Lawrence Berkeley National Laboratory

LBL Publications

Title
An Investigation of Compiler Vectorization on Current and Next-generation Intel Processors using Benchmarks and Sandia's SIERRA Applications

Permalink
https://escholarship.org/uc/item/05x9132w

Authors
Rajan, M
Doerfler, DW
Tupek, M
et al.

Publication Date
2015-04-29

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

An Investigation of Compiler Vectorization on Current and Next-generation Intel Processors using Benchmarks and Sandia's SIERRA Applications

Mahesh Rajan[1], Doug Doerfler[2], Mike Tupek[1], Si Hammond[1]
[1]Sandia National Laboratories, [2]Lawrence Berkeley National Laboratory
mrajan@sandia.gov, dwdoerf@lbl.gov, mrtupek@sandia.gov, sdhammo@sandia.gov

***Abstract*— Trinity, a Cray XC40, with over 19,000 nodes utilizing Intel Haswell and Intel Knights Landing (KNL) processors is the first of NNSA's new Advanced Technology Systems (ATS-1) procured by ACES, the partnership between Los Alamos National Laboratories (LANL) and Sandia National Laboratories (SNL). Phase-1 of Trinity with only the Haswell nodes is anticipated to be installed in mid-2015 and Phase-2 with KNL nodes in the spring of 2016. Effective vectorization of our applications, to take full advantage of the AVX2 vector units on each core of Haswell and the two 512-bit vector SIMD units on each core of KNL, is an important performance goal on Trinity. We carry out a systematic study of vectorization using Cray, Intel and GNU, compilers. The study includes micro-benchmark mini-applications and a set of important kernel operations from Sandia's SIERRA Mechanics applications suite. Cray compilers on Haswell give the lowest value for the total (sum) time of the 151 loops in the TSVC vectorization benchmarks, achieving on the average close to 3X gain in performance. For the LCALS benchmark Intel compiler out performs Cray and GNU in similar measures. For the SNL SIERRA Mechanics complex compute kernels like eigenvalue and material model computations, we present approaches to achieve significant (up to 50%) performance improvement. This study highlights the benefits and limitations of different compilers and the alternate approaches we may need to take full advantage of the promised performance with newer SIMD vector units on Intel processors.**

Keywords; KNC, KNL, Haswell, Vectorization, performance optimization

I. INTRODUCTION

An important element in extracting optimal performance out of the current generation of CPU architectures and systems requires us to take advantage of the wide SIMD registers. Approaches to achieving effective vectorization can vary in effort and complexity starting from simple use of compiler switches, calls to optimized library functions, writing assembly code or calling intrinsic functions that mimic assembly instructions. For HPC systems the effort in tuning for the CPU typically benefits performance on hybrid systems with accelerators like the Xeon Phi or NVIDIA GP-GPUs. For complex multiphysics codes suites like the SNL SIERRA Mechanics package, efficient vectorization of the compute intensive kernels can be quite involved. A good understanding of the kernels and data structures goes a long way when faced with this task for applications with thousands of source code files and functions. Getting compilers to recognize opportunities for vectorization with and without some assistance from the code developer (in the form of directives) is a high priority for our applications. From this perspective a comparative evaluation of the three compilers that are likely to be used by ACES code developers, namely: Intel, GNU and Cray, is of benefit to our user base. Towards this objective we evaluate the three compilers using the TSVC benchmark [1] and the Livermore Kernels benchmark LCALS [2] on Intel processors: Ivy Bridge, Haswell and Knights Corner (KNC). The performance gain seen with these processor architectures, with different SIMD units (AVX, AVX2 and MIC-AVX512 respectively) are investigated.

If the compiler provides good auto-vectorization for important kernels it allows effective optimization of a wide range of codes without requiring a large effort or in depth understanding of the microarchitecture. Compiler unrolling and peeling of compute intensive loops combined with the generation of packed SIMD instructions is our preference. We attempt to identify situations where a programmer may be able to help the compiler vectorize more loops through simple modifications to the program and by explicit help through compiler directives.

This study also investigates a set of compute intensive loops from Sandia's SIERRA Mechanics application suite [3]. An approach developed by the SIERRA Solid Mechanics code team is the creation of an abstraction layer called SimdLib which, by directly using SIMD intrinsics, assures good performance for the loops on all compilers independent of their ability to auto-vectorize. However, we also show that when auto-vectorization is aided with judicious insertion of pragmas it often leads to best possible performance because the compiler is able to take full advantage of loop optimizations and hardware features.

