**Netlib vectors “LCD” optimization**

Tim Prince rev. Mar 23, 2016

**Intro**

These short example loops were picked by the original authors as tests of compiler vectorization around 1990. I tested various Fortran versions, as well as C C++ and Cilk™ Plus versions derived from automated (f2c) translation, using OpenMP to support parallelization (cilk\_for in the one case).

In spite of advances in compilers, source code modifications continue to be needed to gain optimization, comparing the result from machine formatted Fortran 77 against best results obtained with more recent syntax and directives. The original rules set by the authors precluded use of directives, which only with OpenMP have become portable according to a standard. I adhere to the original rules of compiling the drive and test kernels separately, restricting interprocedural optimization to the kernels themselves.

Advertising about getting vectorization reports with selected unrealistic settings continues to be misleading. Note the more realistic recommendation of Maleki, Gao et al. to accept vectorization only with 15% tested performance gain (presumably discounting losses at the shorter tested loop counts). Following discussion of these aspects for Intel® Xeon Phi™ and Xeon platforms, for Intel and gnu Fortran.

Very little source code change, but some conditionals on directives, cover various Intel platforms.

Comments below about array notation are verified both with Fortran and Cilk™ Plus, unless indicated otherwise. Comparison with non-array code is carried out with as many optimizations as possible to approach similar conditions.

**Discussion**

Original source code published by Callahan, Levine, Dongarra is posted at <http://www.netlib.org/benchmark/vectors>

This is colloquially referred to as “LCD” according to publications with the authors’ names in that order.

The present modified test suite was prepared semi-automatically, first using transformation by the UNIXv7 struct (with local bug fixes) followed by ratfor http://wolfram.schneider.org/bsd/7thEdManVol2/ratfor/ratfor.html modified to produce indented Fortran 77. This presents a more uniform style without significant changes in performance obtained with current Fortran compilers. This form is closer to the C code produced by f2c translation <http://www.netlib.org/f2c/> The C code was changed manually to C99-like form, using restrict qualifiers where useful to improve compiler optimization.

Fortran 95, C++, and Cilk™ Plus versions of the kernels were produced by manual translation, using array assignments or STL iterators to the maximum extent possible. The base Fortran version then was produced by using the more effective of the Fortran 77 or Fortran 95 versions.

The test harness portion of LCD was kept in Fortran, as it proved difficult to produce a consistently reliable C translation. The checksums are promoted to double precision so as to distinguish errors reliably, although the tests themselves are performed in single precision (contrary to original authors’ specification).

In the present version, timing is by a C function invoking the rdtsc timer of IA CPUs. The conversion factor for seconds elapsed time is compiled in by macro, so it has to be rebuilt for each platform. It normally is set to the actual clock rate reported in /proc/cpuinfo (using the Cygwin version for Windows). Fortran 2003 system\_clock with 64-bit parameter also may be satisfactory on linux (not on Windows); omp\_get\_wtime is the closest to a satisfactory portable timer.

**Recent related publication**

Maleki, Gao et al. <http://polaris.cs.uiuc.edu/~garzaran/doc/pact11.pdf> published a sponsored study including IBM Power7 and Intel original core-i7 including a modified version of the original LCD, translated to C by f2c. They chose to delete several cases which depended on non-unit-stride optimization, even though that was already a subject of intensive development work in the icc and gcc compilers. They admitted source replacement of if()..else by ? selection (apparently needed by the XLC compiler as well as gcc) and splitting of if..else for() loops, as well as float \* \_restrict definitions and alignment assertions. A criterion of 1.15x speedup was taken for successful auto-vectorization.

**Discussion by test case**

**S111**

For architectures other than MIC, ifort and Cilk™ Plus needed novector and unroll() directives prior to 15.0. gcc 6.0 began to perform inefficient vectorization around Nov. 2015. This can’t be corrected by removing \_\_restrict or adding omp simd safelen(1) but can be fixed (in C or C++ only) by adding \_\_attribute\_\_((optimize("no-tree-vectorize"))). Vectorization in gfortran is prevented by a volatile declaration.

**S112**

Latest gnu compilers for sse4 and avx optimize this effectively; Intel compilers prior to major version 15 need sse intrinsics, which are compiled to avx-128. SSE3 was the first CPU to work adequately with unaligned loads; this doesn’t use sse3 instructions, but the intrinsics didn’t perform on Pentium4. Microsoft achieves a large gain with \_\_restrict pointer (which can’t be edited by cpp), although it doesn’t achieve vector performance.

**S113**

Intel C doesn’t recognize loop invariant operand unless written as explicit temporary, although it vectorizes. Then the vectorized remainder loop uses AVX-128 for C, AVX-256 for Fortran.

