Title
Fusion PIC Code Performance Analysis on The Cori KNL System

Authors
Koskela, T
Deslippe, J
Friesen, B
et al.

Publication Date
2018-07-17

Peer reviewed

Fusion PIC Code Performance Analysis on The Cori KNL System

Tuomas Koskela[1], Jack Deslippe[1], Brian Friesen[1] and Karthik Raman[2]
[1] National Energy Research Scientific Computing Center, Berkeley, CA, USA
[2] Intel Corporation, Hillsboro, OR, USA *

May 18, 2017

Abstract

We study the attainable performance of Particle-In-Cell codes on the Cori KNL system by analyzing a miniature particle push application based on the fusion PIC code XGC1. We start from the most basic building blocks of a PIC code and build up the complexity to identify the kernels that cost the most in performance and focus optimization efforts there. Particle push kernels operate at high AI and are not likely to be memory bandwidth or even cache bandwidth bound on KNL. Therefore, we see only minor benefits from the high bandwidth memory available on KNL, and achieving good vectorization is shown to be the most beneficial optimization path with theoretical yield of up to 8x speedup on KNL. In practice we are able to obtain up to a 4x gain from vectorization due to limitations set by the data layout and memory latency.

1 INTRODUCTION

Magnetic confinement devices are at present the most promising path towards controlled nuclear fusion for sustainable energy production [1]. The most successful design is the tokamak, a toroidal device where a burning hydrogen plasma is confined by a combination of magnetic field coils and an externally induced plasma current [2]. The ITER project [3], currently in construction phase in southern France, aims at demonstrating the feasibility of a tokamak fusion power plant in the 2030's. To ensure the success of ITER, and to pave the path towards commercial production of fusion energy, self-consistent simulations of the plasma behavior in the whole tokamak volume at exascale are absolutely essential in understanding how to avoid the many pitfalls presented by the complex plasma phenomena that are born from the interplay of electromagnetic fields and charged particles in a fusion reactor.

The Particle-In-Cell (PIC) method is commonly used for simulating plasma phenomena in various environments [4, 5, 6], since directly computing the N^2 number of

*[1] Corresponding email: tkoskela@lbl.gov

particle-particle interactions is impractical. A PIC code solves the kinetics of the particle distribution and the evolution of electromagnetic fields self-consistently. Typically PIC codes consist of four steps that are iterated in a time-stepping loop: (1) field solve, (2) field gather, (3) particle push, and (4) charge deposition. In fusion applications that deal with collisional plasmas, a collision step is normally added. Also, at scale, a particle shift step is introduced that consists of communication between processes due to the motion of particles between computational domains. Steps (1) and (3) are computation intensive, involving linear algebra and time integration of the equations of motion. Steps (2) and (4) are mapping steps that are dominated by memory access.

The vast majority of fusion PIC applications use the *gyrokinetic* theory [7] to reduce the dimensionality of the kinetic problem and, therefore, achieve large savings in computation time. However, it requires calculating higher order derivatives in steps (2) and (3) that set them apart from PIC codes in other fields. Typically the compute time in gyrokinetic PIC codes is dominated by the electron push cycle. Electrons move at a much higher speed than ions and therefore need to be advanced with a much shorter time step. Many codes employ an electron sub-cycling loop where electron-scale field fluctuations are neglected and the electrons are pushed for O(10) time steps between field solves. The electron sub-cycling loop is a prime candidate for performance optimization since it's trivially parallelizable and has a high arithmetic intensity. The main obstacle for high performance is random memory access due to the complex motion of the electrons across the grid.

Cori is the first large-scale supercomputer that is leveraging the Intel Knights Landing (KNL) architecture [8]. It is installed at the National Energy Research Scientific Computing Center (NERSC) at Lawrence Berkeley Laboratory (LBL) in Berkeley, CA. At present it has 9688 KNL nodes and 2004 Haswell nodes, making it the world's 5th fastest supercomputer on the top 500 list. However, getting good code performance on KNL is not always trivial due to various features of the KNL architecture; large number of relatively slow processors, high-bandwidth memory and wide vector processing units. In order to enable key scientific applications to run on Cori, NERSC started the NERSC Exascale Science Applications Program (NESAP) in 2014 [9]. One of the outcomes of NESAP is the development a new methodology for optimizing application performance on KNL [10]. The XGC1 code [11] is the only fusion PIC application accepted to NESAP and serves as a test case for other fusion PIC codes that aspire to run on Cori KNL and future Intel Xeon Phi systems. The unique feature of XGC1 is that it uses real-space coordinates and an unstructured mesh for the field solution making it capable of simulating the full tokamak volume simultaneously.