II. TSVC AND LCALS BENCHAMRK

The TSVC (Test Suite for Vectorizing Compilers) benchmark was originally developed by Callahan, Dongarra and Levine [4]. The version used for this study is an extended version developed by Maleki, Gao, Garzaran, Wong and Padua [5]. The extended version took the original version, converted it from Fortran to C and aligned all arrays to 16 byte boundaries. In addition, 23 new loops were added and 7 loops removed that the authors determined were obsolete. The extended version has 151 loops. We chose this benchmark as it provides a somewhat pathological collection of relatively simple loops that could be found in many scientific C codes and forms a good basis for compiler expectations as we explore more difficult code

segments found in real applications. We modified the array alignment parameter to 64 bytes in order to accommodate 512-bit SIMD units used in the Intel Knights Corner. This same alignment was used for Ivy Bridge and Haswell (256-bit SIMD units) for consistency.

The LCALS (Livermore Compiler Analysis Loop Suite) benchmark suite was developed by Rich Hornung at Lawrence Livermore National Laboratory. This suite was chosen as one of our benchmarks because it represents loops and "kernels" taken, or derived from, real codes. LCALS consists of three variants for testing different programming and execution constructs, and hence different aspects of a compiler's performance. The first variant employs traditional C/C++ for-loop syntax and is referred to as "Raw" variants. Other variants explore more complex C++ methods such as functors and lambda functions; these were not explored because the Cray C compiler does not support lambda functions at the time this work was performed. The suite also contains loop variants implemented with OpenMP; these were not explored, as we were only interested in the vectorization aspects of the compiler and not the interaction of OpenMP and vectors. For the "Raw" variants, the suite is broken into three subsets. The subset "A" represents loops representative of those found in application codes. Subset "B" is a collection of basic loops that help to illustrate compiler optimization issues. Subset "C" is extracted from "Livermore Loops coded in C" developed by Steve Langer, which was derived from the original Fortran "Livermore Loops" by Frank McMahon. Modifications to the original source code included: Setting *num_suite_passes* to 3; setting *run_loop_length* to false for LONG and MEDIUM test cases; commenting out all references for lambda function and OpenMP variants; and setting the cache size parameter to 30 MB. For the Intel Knights Corner tests the value of LCALS_DATA_ALIGN was set to 64 bytes to support the 512-bit SIMD unit. The default value of 32 bytes was used for Ivy Bridge and Haswell studies. For this initial study, we only performed the SHORT loop length case because we feel this is the most challenging case for the compiler.

It is a difficult and tedious task to examine the compiler generated vectorization reports for all 151 loops found in the TSVC suite to determine which loops vectorized and which did not. So we used the method of the Maleki et. al. [5] study and did runtime comparisons between timings of code generated with and without vectorization. The baseline timings are made with optimization turned on, but vectorization turned off. Note that since optimization is allowed the baseline timings may employ automatic compiler techniques such as inlining and loop unrolling. A second set of timings with the same optimization flags plus the appropriate vectorization flag set is collected and the ratio of without vectorization and with vectorization is calculated and compared to a threshold. If the ratio is greater than 1.15 we say that the loop vectorized. If the ratio is less than 0.85 we say the loop vectorized, but it is labeled as a slowdown.

The benchmarks are serial implementations and hence were run on a single core of the target processors and are not memory bandwidth limited. The footprint of both benchmarks is very small and the variable arrays of each test loop should fit in at least the last-level cache of the processors evaluated. The footprint of TSVC is ~2.5 MB and LCALS is ~150 MB. Given these constraints, the results should be truly representative of the potential performance improvement of vectorization without the limitation of being memory bandwidth bound.

For this study, we looked at three generations of Intel processors, the Ivy Bridge processor which has a 256-bit AVX SIMD unit, the Haswell processor with a 256-bit AVX2 SIMD unit, and the Intel Knights Corner which has an early implementation of the MIC-AVX512F (AVX3.1) SIMD unit. For the Ivy Bridge and Haswell targets, three compiler suites were evaluated, GNU, Intel, and Cray. The details of each processor architecture and platform are listed in Table 1. The compiler suites used are: Intel 15.0.2, GNU gcc 4.9.2 and Cray compilers under Cray Programming environment 5.2.40.