**S114**

Ifort vectorization by simulated gather is OK on Xeon, but on MIC KNC disabling vectorization gains apparently due to poor prefetch coverage with vectorization. OpenMP is needed, even though auto-parallelization looks trivial. Use of a temporary array defeats the advantage of vectorization for SSE, but the legacy simd directive corrects that even when using Fortran array notation. Poor cache locality produces performance variations. Intel C/C++ were spending 50% of the time unnecessarily in a scalar version of the loop, even though it vectorized successfully. This is corrected by pragma omp simd.

**S115**

dot product rewrite improves performance and accuracy. Gcc gives incorrect results at –O3 with AVX when attempting optimization by omp simd reduction, but –ffast-math without OpenMP is good.

**s116**

gfortran fails to optimize original source unrolled version

**s118**

dot product promotes vectorization but MIC KNC has problems with strided prefetch. Intel C++ doesn’t optimize as well as ifort or gnu compilers. For Intel 15 beta compiler to optimize, the subscripting and loop direction have to be written explicitly for simple stride +1 on that operand.

**s122**

“vector dependence” resolved by array notation or pragma simd (deficiency in analysis of traditional syntax). Vectorization is inefficient on corei7-4, unless the loop is reversed to turn the -1 stride to +1, but needed on MIC. Benchmark is run with variable stride set to 1 so cache performance should be same as stride 1 compilation. Intel C++ optimizes only with vectorization set by pragma together with the stride 1 reversed operand; surprisingly, icc vector performance may be same as ifort non-vector (but gnu scalar performance is better). Gfortran 6.0 also vectorizes array assignments at a loss, by a combination of insertps and extractps scalar moves and vpermd reversal. In March 2016, gfortran 6.0 began vectorizing the f66 version as well, still at a (small) performance loss. The loss with vectorization may be associated with suppression of unrolling to avoid excessive code expansion. Vpermd reversal might be avoided by reversing the loop in source code, but this doesn’t appear to make a significant difference.

**S123**

Not vectorizable, Intel compilers don’t perform well.

**S124**

Preference for array notation on MIC only; alignment assertion helps some CPU targets. This can be vectorized in Fortran 77 code with gfortran by the archaic use of sign() intrinsic, but merge (or C ? operator) are superior for both performance and reliability. Fused multiply-add reduces the advantage of merge() over the sign() trick.

Omp simd aligned(…) is not as effective as ifort vector aligned. Neither Intel nor gnu compilers appear to get much benefit from the simd aligned clause, even though the Intel opt-report confirms recognition of it. Similar problem in s161, s278, s2710 submitted as Intel IPC problem report.

**S125**

Resolve spurious outer loop indexing dependence, specify OpenMP. This is the highest memory bandwidth case of the entire LCD suite, and has 50% unaligned loads, yet 32- or 64- byte alignment is important (from corei7 on). In a larger size, non-temporal store becomes effective. Windows Intel64 Fortran compiler set streaming stores from –Qopt-streaming-stores default but C/C++ need #pragma vector nontemporal. Performance on most platforms peaks at a lower number of threads than with non-bandwidth-limited benchmarks (1 thread on Haswell laptop, 1 thread per core on MIC KNC).

**S126**

Break indexing dependency between outer loops by calculating inner loop start. AVX-256 does not gain over AVX-128 or SSE except on AVX2 CPU, due to 50% unaligned loads. AVX2 compilers still expand the fmadd (fma3) C intrinsic to separate AVX multiply and add instructions. Gnu compilers don’t optimize with combined threading and avx, but do vectorize for avx with the loops nested for a single large stride operand (choosing simulated gather and fma3 instructions). Libgomp on Windows appears to have prohibitive overhead for vector parallel outer loop. On the avx2 4 core platform, the inner loop vector version may be faster than parallelization even with the Intel compiler outer loop vectorization, and about even with the intrinsics strip mined version. Only ifort manages a vector parallel version which optimizes performance on 4 cores.

VTune reports performance is limited by DTLB miss in the outer loop vector parallel version.

The cilk\_for \_Simd clause introduced in Intel C++ 16.0 might appear to be applicable, but it doesn’t match omp parallel for simd or cilk\_for with intrinsics.

**S127**

Only recently did many compilers succeed in optimizing this. Intel C 15.0.1 lost the C code optimization again, never having optimized the CEAN version. Optimization was restored in 16.0 beta update 1. Ifort doesn’t optimize the version using forall as gfortran does now.

**S128**

Intel compilers choose to vectorize for AVX but that performs even worse than scalar with unroll. A way of dealing with this is to add C++ \_\_restrict extension only for MIC. A problem report was filed as to the Intel compilers not complying with #pragma simd safelen(1). There is sometimes an improvement in performance by helping the compiler “linearize” by calculating stride 2 subscripts from i\*2, i\*2-1.