2 Description of the Toypush Mini-App

We have identified the electron push sub-cycle as the main hotspot of the gyrokinetic XGC1 PIC code [12, 13]. In a typical XGC1 run, the electron push kernel contributes 70% - 80% of the total spent CPU time. Furthermore, the electron push kernel scales almost ideally up to a million cores due to its very low communication overhead [14]. Therefore, it is sufficient to tune its performance on a single node, or even a single core, to improve the performance of the XGC1 code at scale. To study the performance of the

particle push step we have developed a lightweight mini-application that we call *Toypush*. The source code is available at `https://github.com/tkoskela/toypush`. The optimizations discussed in this paper are contained in separate git commits. See Appendix A for a reference to the relevant commits in the repository. The mini-app iterates through steps (2) and (3) of the PIC cycle, i.e. field gather and particle push, but does not perform the field solve or charge deposition steps. The motivation for this exercise is to start with a very simple code that ideally can run at close to peak performance and gradually build up complexity and identify which features of the production code limit the performance and study what can be done to optimize them. This way, we avoid the complications of disentangling interconnected parts of the production code for detailed analysis. The code has been written in Fortran 90, using a hybrid MPI and OpenMP programming model for parallelization.

The particle push algorithm integrates the gyrokinetic equations of motion [11] forward in time using a 4th order Runge-Kutta algorithm. At each time step, the electric field $\vec{E}$ and the magnetic field $\vec{B}$ are interpolated from the mesh vertices to the particle position. After each time step, a search is performed on the unstructured mesh to find the nearest grid nodes to the new position of the particle. In this paper we focus on interpolation from the unstructured mesh to the particle position, which is only performed for $\vec{E}$ in the electrostatic version of XGC1 that is used for most production runs.

Two types of data are kept in memory during the run, grid and particle data. All loops of Toypush are vectorizable, so the data must be accessible contiguously on cache lines for optimal performance. Seven double precision floating point numbers are stored for each particle, three spatial coordinates, two velocity space coordinates, mass and charge. For programming convenience, and to follow the convention of XGC1, the seven particle fields are stored in a derived data type `particle_data` that is passed to the push subroutines. All variables are in 1-D arrays whose size is the total number of particles, except for the spatial coordinates(R,ϕ,z). All three coordinates are needed whenever the position of the particle is calculated, and therefore they are stored in a 2-D array where they can be accessed with unit stride. The grid data is stored in global arrays. At each grid node, 12 floating point numbers are stored, three electric field components, three spatial coordinates, and a 2x3 mapping array that maps real space coordinates to barycentric coordinates. In addition, each grid element is defined by three integers that map to node indices.

We have profiled the code using Vtune software and will discuss the main hotspots in more detail in the remainder of this section.

2.1 Triangle Mesh Interpolation

To simulate unstructured mesh field access, we first set up an "unstructured" mesh that consists of a single mesh element and later expand it to multiple mesh elements. Changing the number of elements does not introduce any new features from the computational point of view, but merely adjusts the particles per element ratio.

The interpolation algorithm on the unstructured mesh is a linear interpolation in barycentric coordinates of the triangular mesh element [18]. Three field components and three coordinates are accessed for each of the three vertices of the triangle. Each

Code 1: A sample code of the field interpolation routine on the unstructured mesh. The outer loop index *iv* is a loop over a block of particles. The index *itri* and the coordinates *y* are unique to each particle and an input to this routine. The variables *mapping tri* and *efield* are stored for each vertex at the beginning of the push step. The output of the interpolation is stored in evec for each particle. In the inner double loop index *i* runs over the three vertices of triangle *tri* and index *j* runs over the three Cartesian components of the field *efield*.

```
1  evec = 0D0
2  do iv = 1,veclen
3   dx(1) = y(iv,1) - mapping(1,3,itri(iv))
4   dx(2) = y(iv,3) - mapping(2,3,itri(iv))
5   bc_coords(1:2) = &
6     mapping(1:2,1,itri(iv)) * dx(1) + &
7     mapping(1:2,2,itri(iv)) * dx(2)
8   bc_coords(3) = 1.0D0 - bc_coords(1) - bc_coords(2)
9   do i = 1,3
10   do j = 1,3
11    evec(iv,i) = evec(iv,i) + &
12      efield(j,tri(i,itri(iv))) * &
13      bc_coords(i)
14   end do
15  end do
16 end do
```

particle has a unique identifier `itri` that points to the triangle the particle is in that is updated after every particle push. All data on the grid is accessed indirectly through this identifier and its value is not known before a search function is called after the push is complete. The search function is discussed in Section 2.3. An extract from the interpolation code is shown in Code 1.

2.2 Equation of Motion

The gyrokinetic Equations of Motion (EoM) integrated in the mini-app are equivalent to the one in XGC1,

$$\dot{\vec{X}} = \frac{1}{D}\left[u\hat{b} + \frac{u^2}{B}\nabla B \times \hat{b} + \frac{1}{B^2}\vec{B} \times \left(\mu \nabla B - \vec{E}\right)\right] \quad (1)$$

$$\dot{u} = \frac{1}{D}\left(\vec{B} + u\nabla B \times \hat{b}\right) \cdot \left(\vec{E} - \mu \nabla B\right) \quad (2)$$

$$D = 1 + \frac{u}{B}\hat{b} \times \left(\nabla \times \hat{b}\right) \quad (3)$$

where u is the parallel speed of the particle in the direction of the local magnetic field vector $\vec{B}$, $\hat{b} = \vec{B}/B$, μ is the magnetic moment and $\vec{E}$ is the gyroaveraged electric field. The EoM is integrated with a standard RK4 algorithm. The calculation of the

Code 2: A sample code of the mesh search routine. The calculation of the barycentric coordinates is similar to Code 1 but now the calculation is done for each mesh element. The variable *eps* is set close to zero. When the search condition is fulfilled *itri* is stored in the output array *id* and the search loop cycles to the next particle.

```
1  id = -1
2  particleloop : do iv = 1,veclen
3   triangleloop : do itri = 1,ntri
4    dx(1) = y(iv,1) - mapping(1,3,itri)
5    dx(2) = y(iv,3) - mapping(2,3,itri)
6    bc_coords(1:2) = &
7      mapping(1:2,1,itri) * dx(1) + &
8      mapping(1:2,2,itri) * dx(2)
9    bc_coords(3) = 1.0D0 - bc_coords(1) - bc_coords(2)
10   if(minval(bc_coords) .ge. -eps) then
11    id(iv) = itri
12    cycle vecloop
13   end if
14  end do triangleloop
15 end do particleloop
```

terms in $\dot{\vec{X}}$, $\vec{u}$ and D has high Arithmetic Intensity (AI) due to multiple vector cross products, and can benefit from good cache reuse. The inverses of terms that appear in the denominator, such as B^2 and R are precomputed to avoid unnecessary divide instructions.

2.3 Mesh Search

The mesh search routine takes advantage of the fact that each of the barycentric coordinates is greater than 0 inside a triangle. In order to search if a point p is inside a mesh element, the algorithm computes the barycentric coordinates of p and compares the lowest value of against 0. If the result of the comparison is true the search is successful and exits, if the result is false the search continues to the next element. In the mini-app, the whole grid is searched until a matching element is found[1]. The grid elements are searched in the order in which they are stored in the global grid array[2].

3 Measurement of Baseline Performance

We use the roofline methodology [15, 16] in our performance measurements to discuss performance on an absolute scale. The roofline model is a visual performance

[1] In XGC1 there is a precomputed filtering layer on top of the mesh in the search algorithm that limits the search to a small number of elements, typically less than 20.

[2] In XGC1, the filter sorts the grid elements in order of decreasing probability to complete the search.

model that can be applied to both applications and computing architectures. It describes performance in terms of flops per second (FLOPS) as a function of Arithmetic Intensity (AI), the ratio of the FLOPS executed vs the bytes read from some level of the cache memory hierarchy. A computing architecture will set roofs of achievable performance that are bound by the compute capability and the memory bandwidth. Placing an application's hot kernels on the roofline chart will give information on attainable performance, current performance bounds, and most promising optimization directions [10]. We used Intel Vector Advisor 2017 to measure the flops performed and bytes transferred from memory by the application to construct the roofline chart.

We ran the code on a single Knights Landing node of the Cori Gerty testbed. The Cori KNL system has a peak performance of about 29.1 PFLOPS/s and is comprised of 9,688 self-hosted KNL compute nodes. Each KNL processor includes 68 cores running at 1.3GHz and capable of hosting 4 HyperThreads (272 HyperThreads per node). Each out-of-order superscalar core has a private 32KiB L1 cache and two 512-bit wide vector processing units (supporting the AVX-512 instruction set). Each pair of cores (called "tile") shares a 1MiB L2 cache and each node has 96GiB of DDR4 memory and 16GiB of on-package high bandwidth (MCDRAM) memory. The MCDRAM memory can be configured into different modes, here we only utilize the *cache* mode in which the MCDRAM acts as a 16GiB L3 cache for DRAM. Additionally, MCDRAM can be configured in *flat* mode in which the user can address the MCDRAM as a second NUMA node. The on-chip directory can be configured into a number of modes, but in this publication we only consider *quad* mode, i.e. in *quad-cache* mode where all cores are in a single NUMA domain with MCDRAM acting as a *cache* for DDR, and in *quad-flat* mode where MCDRAM acts as a separate, *flat* memory domain [17]. We utilize the full node with 68 MPI ranks, but only attach Advisor to one of the ranks for performance measurements. The performance roofs shown are single-thread roofs. We did not use OpenMP threading in these experiments. The benchmark case was set up to run in a few seconds for high optimization throughput. The mini-app was pushing 1 000 000 particles, spread among MPI ranks, for 1 000 time steps.

The baseline performance measurement result is shown in Figure 1 for each individual loop of Toypush. One can immediately see that the EoM solver has high AI and is compute bound. It is performing within 50% to the vector peak flop rate and pushing performance higher would require tailoring the computations for good FMA balance. However, the search and interpolate loops have lower AI and lie in a region where memory bandwidth can be a limiting factor. We also note that neither of them exceeds the scalar add peak flop rate, ie. they are not being vectorized by the compiler, or the vector efficiency is very poor due to eg. unaligned or indirect memory access.

4 Applied Optimizations and Obtained Speedups

The primary aim of the optimizations was to improve the vectorization efficiency in the interpolation and search loops. Analysis with Intel Vtune showed that, when the data is ordered in an Structure of Arrays (SoA) format, the L2 cache miss rate is very low, less than 1%. Therefore, we would expect the kernels to reach the L2 bandwidth roof on the roofline chart if the compiler is able to generate vector instructions. To

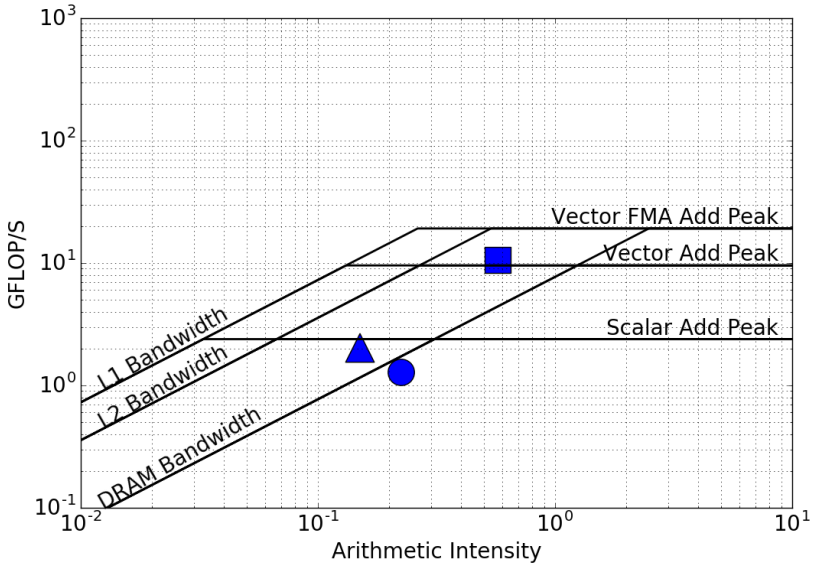

Figure 1: The measured baseline performance on the roofline chart. The square corresponds to the Equation of motion evaluation, the triangle corresponds to the interpolation on the unstructured mesh and the cirlce corresponds to the search on the unstructured mesh. Marker size represents the self time of the loop.

push the performance even further, one has to improve the AI, ie. move to the right on the roofline chart. In the cache-aware roofline model that we are using, changing the AI generally requires fundamental changes to the algorithm. In vectorized code, AI can also be improved by optimizing the data alignment to fully utilize vector load instructions. We will touch on this later in this section. The optimizations we have applied can be broadly divided into two categories: 1) improving the data alignment and 2) enabling vectorization.

4.1 Data Alignment

The vector valued data, field and position, are stored in 2D arrays whose dimensions are ”number of cartesian dimensions” (=3) and ”number of particles”. The aim of the optimization is to place loops over particles as the innermost loop whenever possible to vectorize over particles. Therefore, an SoA data layout results in unit stride access whenever computations can be done one cartesian dimension at a time. We see a 20% speedup with the SoA data layout when data is allocated in MCDRAM. However, when all three or six components of the vector data are needed in a field gather, we find that Array-of-Structures (AoS) data layout can be advantageous since all field components can be loaded from memory on a single cache line.

For vector instructions, it can be beneficial to make sure that the data is aligned on

64 byte boundaries. The Intel Fortran compiler can align 1D arrays automatically with the `-align array64byte` flag. However, on Cori with the Intel 17.0.1 compiler no speedup was observed. Aligning multi-dimensional arrays is less straightforward and for now has to be done by inserting `!dir$ attribute align` - directives into the code when declaring the arrays. We do not implement alignment of 2D arrays in Toypush within the scope of this paper.