Table 1. Platforms and processors used

Processor	Platform Name	Specification/CPU
Ivy Bridge	Edison, Morgan04	Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
Haswell	Mutrino, Shephard	Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz
KNC	Corner, Morgan04	Intel(R) Xeon(R) Phi CPU @ 1.238 GHz

Table 2 summarizes the results of our study with the TSVC benchmark that contains 151 total loops. "Vectorized" are those loops that showed a speedup (>1.15), or slow down (< 0.85), between without and with vector optimization enabled. The "average speedup" includes only those loops that "vectorized", while "total time" is for all loops. TSVC uses single-precision floats, so the expected speedup is 16 for KNC and 8 for Ivy Bridge and Haswell. The Intel and Cray compilers did the best job, seeing a speedup on 66% of all loops versus the GNU compiler's 41%. The KNC results show speeding up 74% of all loops, while slowing down 5%. The Cray compiler sees a slightly higher number of loops that slowed down on the Haswell processor, 6%. The "total time" metric is the aggregate time spent in all 151 loops. For the Haswell, the Cray compiler provided a 1.07 speedup over the Intel compiler, and a 1.28 speedup over GNU. For Ivy Bridge, the trend is

Table 2. Results of the TSVC benchmarks, 151 total loops

	KNC	Ivy Bridge w/AVX			Haswell w/AVX2		
	Intel	GNU	Intel	Cray	GNU	Intel	Cray
vectorized	111	61	99	101	63	91	102
speedup	103	58	96	96	59	88	93
slowdown	8	3	3	5	4	3	9
average speedup	8.04	2.87	2.47	2.80	2.82	2.60	2.88
total time (min)	177.82	21.41	17.15	16.53	17.29	14.45	13.56

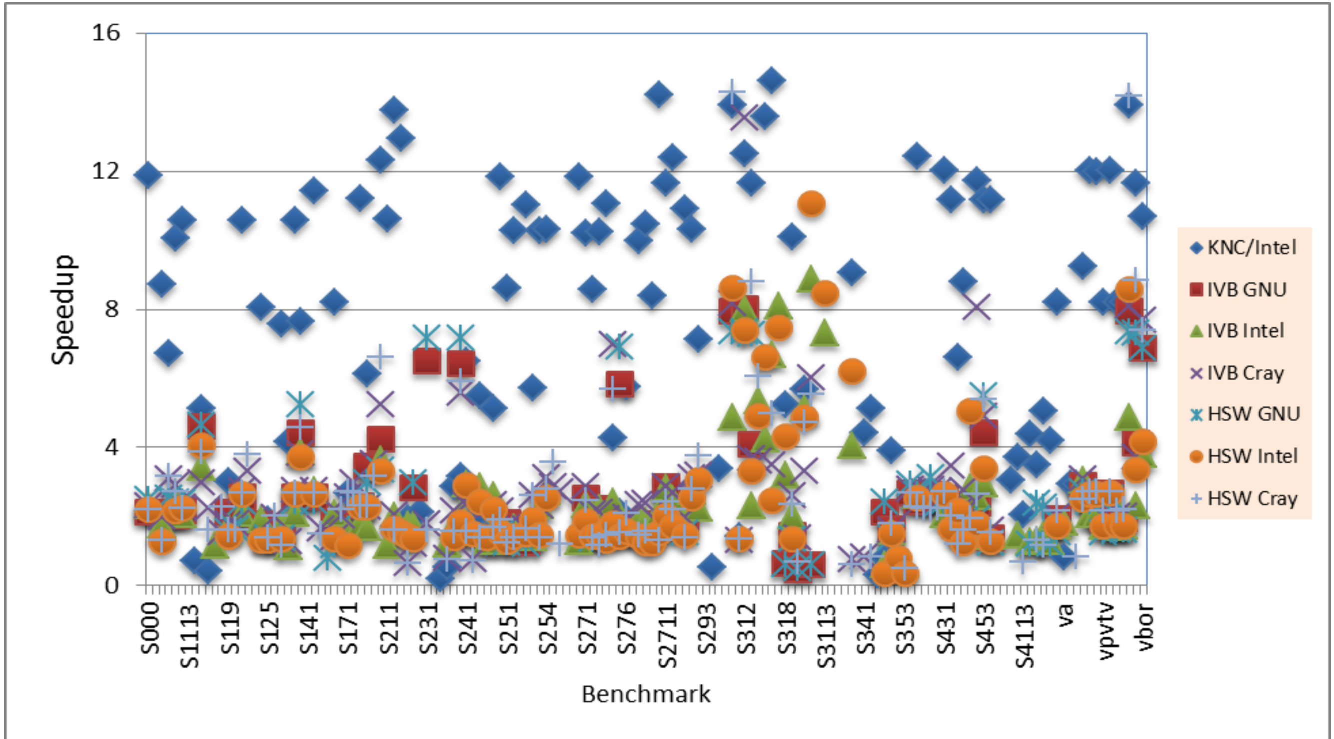


Figure 1. Measured speed up for the 151 loops of the TSVC benchmark

approximately the same. The total time of the KNC processor is significantly higher. This is due to the relatively low performance of a KNC core, which is further penalized by running only a single thread as the KNC requires at least 2 to 3 threads to achieve full instruction issue. A future effort may look at threaded versions of the benchmark in order to fully take advantage of current and next-generation architectures that depend on multiple threads to exploit maximum performance.