**S131**

Vectorization without directive depends on compiler recognizing direction of data overlap avoiding read after write. Gfortran didn’t align loop body, so apparently loses from unroll by 4.

**S132**

Fortran can optimize without directive by relying on bounds restrictions.

**s141**

switch loop nest to stride 1 inner, specify thread work balancing for triangular loop (explicit or by schedule). Dynamic scheduling may optimize without explicit balancing when running on a unified cache. gfortran only began vectorizing the openmp parallel loop with array assignment during 4.10 development. Gcc/gfortran performance still appears to be non-repeatable, possibly due to inconsistent loop body alignment, with gfortran peak performance improved by changing the align\_loop\_max\_skip parameter from default 10 to 12.

**S151**

Depends on in-lining for compiler to resolve direction of data overlap. Intel C/C++ optimization is blocked by passing a scalar by reference. Gfortran optimization depends on use of internal procedure.

**S152**

Optimization depends on in-lining to remove procedure call from loop. Intel C/C++ requires pragma to optimize; gcc is better without pragma

**s161**

The if and else sections must be split into separate loops for full vectorization with the tested compilers, although a single loop is more efficient without vectorization. The objectionable dependency in the single loop may be shifted from c(:) to a(:) by peeling and realigning. Gfortran 6.0 optimizes the split where version by a form of partial vectorization which conditionally executes an entire simd width block using masked moves.

**s162**

Fortran array notation resolves aliasing partially by ifort temporary array allocation. omd simd didn’t work at the time of writing for ifort, although legacy simd did (PR filed). Compilers don’t derive assertion about direction of overlap from control flow (the question posed by the original benchmark authors).

**S171**

Array notation or ivdep resolve partially for Intel. Gnu compilers optimize original by strength reduction without vectorization, but Intel compilers comment about variable stride even though some optimizations take account of loop invariance. Latest Intel compilers are able to optimize s171 and s172 for scalar, if #pragma omp simd safelen(1) is set, but omp simd without safelen is needed for MIC.

**S172**

Intel Fortran formerly needed novector and unroll directives, except for MIC. Intel Cilk array notation is better than C on MIC and equal to omp simd on AVX, but doesn’t match gcc non-vector.

**S173**

Intel compilers don’t propagate constants in order to recognize k == n/2 inside loop, and still don’t optimize fully in spite of vectorization. Vector unaligned directive is a slight improvement, as the read operands are all aligned but are misaligned by normal peeling for store alignment.

**S174**

Intel (prior to 15.0) needed legacy simd directive to suppress temporary. Gfortran doesn’t align the loop as gcc, ifort, and icl do. Icl no longer needs #pragma.

**S175**

MIC requires array notation or simd directive to promote gather-scatter. Gfortran (for array notation but not with DO loop) and ifort use temporary with array assignment. Original source has -1 where it should be –inc (a clue that is the actual test value). Intel C++ still optimizes the scalar version poorly enough that vectorization recovers a bit for AVX2 target. Apparently, AVX512 will vectorize without directives beyond \_\_restrict.

**S176**

For threaded performance scaling, this should be organized in dot product fashion, but gnu compilers didn’t vectorize this until 4.10 (and then lost it), and then avx2 performance was limited by latency of fma addition to a single parallel register. Intel C also didn’t vectorize the C omp simd reduction for MIC successfully until mid-2014. Beginning with Intel Fortran 15.0, the stride -1 is optimized by shuffle, more efficiently (at least for AVX2) than by making a reversed copy of the array which is used repeatedly, as is necessary for C++ optimization. Due to the additional fixed overhead of tree reduction in the dot product, the dot product form is not competitive until the problem size is large enough to benefit from parallel, and then the performance per thread still suffers due to the tree reduction time not benefiting from parallelism. In the Cilk™ Plus version, as the only way to avoid performance degradation from multiple cilk\_for workers on small problems is to make a separate branch with for replacing cilk\_for, the example short loop version uses the original netlib version without dot product. Each inner loop accumulates on top of the previous outer loop iteration. With the Intel compilers, the loop invariance of the scalar array element in non-parallel version isn’t recognized unless it is copied to a local temporary, a relatively minor performance issue where the local avoids broadcasting across the register prior to each use.

AVX2 performance of Intel compilers is in question at smaller loop counts. The practice of unroll-and-jam so as to gain register locality of one operand is done strangely by Intel with the second sum not using the same riffling (multiple parallel sums) as the first. More than necessary riffled sums increases latency of final tree reduction both for short loops and in multi-thread scaling of longer loops. Apparently (for avx2), 2 8-wide sums are combined inside the loop, avoiding fma to reduce total latency, with 2 8-wide sums produced by fma, leaving 3 8-wide sums to be combined in the epilog.

