PIC Vectorization

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 processors. We began with the serial 2D electrostatic (ES) code (pic2) and wrote the 4 most important procedures using Intel SSE2 vector instrinsics. For better performance on the SSE, 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 compiler on the 2.67 GHz Nehalem i7 showed speedups as follows: Push, 3.4x, Deposit, 2.0x, Solver 2.2x, and FFT, 1.3x, with an overall speedup for the entire code of 3.0x. The best one expects for the SSE2 is 4. We then attempted to use compiler directives and code rewrites to see how close we could come to the hand-coded results.

We began with the Intel Fortran compiler. The initial results were very disappointing. The FFT vectorized, but improvement was modest, about 1.2x. The other 3 procedures did not vectorize at all. We then tried to explore the reasons for the inability to vectorize.

The Deposit procedure did not vectorize because of a well known data collision: two particles could be attempting to deposit to the same memory location simultaneously. One solution to this problem used on the Cray vector computers made use of the fact that each particle always deposits its own weights to distinct addresses. The conflicts occur from different particles. If one makes the number of distinct weights the inner loop, it could be safely vectorized. For the 2D electrostatic deposit, there are 4 distinct weights, which also matches the vector length for SSE2. Thus we process particles in blocks and break up the inner loop into two loops. The first loop processes 32 particles in a block, calculates the addresses and weights, and stores them in small temporary arrays. The second loop reads the stored weights and addresses and deposits the charge, with a short inner loop of 4. The compiler vectorized the first loop, but chose did not vectorize the short loop, and the result was a speedup of 1.8x, close to the speedup of 2.0x obtained from the hand-coded SSE2 code. If one forced the short loop to vectorize with a simd directive, the result was worse.

For the Push procedure, we discovered that if the loop had 2 array indices that used indirect addressing, vectorization was inhibited, with the message: subscript too complex. It was a minor change to rewrite the loop so that the two indirect indices where collapsed into one, and the loop then vectorized. However, a speedup of only 1.1x was obtained. We then proceed to again process particles in blocks and isolate the part of the code with indirect addressing into a short loop. This improved speedup to 1.3x. After further experimentation we discovered that the boundary condition checks were very slow in vector mode. We then added two more inner loops, the particle advance, which stored the co-ordinates into a small temporary array, and a non-vector loop which checked the coordinates and wrote out the result into the particle array. We then obtained a speedup of 2.0x. This was the best we could do, and considerably worse than the speedup of 3.4 obtained from the hand-coded SSE2 procedure. The best strategy was to break up the original particle loop into 4 inner loops, two of which were vectorized and two of which were not. On the Cray a similar strategy was used in the early days, where vectorizable parts were placed in their own loop, and non-vectorizable parts into another. Here the situation is

that it is sometimes better to not vectorize vectorizable loops. The compiler attempts to make this calculation, but it sometimes does a poor job. Eventually the Cray (and Japanese) compilers were able to break up such loops without user intervention, but the Intel compiler does not appear to be that sophisticated at this time.

The last procedure we examined was the Solver. This procedure contains a loop which uses complex variables. The compiler detected a vector dependence on a sum reduction which had a complicated expression. After we simplified the expression into 2 separate expressions, the loop was vectorized, and a speedup of 2.0x, close to the speedup of 2.2x times obtained from the hand-coded SSE2 procedure. Compiler vectorization produced an overall speedup for the entire code of 2.0x.

The conclusion was that in 3 out of 4 cases, we were able to obtain speedup similar to hand-coded SSE2 code by rewriting our loops and using compiler directives. However, the most important procedure (push) was an important exception. Sometimes we had to identify by trial and error where the vectorization was worse and separating out vectorizable and scalar codes into different loops. It became clear when writing SSE2 code that the instruction set is rather primitive and some operations, such as integer multiplication are complicated, and some are not supported, such as accessing non-contiguous blocks of memory. This is why calculating the cost of whether to vectorize is not trivial. Benchmark results for the particle processing are shown in Table I below.

We also experimented with the gnu compilers, but they are much worse in vectorization than the Intel compilers, achieving either very little speedup or actual worse performance.

We then continued with the 2-1/2D electromagnetic (EM) code (bpic2) and wrote the 6 most important procedures using Intel SSE2 vector instrinsics. The relativistic results with the Intel compiler on the 2.67 GHz Nehalem i7 showed speedups as follows: Push, 3.0x, Charge Deposit, 2.2x, Current Deposit, 3.0x, Solver 1.2x, and FFT, 1.1x, with an overall speedup for the entire code of 3.0x. The best one expects for the SSE2 is 4. The algorithm was similar to that implemented for the electrostatic code. The major difference is that the interpolation for the deposits and push operated on the three components (x,y,z) simultaneously. Since the SSE2 vector length is 4, we set the 4th element to zero, the best result for interpolation would only give a speedup of 3x. We then attempted to use compiler directives and code rewrites to see how close we could come to the hand-coded results.

As before, the vectorizing compiler gave very little speedup, unless the code was restructured. The changes we made were very similar to those performed for the electrostatic code. The speedups obtained were as follows: Push 1.7x, Charge Deposit, 1.8x, Current Deposit 1.7x, Solver 0.9x, and FFT 1.1x, with an overall speedup for the entire code of 1.7x. We observed that the original push was able to vectorize but only achieved a modest speedup of 1.2x. Breaking up the vectorizable loop over particles into 4 inner loops, storing intermediate calculations, achieved better results (1.7x). It appears long vector loops are not optimal for the compiler. Writing the Push and Current Deposit Procedures using the vector intrinsics gave substantial performance boost compared to using the vectorizing compiler. For the other procedures, the two approaches

gave similar performance. We also observed that vectorized results on the push were somewhat different with -O3 optimization, compared to non-vectorized results.

The benchmark programs (Table I) had a grid of 512x512 with 9, 427,184 particles (36 particles per cell) and was run on the Intel i7 CPUs (Dual Intel Xeon -X5650), in single precision, running at a clock speed of 2.67 GHz. Intel Fortran is used with -O3. Time reported is per particle per time step. A plasma in thermal equilibrium was simulated for 100 steps in the ES case and 250 steps in the EM case.

Electrostatic Code, with dt = 0.1

	Intel i7(1 core)	Vectorized	SSE2
Push	26.5 ns.	12.9 ns.	7.75 ns.
Deposit	8.3 ns.	4.6 ns.	3.80 ns.
Reorder	0.3 ns.	0.2 ns.	0.32 ns.
Total Particle	35.1 ns.	17.8 ns.	11.87 ns.

Total particle speedup was about 2.0x for the Vectorized code, 3.0x for the SSE2 code.

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

	Intel i7(1 core)	Vectorized	SSE2
Push	62.4 ns.	37.7 ns.	20.10 ns.
Deposit	37.7 ns.	21.7 ns.	13.42 ns.
Reorder	0.4 ns.	0.3 ns.	0.32 ns.
Total Particle	100.5 ns.	59.7 ns.	33.83 ns.

Total particle speedup was about 1.7x for the Vectorized code, 3.0x for the SSE2 code.

Table I: Benchmarks for particle processing of Vectorized codes using compiler directives and SSE2 codes using vector intrinsics.

The benchmark codes are available at: https://idre.ucla.edu/hpc/parallel-plasma-pic-codes.