Figure1 is a plot of the speedup with vectorization of the 151 loops in the TSVC benchmark. This plot shows the speedup (> 1.15) and slowdown (< 0.85) for all loops that "vectorized". For the Intel KNC, the expected max speedup is 16. There are two loops not shown that showed a greater speedup, loops S314 (17.45), S3111 (20.30), S3113 (30.8). It can be seen that for the Ivy Bridge and Haswell results there are cases where the speedup is greater than the expected value of 8, but there is little correlation with the KNC results.

Table 3 summaries the results of the 30 loops of the LCALS benchmark. LCALS uses double-precision floats, so the expected speedup is 8 for KNC and 4 for Ivy Bridge and Haswell. For LCALS, the Intel compiler provided the best performance, vectorizing 53% of the loops for Ivy Bridge and 57% for Haswell. The GNU compiler provided the next best result with 30%, while the Cray compiler

achieved 20%. The Cray compiler showed a good speedup on the loops it did vectorize on Haswell, 2.98X, significantly higher than the Intel and GNU. Although the Intel compiler vectorized significantly more loops, its average speedup also includes 3 slowdowns.

Table 3. Results of the LCALS benchmark, 30 loops

	KNC	Ivy Bridge w/AVX			Haswell w/AVX2		
	Intel	GNU	Intel	Cray	GNU	Intel	Cray
vectorized	17	9	16	6	9	17	6
speedup	17	8	16	6	8	14	6
slowdown	0	1	0	0	1	3	0
average speedup	3.80	1.77	2.12	2.07	2.00	2.36	2.98
total time (min)	5.57	0.83	0.59	0.87	0.65	0.42	0.65