Among Visual Studio compilers, only VS2012 is vectorizing (only the version optimized for single thread), only with /arch:AVX (AVX128 only).

This could be used as a test case for array reduction, which was added for C in OpenMP 4.1. Intel auto-parallel demonstrates the array reduction, effective for a small number of threads. Array reduction without parallelization is best for current MSVC and gnu compilers.

**S211**

Avoid vector reload misalignment by explicit peeling for alignment or by using omp or nofusion directive to prevent fusion. Ifort’s seemingly logical strategy of interleaving unrolled iterations doesn’t work out.

**S212**

Loop fusion doesn’t show consistent gains but Intel hits target of optimization for loop length 100.

**S221-S222**

gfortran needs –O3 to take advantage of speculative execution hidden in latency of loop carried dependency. Gcc/g++ are buggy with full optimization and –fopenmp even though not used here. Splitting into separate assignments to promote partial vectorization is significantly better on MIC in spite of loss of register locality, assuming L1 locality is preserved. Intel C appears to need explicit unroll directives to optimize the loop carried dependency without explicit scalar replacement.

**S231**

**I**ntel compilers vectorize by switching loops, but the effective way for 8 or more threads in C is with omp parallel simd. ifort performs outer loop parallel vectorization better without simd clause, maintaining performance at 1 or 2 threads while supporting up to 3 times performance on many threads. For C compilers without omp simd, conditional compilation nests the loops to favor auto-vectorization (or at least cache locality) over threading. Gcc is ignoring the simd clause for outer loop vectorization in this and the following cases, and so the single thread vectorized version is superior up to a larger number of cores. Cilk\_for depends for outer loop vectorization on explicitly breaking the loop into an appropriate number of chunks, which seems inconsistent with the idea of supporting arbitrarily varying number of workers. Cilk\_for parallel speedup, even on 61-core MIC, doesn’t show advantage over single thread vector speedup.

**S232**

80x performance gain by parallelization on MIC even though there is no useful vectorization: explicit unroll and jam + OpenMP with work balance (explicit, which is better on multi-cpu NUMA, or by schedule). icc depends more than gcc on explicit scalar replacement, as icc doesn’t observe the unroll4 option.

**S233**

Split loops, one outer loop vectorized by Intel with omp parallel simd, the other OpenMP parallelized with inner loop serial. As the loop with no vectorization may be slower, placing it first with nowait allows threads with no more work to proceed into the other loop, improving cache locality. Compilers and relative numbers of cores and vector widths affect this balance. The ifort AVX2 vectorization of the large strided loop is best suppressed by directive. Microsoft reached satisfactory AVX performance in recent updates, using OpenMP 2.

Cilk\_for \_Simd clause for the outer loop vectorization produces a report of gather vectorization, which shows no advantage.

**S234**

Original automatic translation to C didn’t make a local value copy of for end limit, stopping Microsoft from optimizing (but Microsoft fails to recognize stride 1, so doesn’t vectorize).

**S235**

The Intel compiler implementation of omp parallel outer loop vectorization distributes vectorization over single and double nested loops. Typical of cases with outer loop vectorization, cache and TLB misses seem fairly high at the beginning of strip mined inner loop, and vector speedup over scalar parallel is relatively small.

Ifort outer loop vectorization without omp simd clause is fairly efficient at small numbers of cores, while icc (with simd clause) requires a large number of cores to make parallelization pay off.

VTune reports performance limited by data sharing among threads in the outer loop parallel simd version (false sharing?).

This is the only case of the series where Microsoft auto-vectorizes by taking advantage of the conditional non-parallel code for compilers with only OpenMP 2.

**S241**

Split expression with explicit temporary to allow full before- and after- modification local vectors according to original authors’ comment.

**S242**

Parenthesize for pipelining, leaving loop carried dependency for last. gcc shows a benefit for splitting into vectorizable and non-vectorizable loops (as did earlier icc versions).

**S243,s244**

Eliminate redundant assignments which create false dependency. Embedded assignment permits Cilk™ extended array notation to gain the register re-use of plain C.

**S252**

Conditional compilation with array assignments or equivalent for SSE4 (which was the first architecture to support misalignment efficiently)

**S253**

Alignment assertion improves MIC prefetch efficiency (?). Cilk™ Plus version will fail on MIC if not all arrays are aligned; performance relies on aggressive fusion. Ifort 16.0 improves do concurrent to where it is the fastest choice, presumably because it removes the requirement for an explicit temporary. It avoids use of fma. gfortran 4.10 (but not current version) vectorizes the optimized version with merge and temporary. Gfortran 6.0 began to give adequate performance in the do concurrent version around Nov. 2015. Gcc gives good performance for the temporary with repeated ? operator, although an operand is reloaded for redundant compare. ICL generates more compact but not faster code for that version than for the multiple assignment under if. None of these versions get loop alignment.

