10.1109/PDP.2009.43.

[29] Fang J, Varbanescu AL, Sips H, Zhang L, Che Y, Xu C. Benchmarking Intel Xeon Phi to Guide Kernel Design. *Technical Report PDS-2013-005*, PDS May 2013. URL `http://goo.gl/qlt3bf`.

Distribution A: Approved for public release; distribution is unlimited.

Table 1. Standard Error [%] for Different Compiler Options

Number of Threads	1	2	4	8	16	32	64	128
SB -openmp -no-vec -O3	0.028	0.174	0.193	0.285	0.153	0.316	0.153	0.210
SB -openmp -O3	0.001	0.274	0.217	0.108	0.190	0.169	0.051	0.292
MIC -openmp -no-vec -O3	0.786	2.338	1.645	0.680	0.705	0.240	0.292	0.485
MIC -openmp -O3	0.082	1.433	0.491	1.001	0.457	0.384	0.170	0.246
Offload -openmp -O3	0.293	2.474	0.280	0.173	0.999	1.559	1.001	2.149
GPU (CUDA)	-	-	0.007	0.006	0.006	0.005	0.004	0.002
SB Speed-up	-	-	0.434	0.325	0.407	0.386	0.268	0.509
MIC Speed-up	-	-	0.982	1.492	0.949	0.875	0.662	0.737
GPU Speed-up	-	-	0.014	0.013	0.013	0.012	0.011	0.009

Distribution A: Approved for public release; distribution is unlimited.