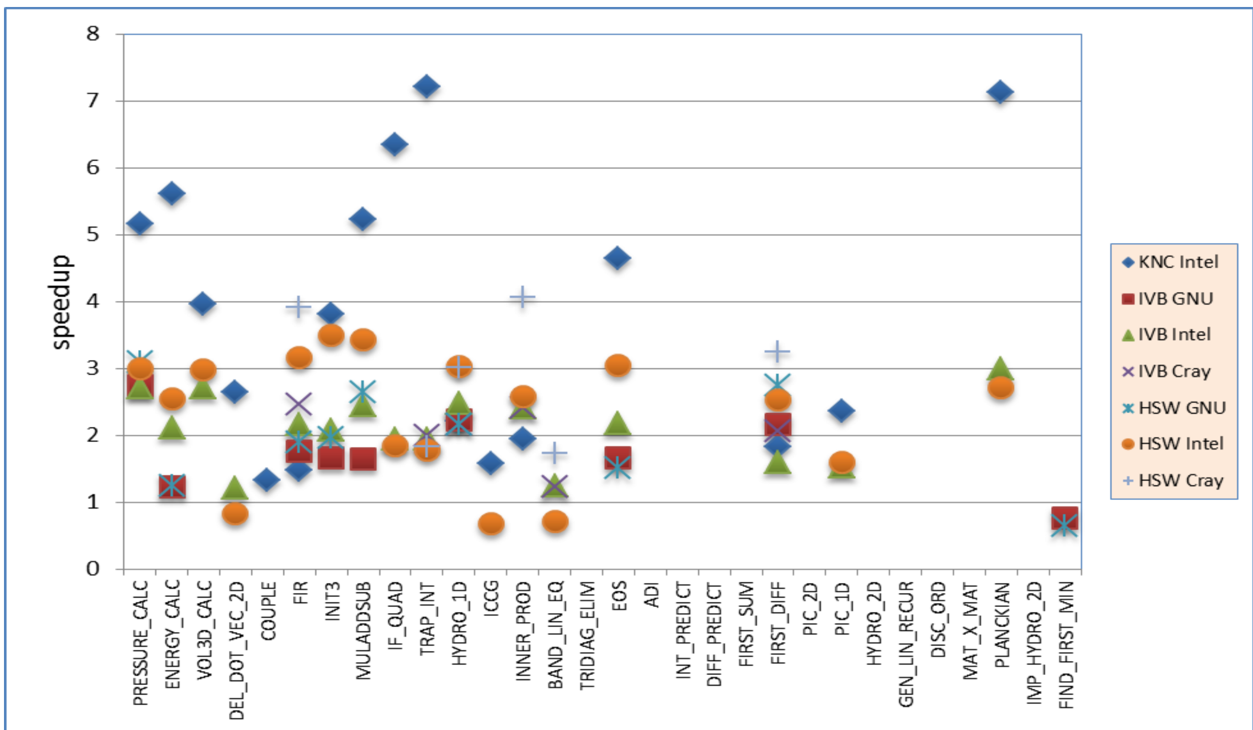


Figure 2. Measured speedup of the 30 loops of the LCALS benchmark

The slowdowns were not seen on the Ivy Bridge and Intel showed the best average speed up The Intel result showed the best total time, a speedup of 1.55 over Cray and GNU. For Ivy Bridge, the overall speedup using Intel was 1.47 and 1.41 respectively. As was seen with TSVC, the KNC processor showed very good average speedup, but the aggregate run time is much higher than the traditional Xeon processors. Future work may look at the loops that did not show a speedup and investigate code modifications (including directives) and more aggressive compiler techniques to see if improvements can be made to this baseline measure.

III. SIERRA APPLICATION KERNELS

Under the NNSA's Advanced Simulation and Computing (ASC) program, the SIERRA Mechanics finite-element codes have been developed and used as the principal tool in support of the U.S. stockpile stewardship program. This suite of codes includes coupled simulation capabilities for

thermal, fluid, aerodynamics, solid mechanics and structural dynamics. These large-scale codes incorporate physics and engineering models and specialized codes to predict, with reduced uncertainty, the behavior of weapons and their components in a variety of environments. In addition to supporting the stockpile, a number of other national security missions use these simulation tools for innovative product engineering.

In this section we investigate the performance tradeoffs of different vectorization implementations for important real SIERRA mechanics kernels, in contrast to the synthetic kernels with TSVC and LCALS presented in the previous section. In particular, we consider three time critical kernels from Sandia's SIERRA/Solid Mechanics finite element code [6]. SIERRA/Solid Mechanics is a general purpose massively parallel nonlinear solid mechanics finite element code for explicit transient dynamics, implicit transient dynamics and quasi-statics analysis of structures. It is built with extensive material, element, contact, and solver libraries and used at SNL for analyzing structural response of weapon components to normal, abnormal, and hostile environments. The kernels we investigate here constitute a significant portion of the computational expense for explicit-dynamics simulations of nonlinear material behavior (in the absence of contact). Each of these routines is computed once per element every time step, where the typical numbers of elements per MPI rank is in the thousands to hundreds of thousands. Identical computation for each element enables vectorization, provided the data structures are organized appropriately.

A. *Eigenvector kernel:*

This kernel computes for each element the eigenvectors and eigenvalues for a symmetric 3x3 matrix. The symmetric 3x3 is stored as a 6 long array, taking advantage of the matrix symmetry to reduce the memory footprint. The eigenvectors/eigenvalues are computed using an analytic formula which requires evaluation of conditionals and trigonometric functions. In order to allow for vectorization, these trigonometric functions are calculated to very near machine precision using a Padé approximation. The details of this approximation is beyond the scope of this paper, but it is relavant to point out that it only requires double precision multiplies, adds and divides. Conditionals are implemented via ternary operators.

B. *Elasticity Kernel:*

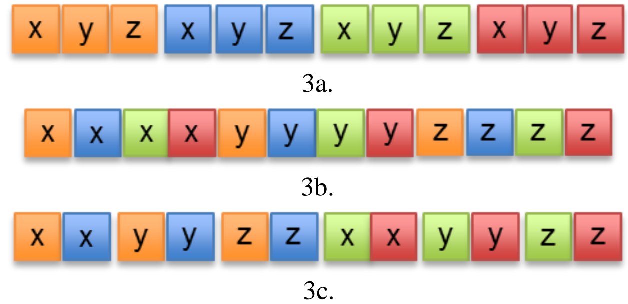
This kernel computes for each element a mechanical stress (symmetric 3x3 matrix) given a stretching tensor (symmetric 3x3 matrix) and a rotation tensor (non-symmetric 3x3 matrix). A Neo-Hookean elasticity model is used [7], where the material properties which characterize this model are the bulk modulus (which relates pressure with volume change) and the shear modulus (which relates shear stress with shear strain). This calculation is relatively straightforward in that it does not require any conditionals and the most complicated math operation is a cube-root.