**S254**

Vectorizable by peeling to eliminate loop-carried dependency (without peeling, it’s favorable to pipeline compilation). Fortran cshift doesn’t perform well.

**S255**

Omp simd lacks firstprivate, but out-performs explicit peeling on Intel compilers. Microsoft vs2015 and gcc can vectorize AVX2 with explicit peeling.

**S256**

Explicit correction of loop nesting with directive to stop splitting off of outer loop stuff, so as to promote cache locality.

**S257**

100x speedup on MIC by suppressing repeated stores to shared array, OpenMP with nowait and single, avoiding introduction of partial vectorization in favor of register locality in the single region, also avoiding outer loop strided vectorization which degrades prefetch and consumes more memory bandwidth. Explicit remainder loop for outer loop can be started by a thread which has completed its chunk of work in the parallel loop. Updating the 1D array in parallel loop without single would raise a race condition, which could be avoided at significant expense by setting it as firstprivate lastprivate.

Unroll(2) appears to be optimum. Due to the long dependency latency, further unrolling might be expected not to be useful, and Intel compilers don’t align these non-vectorizable loops.

**S258**

The practice of expanding (s+1.)\*aa(i,1) to s\*aa(i,1) +aa(i,1) for improved accuracy is particularly effective for fma targets. gfortran chooses not to use fma unless parentheses are used to prevent re-association. Microsoft (when fma is enabled by –fp:fast) also inserts the + 1f to prevent fma optimization. Such parens are ignored by gcc –ffast-math, and, surprisingly, by Microsoft. Re-writing for partial vectorization, using a temporary array, improves performance. Intel compilers require directives to prevent fusion which improves data locality but prevents partial vectorization.

**S261**

Fix vector misalignment explicitly (MIC compiler takes care of it). Prior to Barcelona and core-i7, simd vectorization had to be done with separate loops, avoiding fusion which would incur a store-forward stall implied by reading back the updated vector at different alignment.

**S271**

Faster (at least when there’s instruction level support for max) to clip the input operand explicitly rather than make the write-back conditional. While gcc will optimize fmax() with –ffinite-math-only setting, lack of portability and failure of g++ and MSVC++ to optimize std::max leave ? operator as a solution (which fails to optimize in Cilk™ Plus/C99).

**S272**

Gnu compilers vectorize (for avx only) with explicit loop split, although that’s precisely what is implied by forall. Intel compilers do best without do concurrent, by fusing the split loops, when the merge conditions are written to be identical.

**s273, s274**

Gcc-4.9 avx2 vectorizes with 1 ? operator assignment moved to last (also organized to allow fma). Take advantage of gfortran/ifort/icpc optimization of max intrinsic operator, using fmax for same purpose with gcc/g++. S274 shows largest gain of any vectorized cases for use of “fma” fused multiply-add ( –fp-contract=fast –march=core-avx2 set for gnu 4.9+ compilers). Placing the d[] \* e[] product inside the conditional need not raise a compiler concern about conditional exceptions, as the expression appears earlier without conditional. For MIC, the combination of alignment assertions and encouraging fma by arranging the conditional can show 40% performance gain in s273.

Due to g++ apparently not taking omp simd to permit vectorizing a floating point expression inside ? operator, even though the expression was already used unconditionally, a local temporary assignment is introduced. Intel compilers are remarkably slow with such a substitution, which misses a potential fma.

Intel and gnu compilers are seen not to issue a code body alignment directive, which may be a factor in occasional poor performance when testing variations.

**s276**

split loop to eliminate conditional based on loop counter

**s277**

change statement order to match vector update before use

**s278,279,2710,2711,2712**

alignment assertion, minimize conditional sub-expression. If using Fortran where(), check advantage of splitting loops to where() and where(.not. ….) which can double performance (but still not optimum). All conditionals without else are replaced by merge/?. For C/C++, loop invariants are made explicit. Gnu compilers need the unconditional assignments split out and the simple conditionals minimized, but still fail to vectorize the more difficult conditionals.

**s2711,2712**

By expressing the conditional as a selection of 0 operand, the need for Intel vector align directives is eliminated, and full advantage can be taken of fused multiply-add. Gnu compilers show 50% gain of fma, which is exceptional for a case without explicit multiply-add latency and with multiple memory references.

**s281**

omp simd directive or array assignments. Correctness of vectorization seems doubtful unless loop is split, as the original serial loop doesn’t use any original elements of a[] below the median element, but vectorization without split will run across and read some elements prematurely. Shuffles are superior to gather for stride -1, particularly on AVX2. VS2015 will vectorize only the 2 assignments up to the median element, only if they are split.

**s291**

peeling is best portable optimization but is slower than “pipeline” for MIC

**s292**

Promote scalar replacement by writing loop private temporaries for legacy simd directive. Compilers vectorize but do not fully optimize the peeled version with 4 memory accesses, 2 of which are repeats with alignment shift.

**s293**

array assignment or explicit temporary copy needed to resolve overlap while permitting aligned store

**s2101**

OpenMP and strided vectorization consistently effective only for Intel Fortran/C; 2x performance from combined vectorization and threading.

**S2102**

Correct loop nesting, merge values to eliminate over-store, use OpenMP. Cilk™ Plus idioms also work.

**S2111**

Explicit scalar replacement is needed by all compilers except ifort and msvc++. Reorganization for strided vectorization (as expected by original authors) doesn’t pay off.

**S311**

Omp simd may be used on some of these cases to preserve optimization when setting icc –fp:source for standard compliance; on others, and with gcc, it ruins optimization.

**S313**

As with vdotr, gnu compilers perform better with –march=corei7-avx to prevent using the higher latency fma instructions on AVX2, in spite of using twice as many floating point instructions. AVX without riffling performs midway between AVX2 with full riffling (Intel compiler) and without riffling. Riffling meaning interleaving multiple parallel sums so as to avoid latency of repeated sums to the same register.

**S314,316**

Maxval/minval needed for gfortran; macro replacement of max/min for g++/MSVC. Intel 15.0.0 compiler needed omp simd reduction (which didn’t work in 14.0); later Intel (and gcc) vectorize without omp (when /fp:fast or –ffast-math are set). Penalty for not setting aggressive modes for vectorization is not as great with Intel as with gnu compilers. It’s not clear that aggressive optimization can affect the result.

Neither icc nor gcc riffle the results. Clang riffles these cases into 4 independent simd parallel reductions, showing a 2x performance gain over ICL or icc on Haswell laptop.

**S315.S318**

Not optimized by gnu or Microsoft compiler. Deficit for non-vectorization of s318 is smaller, as it is strided. #pragma omp simd reduction… lastprivate… helps gcc with s318, but is broken for s315 with gcc.

**S319**

Separate array assignments don’t fuse well, in part because there is no need to riffle the sum, when the array elements are added first (contrary to original). Excessive riffling does produce more accuracy. C++ with 3 transforms, a temporary, and accumulate would be ridiculous.

**S3110**

Inner C loop may use directive based optimization (the legacy directive may work with icc). Intel Fortran vectorizes maxloc without directive (prior to 15.0 and in 16.0.1, only with the non-standard old\_maxminloc option). Maxloc is cut down to rank 1, using OpenMP 3.1 reduction or 2.0 critical for outer loop. Outer loop OpenMP 3.1 tree reduction is important for larger numbers of cores so is a potential advantage over OpenMP 2. Ifort is able to parallelize rank 2 maxloc with omp parallel workshare, but gets only 30% of the performance with 2 threads. Ifort under-performs Intel C, possibly due to excessively aggressive unrolling (which seems not to be controlled by compiler option or directive).

**S3111**

Fortran masked sum is syntactically correct, but compilers don’t recognize the special case which can be optimized with max instructions. Nor do gnu or Microsoft compilers recognize the optimization (the latter on scalar basis) unless ? operator (or fmax with –ffinite-math-only) substitution is made in place of std::max. VS2015 reduced the penalties for std::max.

Intel 16.0 corrected the missed vectorization of C ? operator, so, for example, Cilk reducer became competitive in C as well as C++.

**S3112**

Corresponds to C++ partial\_sum(). In plain C or Fortran, speedup of more than 40% is possible by unrolling to produce multiple results per iteration not limited by addition latency. It’s difficult to see how the gnu parallel partial\_sum could ever be useful. This is the prefix sum apparently beloved of computer science parallelization classes. Apparently, much larger speedup can be demonstrated with expert intrinsics vector coding. Recent gcc 6 builds show a gain (matching Intel) for compiler unrolling on top of the explicit unrolling.

**S3113**

abs(float) is a C99 and C++11 intrinsic supported in recent Visual Studio. No need for the legacy f2c or other traditional C/C++ expansion. Abs() doesn’t work here with gcc –fcilkplus. Intel compilers removed requirement of 15.0.0 for omp simd reduction.

**S322**

Intel and gcc C/C++ compilers de-optimize by ignoring parens. This is solved by ICL 16.0 option –Qprotect-parens and removing gcc –ffast-math option. Gcc has regressed in its ability to perform scalar replacement optimization on the loop carried dependency, but none of the tested C or C++ compilers optimizes all such cases consistently, while all current Fortrans do optimize without explicit scalar replacement.

**S323**

Fma is disadvantageous on the critical path due to loop carried serial dependency. Gfortran will avoid it if the multiplication is parenthesized. Both Intel and gnu C compilers benefit from explicit scalarization of the loop carried dependency.

**S331**

Runs faster backwards and not vectorized

**S332**

Latest Intel compilers vectorize effectively using plain C code, Fortran do..exit or (less effectively) maxloc (with old\_maxminloc setting), or Cilk™ \_\_sec\_reduce\_max\_ind. “not found” case after C break or Fortran exit can be set by checking whether the induction variable has been stopped within the search range. An explicit temporary array is required to vectorize the maxloc or max\_ind, rather than putting an anonymous array expression inside the reducer (same thing for s415). The reload of the temporary is suppressed by fusion, but the loop can’t exit early; hence the desire for an f2008 findloc intrinsic. Integer data type for temporary is faster in AVX2 with Intel compilers, beyond loop count 100, but real data type is required to vectorize prior to AVX2. Integer data type is OK for compilers which don’t vectorize the max index reducers (gnu or MIC).

**S341**

Effective use of MIC pack instruction and Fortran intrinsic.

**S342**

Speculative loads used only by Intel C++ appear unproductive.

**S343**

PACK version is slower on all tested architectures.

**S351,352,353**

Expose deficiencies of Intel compilers on source-unrolled code. Intel compilers recovered some long-lost performance with 15.0 beta test. Gcc loses performance on s352 with –ffast-math; Intel loses on host without the combination of protect-parens and no vector.

**S412**

Exhibits failure of Intel compilers to recognize alternate forms of vectorizable loop

**S413**

Peel for alignment of one multiply-read array, MIC compiler takes care of it

**S414**

Current compilers can recognize counted loops with while constructs

**S415**

Split into search for termination value and loop count prior to data modification, use counted loop for data update. See comments about vectorizing linear search s332.

**S421**

EQUIVALENCE test, author’s comment seems confusing. Gfortran vectorizes with temporary and memcpy but the temporary can be avoided by the patch enabling omp simd. Ifort uses temporaries for most of these equivalence cases when using array assignments.

**S422**

Omp simd or legacy equivalent needed by Intel compilers. We aren’t certain whether gcc is taking a chance or is using additional information present in this test case but possibly not in more common situations. Gfortran needs the patch enabling omp simd for full performance.

**S423**

Ifort needs legacy simd to suppress temporary, but it still doesn’t reach full performance.

**s424**

!$omp simd safelen promotes multi-versioning with vector version taken at run time. Safelen parameter had to be adjusted to architecture in ifort prior to 14.0.2.

**S431**

A Fortran parameter constant was eliminated during f2c translation to C

**S432**

The constant (not protected against modification) with value 0 is still in the source code, all languages.

**S441**

Alignment and ignore exception assertion. G++ 5.2 vectorizes

**S442**

Small gain for OpenMP (but not cilk\_for) only on Intel library, loss for vectorization

**S443,vif**

Alignment and ignore exception assertion But why does gfortran prefer if to merge in the one case of vif (no exception possible)?

**S451**

OpenMP (not so useful for MIC, as it reduces vector length) gain for vector math library. Ability in C99 and C++98 to treat math functions as generic, in order to optimize float data type, seems to have gone out of fashion. Vector nontemporal directives show a gain, but it’s not clear they are used consistently.

Even with Intel 16.0 compilers, cilk\_for was disadvantageous relative to single thread vectorization on dual core CPU. This may be a deficiency in taking advantage of svml (short vector math functions)/

**S452,453**

Simple cases using loop index in calculation where forall is fully effective. Fortran array constructors don’t perform well. Cilk™ Plus \_\_sec\_implicit\_index() requires (int) cast on account of lack of hardware support for (float) case from unsigned 64-bit.

**S471**

Inconclusive test attempting to show overhead of calling separately compiled function, performance not reproducible.

**S481,482**

Current compilers don’t vectorize loops with conditional exit until they are reformed in accordance with ifort/icc vectorized search criteria, requiring just one conditional exit and all arrays aligned the same.

**S491**

Assert vectorization contrary to cost model for Xeon (omp simd works)

**S4112,4113,4114,4115,4116**

Cases with indirection, which are sensitive to optimum level of compiler unrolling, particularly if attempting vectorization, and show advantages of gnu compilers over Intel when using avx2. An evident reason is that fma may be used as it doesn’t run into latency even without riffling the sum reductions. Intel C++ 16.0 miscalculates the effect of vectorization in s4113 so it is better to remove \_\_restrict to keep it scalar and avoid the large performance loss with AVX2 vectorization. It seems that it should not vectorize anyway on account of possible repetition of scatter indices. AVX512 has vconflict instructions to deal with repeated indices in scatter.

**S4117**

CEAN notation allows for array assignment. 32-bit right shift replacement for divide is particularly important to avoid performance loss on MIC. Fortran shift notation generates more compact code than divide by 2. Gfortran doesn’t vectorize the shift version, but performance comes out even.

**S4121**

A test of obsolescent Fortran syntax (statement function)

**Va**

Recent compilers may multi-version, with in-line code to accelerate the simple case and avoid the overhead of automatic memcpy() substitution.

**Vag**

Gather gives Intel compilers opportunity to take disadvantage of corei7-4 instruction support which is worse with default unroll

**Vas**

Vector always assertion may be wrong if it promotes scatter instructions when there may be repeated indices.

**Vif**

A rare case in which gnu compilers perform different forms of vectorization with if or merge/?. The latter appear closer to optimum with latest gnu versions, in view of their tendency to make each masked move group conditional on at least one bit set in mask.

Vpvpv

Gcc error propagation is worse with –ffast-math.

**Vdotr**

Lower performance of gnu compilers is due to not “riffling” (using more batched sums than vector width) plus fma when available. The comparison is favorable to gnu when including indirection (s4115).

**Vbor**

Alignment assertion + explicit use of Fortran alternative expression evaluation rules to reduce number of flops. Without source rationalization, Fortran could not be compared against C. Intel compilers require all common subexpressions to be parenthesized and an option to observe them to improve performance; otherwise there are a large number of extra additions on critical path. 2 of the eliminated add instructions are replaced by equally time-consuming moves. Other C or C++ compilers recognize common subexpressions by left-to-right evaluation.

**Summary**

Current compilers generally prefer to vectorize conditionals by using Fortran merge or C ? at the earliest possible point in expression evaluation. Gnu compilers prefer to have such a conditional only in the last assignment in a loop body, with if working only where there is no arithmetic.

Ifort goes part way toward implementing masked do concurrent with local merge, but it seems usually it will be possible to improve on it.

Fortran compilers prefer max or min intrinsics, where they translate directly to the instruction set. C++ compilers need macro substitution according to the preference of each compiler (std::max for Intel, ? for MSVC (no vectorization there), fmax with –ffinite-math-only for gnu, ……)

Vectorizable loops with if..else not directly replaceable by merge/? are frequently best split with the conditional expression repeated so it can be recognized as common if the compiler is able to re-fuse.

Gfortran and ifort each have several cases where an unnecessary temporary result and memcpy is included in the f90 array assignment case. The cure is to switch to DO loops with legacy simd directive for ifort; for gfortran, one case requires omp simd. For the two cases where ifort requires legacy simd but gfortran requires OpenMP 4, ifort problem reports have been submitted.

Gnu compilers differ from Intel in the implementation of omp simd. Intel takes the directives as over-riding compiler options, so that it may be possible to vectorize in spite of a standards compliance setting which otherwise would prevent it. Gnu compilers treat the omp simd directive more like an IVDEP directive, which doesn’t replace the requirement for options such as –ffast-math (-ffinite-math-only) to enable vectorization of conditionals. A PR was filed for a case where omp simd kills vectorization otherwise enabled by –ffast-math.

**Implementation**

Gnu make compatible Makefiles are currently maintained:

Makefile.intellinux (Intel x86\_64)

Makefile.windows (Intel/Microsoft X64)

Makefile.cygwin (cygwin64)

Makefile.gfortran (gnu x86\_64)

Makefile.micn (MIC KNC native linux)

On HyperThread enabled systems, running 1 thread per core is likely to be optimum.

On multiple CPU Xeon systems, OMP\_PROC\_BIND=close should be sufficient when HT is disabled.

MIC systems typically require

Ulimit –s unlimited

KMP\_PLACE\_THREADS=59c,2t (leave 1 or 2 cores for MPSS)

CILK\_NWORKERS=118

OMP\_PROC\_BIND=close

Compilers primarily supported are gcc/g++/gfortran 4.9,4.10,fortran-dev and Intel 14.0/15.0 (with specific ifdef for each, as well as some by target architecture).

Example:

Make –f Makefile.cygwin lcd\_ffast lcd\_cfast lcd\_cxx lcd\_f90

Builds the mixed Fortran 77/90/omp4, C, C++, and Fortran95 versions.

Gfortran –c lcdmod.f90 may have to be done first (why not automatic in Make?)

For windows runs, a way of setting up cpuinfo to be opened by Fortran is to install Cygwin procps and run ‘cat /proc/cpuinfo > /cygdrive/c/proc/cpuinfo’ The CLOCK\_RATE in Makefile can be set to be consistent with the one in cpuinfo (6 zeros appended).