Finally, we found up to 40% of total compute time was being spent in calls to `avx512_memset` when running out of DRAM. The number dropped to 10% when running out of MCDRAM, but still remained significant. While the compiler should use avx512 memset instructions, such a large overhead is not expected. We tracked this down to an array initialization at the beginning of the interpolation step, shown on the first line of Code 1. The array `evec` is not large, roughly 1500 floating point values, but it is being initialized at every step of every particle and doing it before executing the loop was costly. By moving the initialization inside the loop and only initializing the element operated by the current loop iteration, we were able to remove this overhead completely, speed up the code by 20% and eliminate any difference in performance between DRAM and MCDRAM. Equal behavior in DRAM and MCDRAM is expected since the code is compute/latency bound, not memory bandwidth bound.

4.2 Vectorization

The key for vectorization is moving loops with long tripcounts to the innermost loops. In a PIC code, loops over particles offer a good candidate for vectorization since the particle arrays are typically much larger than the grid arrays, and concurrent iterations of the particle loop are independent. We tried implementing this in a straightforward manner into the interpolation loop, recall Code 1, by splitting the loop over particles into two parts between lines 9 and 10 and in the latter part moving the short trip counts loop over `i` and `j` to the outside of the particle loop. However, this resulted in poor performance and did not resolve the real issue in the loop, which was indirect memory accessing thourgh `itri(iv)`. We obtained better performance by forcing vectorization of the outer loop by adding an `!$omp simd` directive before the loop and declaring the temporary arrays inside the loop `!$omp private`.

To resolve the indirect access, a change had to be made to the algorithm. We developed two variants of the algorithm

1. Purely scalar grid access. Before entering the loop over `iv`, check if all values of `itri(1:veclen)` are equal. If they are, copy the value to a scalar variable and use it inside the loop. If not, carry out the loop as before.

2. Scalar chunk grid access. We added a preprocessing loop before the main loop, where we calculate the indices where the value of `itri` changes. Then the loop over `iv` is divided into blocks where scalar `itri` is used.

Both algorithms require that the particles are sorted in the beginning of the cycle. Algorithm 1 can only work when the number of particles per mesh element is very large. However, we found that even when the number of mesh elements is set to 1, ie. the

scalar access algorithm can always be used, the overhead from using algorithm 2 instead is only in the order of 10% of the total computation time. A short code snippet demonstrating algorithm 2 is shown in Code 3. The results of the optimizations discussed here are shown on the roofline chart in Figure 2.

Code 3: A sample code of the optimized interpolation routine. The main differences to Code 1 are the preprocessing loop on lines 1 to 10 and the blocking of the loop over iv into num_vec_chunks blocks with direct access to efield and mapping via itri_scalar in each block.

```
1  num_vec_chunks = 1
2  istart(1) = 1
3  do iv = 1, veclen - 1
4   if(itri(iv) .ne. itri(iv+1)) then
5    iend(num_vec_chunks) = iv
6    istart(num_vec_chunks + 1) = iv + 1
7    num_vec_chunks = num_vec_chunks + 1
8   end if
9  end do
10 iend(num_vec_chunks) = veclen
11 do i_vec_chunks = 1,num_vec_chunks
12  itri_scalar = itri(istart(i_vec_chunks))
13  !dir$ simd
14  !dir$ vector aligned
15  do iv = istart(i_vec_chunks),iend(i_vec_chunks)
16   evec(iv,:) = 0D0
17   dx(1) = y(iv,1) - mapping(1,3,itri_scalar)
18   dx(2) = y(iv,3) - mapping(2,3,itri_scalar)
19   bc_coords(1:2) = &
20     mapping(1:2,1,itri_scalar) * dx(1) + &
21     mapping(1:2,2,itri_scalar) * dx(2)
22   bc_coords(3) = 1.0D0 - bc_coords(1) - bc_coords(2)
23   do inode = 1,3
24    do icomp = 1,3
25     evec(iv,icomp) = evec(iv,icomp) + &
26       efield(icomp,tri(inode,itri_scalar)) * &
27       bc_coords(inode)
28    end do
29   end do
30  end do
31 end do
```

The total particle loop is blocked into blocks of size `veclen` and inner loops of size `veclen` are vectorized. We scanned for an optimal value of `veclen` and found that a value of 64 resulted in the best performance. It should be noted that this is 8 times larger than the vector register length of 8 double precision values on KNL, so the performance is a result of L2 cache reuse optimization.

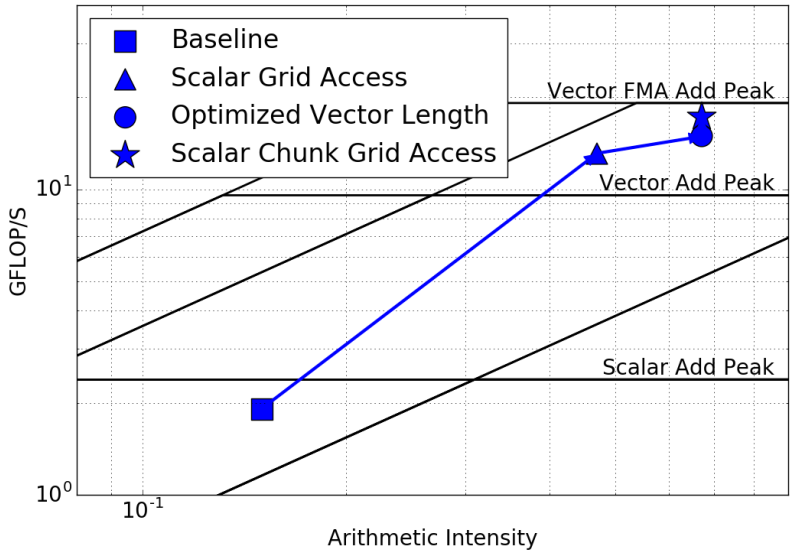

Figure 2: The evolution of the single element triangle mesh interpolation routine on the roofline chart through the optimizations discussed in this paper. The order of optimizations is triangle - scalar grid access, circle - optimize vector length, star - grid access in scalar chunks.

In the search routine we discovered two factors preventing vectorization. First, the cycle of the loop over triangles results in an indefinite trip count of the inner loop. This can be forced to vectorize with a `simd` directive, but on Cori we found the performance to be very poor, worse than the scalar version of the code according to compiler reports. We decided to mask out the iterations of the loop that are not required by the search, which allows the vector lanes that are still searching to keep using a fraction of the vector register. Simultaneously, we reversed the order of the loops in Code 2 to vectorize over the particle loop that has a longer trip count. Second, the compiler could not determine that the local arrays `dx` and `bc_coords` are private to each loop iteration and chose not to vectorize to avoid data races. There are three ways to resolve this issue. In the code snippet shown in Code 4, we chose the least intrusive way, using the `!$omp simd private` directive to instruct the compiler that the arrays are private. The other two ways would be to either separate the array elements into scalar variables that would be treated as private by the compiler or create an extra dimension to the arrays with the size of the trip count of the loop so that loop iterations would access a different element of the array. The effects of the search optimizations to the multiple element version are shown on the roofline chart in Figure 3.

Code 4: A sample code of the vectorized search routine. The main differences to Code 2 are the private declarations on line 2 the reversed order of the loops and the

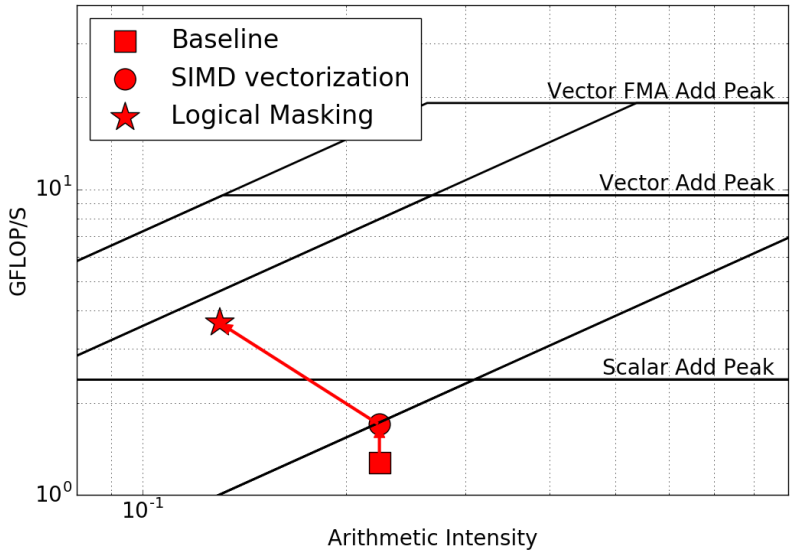

Figure 3: The evolution of the search routine in the multiple element triangle mesh interpolation algorithm on the roofline chart trough the optimizations discussed in this paper. The order of optimizations is circle - simd vectorization, Star - replace cycle with logical mask.

replacement of the cycle command with a logical mask.

```
1  do itri = 1,grid_ntri
2   !$omp simd private(dx,bc_coords)
3   !dir$ vector aligned
4   do iv = 1,veclen
5    if(continue_search(iv)) then
6     dx(1) = y(iv,1) - mapping(1,3,itri)
7     dx(2) = y(iv,3) - mapping(2,3,itri)
8     bc_coords(1) = mapping(1,1,itri) * dx(1) + &
9                    mapping(1,2,itri) * dx(2)
10    bc_coords(2) = mapping(2,1,itri) * dx(1) + &
11                   mapping(2,2,itri) * dx(2)
12    bc_coords(3) = 1.0D0 - bc_coords(1) - bc_coords(2)
13    if(all(bc_coords .ge. -eps)) then
14     id(iv) = itri
15     continue_search(iv) = .false.
16    end if
17   end if
18  end do
```