C. *Plasticity Kernel:*

This kernel computes for each element a mechanical stress (symmetric 3x3 matrix) given a strain rate tensor (symmetric 3x3 matrix), the old stress tensor (symmetric 3x3 matrix), and an array of length 11 which stores the internal state history of the material. The model used is a standard J2 plasticity model with linear hardening [7]. The properties for this model are the bulk modulus, shear modulus, yield stress and hardening modulus. This model is the most complicated for vectorization as it has structs with stride 11 (i.e. 11 doubles), has many inputs, has conditionals and even has a while loop at the inner most level to assess convergence of the material model's plastic strain updates.

D. *Data Layouts*

We have measured the performance of each of the above three kernels using three different data structures: array-of-structs layout, struct-of-array layout, and SimdLib which uses a hybrid layout and directly uses vector intrinsics instead of relying on auto-vectorization. Figures 3a, 3b, and 3c are schematics for the three data structures, namely: the array-of-structs (**AOS**), struct-of-array (**SOA**) and SimdLib with intrinscics (**SLI**). For simplicity we show the case where the struct is a 3-vector. Blocks of the same color correspond to entries in the same 3-vector.



Figures 3a, 3b, 3c illustrate AOS, SOA and SLI data layouts

The layout, used for the SimdLib implementation of the kernels, is an array of structs-of-arrays, where the innermost array length is determined at compile time to be the SIMD vector-length. The Figure 3c depicts the layout for the case when the vector-length is 2 (i.e. SSE instructions). Note that, while we show the case of a vector-length of 2, this is only for purpose of the schematic. All the results presented below use AVX, AVX2, or MIC-AVX512 instructions with vector-lengths of 4, 4 and 8, respectively (for double precision floating point numbers). The layout changes described here are uniformly applied to all the data structures used as inputs and outputs to the kernels. The advantage of both the struct-of-array layout and hybrid layout over the more typical array-of-structs layout is that

data can be loaded directly into SIMD registers without the need for shuffle instructions to get the data into the correct layout required for vectorization across elements.

E. SimdLib

Here we provide a brief summary of the key motivations for and features of SimdLib. As previously mentioned, an alternative vectorization strategy to compiler auto-vectorization is the explicit use of SIMD vector intrinsics, which directly call corresponding assembly instructions. Direct use of intrinsics is typically ill-advised as they can be platform and compiler dependent. However, an approach developed by the SIERRA Solid Mechanics team overcomes this limitation by providing a simple platform portable abstraction layout using C++ templates and structs (similar to the Boost.SIMD library [8]). The key components of this library are a "Doubles" struct, a "Bools" struct, and an integer valued vector-length. At compile time, when the available SIMD instructions are detected, the vector-length is set to 1, 2, 4 or 8 depending on whether: no double precision SIMD instructions are available, SSE2 instructions are available, AVX instructions are available, or AVX512 instructions are available. The "Doubles" and "Bools" structs are then sized to the vector-length and most common mathematical operations (such as +,-,*,/,sqrt,<,<=,!=,&&,||,etc.) are overloaded to use the appropriate SIMD intrinsics on the data members of the "Doubles" and "Bools" structs.

In order to use SimdLib, it is necessary to get the data into the correct layout (as described in the previous section, with either an array of struct of array layout or just a struct of array layout) and to template relevant kernels on the "double" type. In addition, all arrays must be appropriately aligned to 16, 32, and 64-bit boundaries for SSE2, AVX, and AVX512, respectively. This approach assures good performance for application kernels on all compilers independent of their ability to auto-vectorize.

F. Vectorization Speed-up Results

To evaluate the effectiveness of the different vectorization strategies, we collected timing results from three Intel architectures: Ivy Bridge, Haswell and Knights Corner. We compute the kernels assuming that the number of elements is 200,000, with the arrays sized accordingly and with a 'for' loop over these 200,000 elements. An additional outer loop of 100 is used to increase the run-time and therefore decrease run-to-run timing variations.

Before going into the results, we mention that a few initial steps were required to get auto-vectorization to work at all for some of these kernels. The first change was to ensure that all of the kernels could actually inline, a prerequisite for auto-vectorization. In particular this required implementing the functions as inline functions in a header file. Second, it was necessary to increase the max inline size to 10000 with the icpc flag *–inline-max-total-size=10000*, due to the fact that some of these functions are around 200 lines long. With these two changes, the kernels inlined easily. A second difficulty had to do with the Intel compiler detecting vector dependencies which were not actually there. This is likely due to the fact that the Intel compiler has heuristics to efficiently detect these situations, but it may create a lot of false negatives. We have run these kernels through Intel's Thread Advisor tool, which performs a more thorough dependency analysis and determined that there are no dependency issues in these kernels. To overcome this limitation, we collect timing results for each kernel in two ways: once with no changes to the way the loops are called, and once with #pragma ivdep right before calling the relevant loop. To ensure correct code, the compiler treats an assumed dependence as a proven dependence, which prevents vectorization. This option tells the compiler to ignore dependency warning and vectorize anyways if it is profitable. The IVDEP directive assists the compiler's dependence analysis. It can only be applied to iterative DO loops.

Tables 4, 5 and 6 summarize the Ivy Bridge, Haswell and KNC results, for the three kernels investigated. They present speedup fractions relative to the baseline array-of-structs layout and no compiler auto vectorization. In other words the speed up ratio is computed in reference to run times when compiler vectorization is prevented through the use *–no-vec* flag.

It is an interesting finding from these results that for the two of the three kernels (Eigenvector and Elasticity) the SOA+IVDEP performance was indeed better than the SLI performance. Although we do not have a full understanding of the reasons behind this somewhat surprising outcome, it is suspected that "prefetch" instructions introduced by the compiler for SOA, must be leading to better streaming of data into the SIMD units. We studied this with CrayPat on our Cray XC30 with the Haswell processors. CrayPat measured ratio of the metric: MEM_UOPS_RETIRED:ALL_LOADS for the SimdLib runs and the SOA+IVDEP runs, yielded a value of 1.4 which was very close to the observed performance ratio of 1.38. CrayPat measurements also showed another metric that measures L2 prefetch hits: L2_RQSTS:L2_PF_HIT registered 3 times higher value for SOA+IVDEP over Simdlib while the misses as measured by the counter: L2_RQSTS:L2_PF_MISS were nearly the same. This suggests possible improvement of our SimdLib implementation through the addition of appropriate prefetch intrinsics.

Data in Tables 4,5 show that for the best performing SimdLib, we see an increase in performance of Haswell over Ivy Bridge of: 26.5% for the Eigenvector, 30.7% for the Elasticity and 21.5% for the Plasticity kernels. Interestingly the worst performing case, SOA without

IVDEP, shows correspondingly 17.1%, 32.7% and 22.7% gains on Haswell over Ivy Bridge.

Table 4. Ivy Bridge: SIERRA kernels speedup relative to AOS layout and no vectorization

	Eigenvector	Elasticity	Plasticity
AOS	1.62	1.01	0.99
AOS, IVDEP	1.67	1.61	0.98
SOA	1.09	0.99	0.70
SOA, IVDEP	2.45	2.19	0.71
SLI	2.27	1.86	1.80

Table 5. Haswell: SIERRA kernels speedup relative to AOS layout and no vectorization

	Eigenvector	Elasticity	Plasticity
AOS	1.80	1.00	0.97
AOS, IVDEP	1.74	1.37	0.97
SOA	0.90	0.99	0.58
SOA, IVDEP	2.53	2.45	0.59
SLI	2.03	1.79	1.54

Table 6. KNC: SIERRA kernels speedup relative to AOS layout and no vectorization

	Eigenvector	Elasticity	Plasticity
AOS	2.28	1.00	1.00
AOS, IVDEP	1.64	0.92	1.00
SOA	0.95	0.84	0.63
SOA, IVDEP	5.14	7.16	0.63
SLI	5.10	2.39	2.63

IV. TUNING WITH HARDWARE COUNTERS

Mini Applications as typified by Sandia's Mantevo project [9] are frequently used to investigate performance of new computer architectures and processors. Trinity acceptance testing includes (among other performance goals) investigation of performance of four mini application benchmarks: miniFE, AMG, UMT and SNAP. The details of these codes and benchmarks are available at http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/
We investigated performance of these four benchmarks with Intel compiler on KNC (native mode), measuring run times with compiler vectorization (*-O3* compiler switch) and with no vectorization (*-O3 –no-vec*). Results are shown in Table 7. These results suggest that further effort is needed to fully exploit the promise of substantial performance gains from the vector units in KNC and KNL.

Table 7. Mini applications vectorization performance

Application	miniFE	AMG	UMT	SNAP
% speedup	4.68%	6.52%	17.95%	19.52%

MiniFE, as it is representative of the SIERRA mechanics applications whose run times are predominantly in sparse matrix solver functions, was studied further for possible strategies for improving performance. Figure 4 shows the performance of miniFE on a 2 socket Sandy Bridge node and on a single KNC with 57 cores.

The baseline performance using only MPI with one task per core on the Sandy Bridge front end processor with 16 MPI tasks and on the KNC in native mode with 57 MPI tasks showed the KNC performance to be 23% slower than the front-end Sandy Bridge node. As a first step in-lining the Sparse MV kernels and adding OpenMP threading improved the performance by 23%. Additional gains in performance were achieved by disabling transparent huge pages and using selectively large page allocations for vector data structures to lower TLB miss rates. These tuning measures improved the KNC performance by 33% and exceeded the front end Sandy Bridge node performance by 20%.

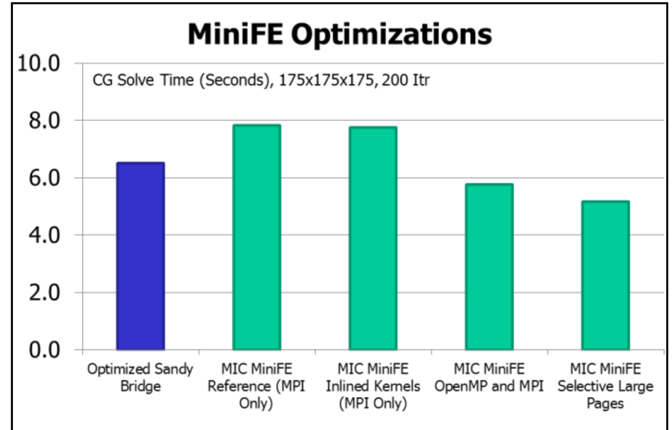


Figure 4. MiniFE performance optimization on KNC

Micro-architectural performance tuning using the hardware events available through the built-in Performance Monitoring Unit (PMU) on KNC can be accessed through Intel's Vtune. We have also installed a version of the TAU performance tool and used it to measure hardware counter metric ratios on KNC like *Vectorization Intensity* defined as:

Vectorization Intensity=VPU_ELEMENTS_ACTIVE / VPU_INSTRUCTIONS_EXECUTED

A matrix multiply benchmark using MKL's DGEMM on the KNC showed that the percentage of peak double

precision floating point operations achieved is about 30%, which is considerably less than published best performance of close to 90% [10]. A few measurements of Vectorization Intensity metric ratio on the MIC, gave a vectorization intensity value of 7.84. This metric has an upper bound of 8 and so values close to 8 suggest efficient use of MIC's SIMD units. However since the *VPU_ELEMENTS_ACTIVE* counter measures vector instructions like vector load/stores from memory, and instructions to manipulate vector mask registers, in addition to the double precision floating point instructions of interest to us, caution is needed in use of this metric for performance tuning. The fact that our measurement of this metric achieves close to the peak showing high vectorization intensity is misleading if our goal is to attain high floating point operations throughput. However it is anticipated that on the Intel Knights Landing processor the PMU will provide a FLOPS counter enabling easier identification of effective use of the vector units for floating point operations.

V. CONCLUSIONS

The TSVC and LCAL benchmarks show that significant improvements (up to 3X) in performance can be achieved if the compute intensive kernels of our applications are vectorized. Our study also points out that for some of the complex kernels as typified by the J2 plasticity kernel in SIERRA, direct use of SIMD intrinsics (in our case using the SimdLib abstraction layer) is necessary to achieve the desired performance. An important objective of the design of the SimdLIB is easy portability to processors with different lengths of the vector registers. However, an interesting observation from our study of the Elasticity and Eigenvalue kernels is that compiler auto-vectorization can indeed give the best performance when kernels have appropriate data structure layout and the compiler is aided by pragma directives. The importance of hardware performance counter measures to identify all aspects of effective use of the SIMD units is pointed out.

ACKNOWLEDGMENT

This work was supported in part by the U.S. Department of Energy. Sandia is a multi program laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States National Nuclear Security Administration and the Department of Energy under contract DE-AC04-94AL85000.

REFERENCES

1. Extended Test Suite for Vectorizing Compilers. http://polaris.cs.uiuc.edu/ ~maleki1/TSVC.tar.gz.
2. LCALS ("Livermore Compiler Analysis Loop Suite"). https://codesign.llnl.gov/LCALS.php.
3. H.C. Edwards, and J.R. Stewart. SIERRA: A Software Environment for Developing Complex Multi-Physics Applications. In K.J. Bathe (ed.) *First MIT Conference on Computational Fluid and Solid Mechanics*, Amsterdam, Elsevier.
4. D. Callahan, J. Dongarra, and D. Levine. 1988. Vectorizing compilers: a test suite and results. In *Proceedings of the 1988 ACM/IEEE conference on Supercomputing* (Supercomputing '88). IEEE Computer Society Press, