PIC Vectorization with Intel KNC MIC and OpenMP

Viktor K. Decyk, UCLA, USA, and Ricardo Fonseca, ISCTE, Portugal

In this part of the research project, we investigated the use of vectorization with Intel PHI co-processors. The co-processor we are testing is the 1.1 GHz Xeon Phi Coprocessor 5110P (Knights Corner) MIC. We began with the Intel 16.0.2 Fortran compiler and compiled the Fortran version of the OpenMP 3D electrostatic (ES) code (mpic3.f90) without vectorization (-O3 -mmic -no-vec) and executed on the KNC MIC to establish a baseline performance benchmark.

We then wrote the 5 most important procedures using the Intel KNC MIC C/C++ vector intrinsics. For better performance on the KNC MIC, we transposed the particle array so that x locations for all the particles are stored together, followed by the y locations, etc. In addition, the loop which first assigned data to the particle array was performed in parallel. Memory is distributed among nodes in recent NUMA Intel architectures, and the assignment is made when first written. Since subsequent processing of particles used the same parallel loop structure, this avoids subsequent parallel loops where threads process data which resides on a remote node. With OpenMP 3.1, setting the environment variable OMP_PROC_BIND=true was also beneficial.

The results with the Intel 16.0.2 C/C++ compiler on 60 cores (with 240 threads) showed speedups compared to the baseline as follows: Push, 2.8x, Deposit, 2.8x, Sort = 0.9x, Solver 1.1x, and FFT, 1.0x, with an overall speedup for the entire code of 2.0x. The best one expects for the KNC MIC is 16, so the results showed modest improvements. The sort procedure had a modest slowdown. This was because that part of the sorting prefers the particle data not be transposed, whereas the particle calculations prefer the transposed structure. Overall the transposed structure is better, since the amount of time spent sorting is small.

We then attempted to use compiler directives and code rewrites to see how close we could come to the hand-coded results. We followed the same procedure as described in the document VectorPIC3.pdf in the directory vpic3, where one breaks up long loops into multiple subloops with vectorizable parts and parts which run better in scalar mode, storing results into temporary arrays.

The results with the Intel 16.0.2 Fortran compiler on 60 cores (with 240 threads) showed speedups compared to the baseline as follows: Push, 1.7x, Deposit, 1.7x, Sort = 1.2x, Solver 1.0x, and FFT, 1.0x, with an overall speedup for the entire code of 1.4x.

In conclusion, the Fortran compiler based vectorization achieved about 70% of the performance achieved by using vector intrinsics. However, the performance was far from 16x improvement desired.

We continued with the Intel C compiler. The C version of the OpenMP code mpic3.c executed about 20% slower without vectorization (-O3 -mmic -no-vec) than the Fortran version overall.

We then transposed the particle array and made small simplifications in addressing as we had in the Fortran version, The C compiler based vectorization achieved a speedup of 14% compared to the original baseline code versus a speedup of 2.4x by using vector intrinsics. Thus compiler vectorization with the C compiler is ineffective.

We then continued with the 3D electromagnetic (EM) code (mbpic3). We began with the Intel 16.0.2 Fortran compiler and compiled the Fortran version of the OpenMP 3D electromagnetic (EM) code (mbpic3.f90) without vectorization (-O3 -mmic -no-vec) and executed on the KNC MIC to establish a baseline performance benchmark.

We then wrote the 6 most important procedures the Intel KNC MIC C/C++ vector intrinsics. For better performance on the KNC MIC, we transposed the particle array so that x locations for all the particles are stored together, followed by the y locations, etc. The results with the Intel 16.0.2 C/C++ compiler on 60 cores (with 240 threads) showed speedups compared to the baseline as follows: Push, 2.4x, Charge Deposit, 2.6x, Current Deposit, 1.8x, Sort = 1.9x, Solver 2.3x, and FFT, 1.6x, with an overall speedup for the entire code of 2.0x.

We then attempted to use compiler directives and code rewrites to see how close we could come to the hand-coded results. We followed the same procedure as described in the document VectorPIC3.pdf in the directory vbpic3, where one breaks up long loops into multiple subloops with vectorizable parts and parts which run better in scalar mode, storing results into temporary arrays.

The results with the Intel 16.0.2 Fortran compiler on 60 cores (with 240 threads) showed speedups compared to the baseline as follows: Push, 1.8x, Charge Deposit, 1.6x, Current Deposit, 1.7x x, Sort = 1.6x, Solver 1.2x, and FFT, 0.9x, with an overall speedup for the entire code of 1.5x.

In conclusion, the Fortran compiler based vectorization achieved about 75% of the performance achieved by using vector intrinsics. However, the performance was far from 16x improvement desired.

We continued with the Intel C compiler. The C version of the OpenMP code mpic3.c executed about 20% slower without vectorization (-O3 -mmic -no-vec) than the Fortran version overall.

We then transposed the particle array and made small simplifications in addressing as we had in the Fortran version, The C compiler based vectorization achieved a speedup of 25% compared to the original baseline code versus a speedup of 2.3x by using vector intrinsics. Thus compiler vectorization with the C compiler is rather ineffective.

Electrostatic Code, with dt = 0.1

	Scalar	Vectorized	KNC MIC
Push	2.00 ns.	1.20 ns.	0.71 ns.
Deposit	1.56 ns.	0.93 ns.	0.56 ns.
Reorder	0.69 ns.	0.62 ns.	0.80 ns.
Total Particle	4.26 ns.	2.76 ns.	2.07 ns.

Total particle speedup was about 1.5x for the Vectorized code, 2.1x for the KNC MIC code. 240 Threads

Electromagnetic Code, with dt = 0.04, c/vth = 10

	Scalar	Vectorized	KNC MIC
Push	4.40 ns.	2.45 ns.	1.85 ns.
Deposit	4.77 ns.	2.90 ns.	2.42 ns.
Reorder	1.08 ns.	0.68 ns.	0.57 ns.
Total Particle	10.25 ns.	6.02 ns.	4.84 ns.

Total particle speedup was about 1.7x for the Vectorized code, 2.1x for the KNC MIC code. 240 Threads

Table I: Benchmarks for particle processing of Vectorized codes using compiler directives and KNC MIC codes using vector intrinsics, with 128x128x128 grid and 56,623,104 particles

The benchmark codes are available at: https://picksc.idre.ucla.edu/software/skeleton-code/

The Intel 16.0.2 vectorizing compiler is rather primitive, similar to the state of the Cray compiler when it was first introduced in the 1980s. The most important flaw shared by both compilers was that any inhibition to vectorization prevented the vectorization of an entire loop. Later Cray compilers were able to separate such loops into scalar and vector loops. The Intel compiler also has a serious problem in estimating speedups, which can result in a loop vectorization that is slower than the unvectorized version. For the procedure VMPOIS33, which is easy to vectorize, the compiler estimated a speedup of 8 for the main loop, yet there very little actual speedup compared to the scalar loop. It was observed that the compiler would use gather/scatter load/store instructions even when masked contiguous load/stores would have been better. The compiler reported vector dependencies were were false. Another issue is that Fortran90 array syntax such as zeroing out an array qe = 0.0 would execute on a single core. Special subroutines had to be written to have this execute on multiple cores.