19 **end do**

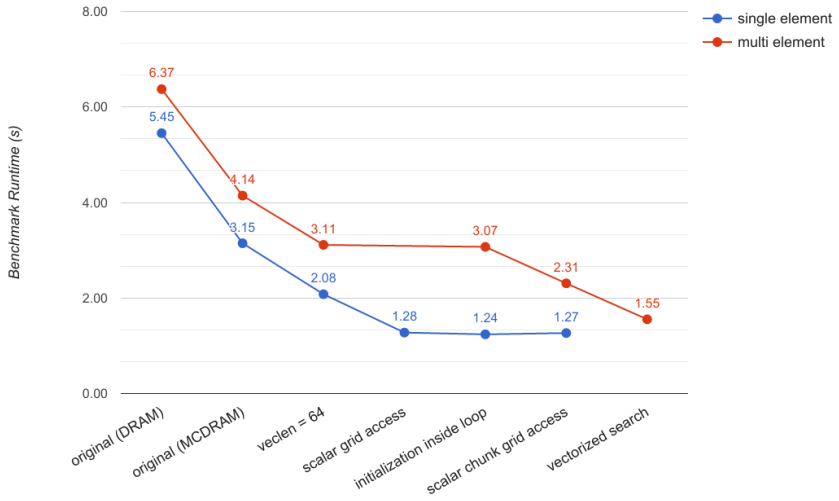

Figure 4: A summary of the obtained speedups on Cori KNL

4.3 Summary of Speedups

The speedups obtained in the Toypush mini-app are summarized in Figure 4. A roughly 4x speedup was obtained in both single-element and multi-element versions of the code, compared to the baseline performance on KNL. This number should be taken with a slight grain of salt, since a large part of the speedup came from allocating into MCDRAM, which is essentially ”free” on KNL. However, the need to use MCDRAM was not necessary after the optimization to array initializations which in a full PIC code might free up MCDRAM for memory bandwidth bound kernels. The most beneficial single optimizations were eliminating the gather/scatter instructions in the interpolation routine, and privatizing temporary variables in the search routine. With the optimizations, the performance of the multiple element code is within 20% of the single element code, provided that the number of particles per element is sufficiently large[3].

The measured performance is compared to the baseline performance on the roofline chart in figure 5. The most significant increase in performance is seen in the interpolation routine, marked by triangles. The optimized performance is close to the peak flop rate and the self time has shrunk by 5x. These measurements were made with the Intel Advisor 17 Update 1 software. The baseline code was limited by the scalar

[3]In XGC1 production runs, the number of particles per element is typically between 10^3 and 10^4, more than enough to fulfill this condition.

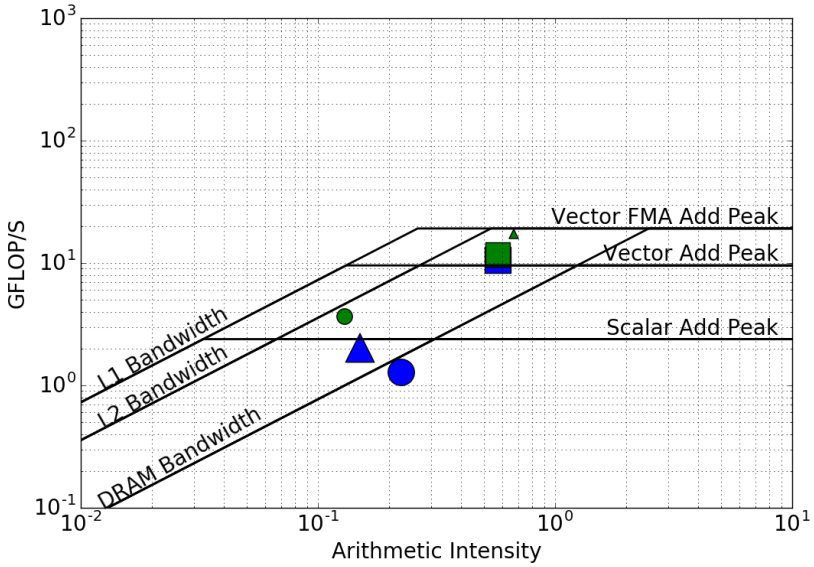

Figure 5: The measured optimized performance on the roofline chart. The square corresponds to the Equation of motion evaluation, the triangle corresponds to the interpolation on the unstructured mesh and the cirlce corresponds to the search on the unstructured mesh. Marker size represents the self time of the loop. Blue markers represent the baseline performance and green markers the optimized performance.

add roof, it was not vectorized due to the indirect memory access inside the loop. The main optimization in both single and multi element codes is eliminating the indirect memory accesses, which increases the FLOPS by a factor eight and also increases AI substantially. The increase in FLOPS is clearly due to utilizing the vector registers, the increase in AI is due to only having to load the grid data once per block of particles instead of once per particle. The loop is now purely compute bound and it is performing at very close to the theoretical peak of the machine. Therefore, further optimizations are not likely to yield gains in performance.

5 Summary and Discussion

In this paper we have discussed recent efforts to optimize the particle push algorithm commonly found in particle-in-cell codes for good performance on the NERSC Cori Phase 2 system that utilizes the Knights Landing manycore architecture. The work has been done on a mini-application that has been built on the basis of the XGC1 one code. The code uses an electron sub-cycling loop to resolve the electron time scale, solves the Poisson equation on an unstructured mesh, and does the particle pushing in real-space cylindrical coordinates. The optimizations that have been discussed are available on

github (see Appendix A) and can be applied back to XGC1, which will be discussed in a future paper.

The optimizations resulted in a 4x speedup of the mini-application due to enabling vectorization and eliminating slow gathers into memory from the most time-consuming loops. The largest gains were made in the electric field interpolation routine, which is now performing at close to theoretical peak of the machine. The search algorithm, that is required after each particle push step to find the correct element on the unstructured mesh, was also vectorized, but is still limited by some inefficiency due to the unknown number of loop iterations before the search is successful. We also saw vectorization reduce the arithmetic intensity of the search routine due to unaligned data access. The main computation loop in the equation of motion is performing at roughly 50% of peak, we were not able to improve on it's performance. Most likely a combination of better FMA balance and register optimization would be required.

The Intel compiler offers some options to reduce the precision of divide operations, that would speed up the computation of the equation of motion. We experimented with removing the divides completely, and were able to reach most of the obtained speedup by lowering the precision of divide operations. However, a more careful validation study is required to understand the implications of reduced precision divides on the scientific results before such optimization can be applied.

ACKNOWLEDGMENT

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231

References

[1] INTERNATIONAL ATOMIC ENERGY AGENCY, "Fusion Physics", Chapteer 1, IAEA, Vienna (2012).

[2] L.A. Artsimovich, Nuclear Fusion, vol 12, no. 2, pp. 215, 1972.

[3] `http://www.iter.org`

[4] Ethier, S., W. M. Tang, and Z. Lin. Journal of Physics: Conference Series. Vol. 16. No. 1. IOP Publishing, 2005.

[5] `http://warp.lbl.gov`

[6] Stefano Markidis, Rizwan-uddin. Giovanni Lapenta, Mathematics and Computers in Simulation, 80 (7) (2010), pp. 1509-1519

[7] A. J. Brizard, T. S. Hahm, Rev. Mod. Phys., vol 79, no 2, pp. 412-468, 2007.

[8] `http://www.nersc.gov/users/computational-systems/cori/`

[9] T. Barnes, et. al., Supercomputing Conference, 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (pp. 43-53), 2016.

[10] D. Doerfler, et. al., International Conference on High Performance Computing (pp. 339-353), 2016.

[11] S. Ku, et. al., Nuclear Fusion, vol. 49 no. 11, Article 115021, 2009

[12] T. Koskela, et. al., 2016 IXPUG US Annual meeting, Argonne, IL, September 19-22, 2016.

[13] T. Koskela, et. al, Intel HPC Developer Conference, Salt Lake City, UT, November 12-13, 2016.

[14] T. Koskela, et. al., Submitted to the International Supercomputing Conference IXPUG Workshop, 2017.

[15] S. Williams, et. al., CACM, vol. 52, no. 4, pp. 65-76, 2009.

[16] A. Ilic, et. al., IEEE Computer Architecture Letters, vol. 12, no. 1, pp. 21-24, 2013

[17] T. Kurth, et. al., Submitted to the International Supercomputing Conference IXPUG Workshop, 2017.

[18] M. Adams, et. al., Journal of Physics: Conference Series, vol. 180, no. 1, pp. 012036, 2009.

A Git repository reference

Table 1: Reference from optimizations discussed in this paper to the commits in the git repository `https://github.com/tkoskela/toypush`.

Optimization	git commit SHA
Optimize veclen to 64	ed1c103c491ff087ffc865b039f852116e14757e
Split and reorder loops	c463c05b7d1fa5fa03f3f10f2e946ff8da63793f
Access grid with scalar index	d42cde2f0dd814cfc0d55b024a502850e5ce8518
Initialize evec inside the particle loop	c42e42c34bb4cbc8c06ed367c257bd1c5212e11a
Access grid with chunks of scalar index	df094efd0a6c93600ea5aee5c59ff8f1b79c6b8a
Declare temporary variables omp private	f07e1154bc6170ebb48580bab6a99f902d6b8b52
Change order of loops in search and remove cycle	9347c131dc177edefa87df3509dac0cde6766b5a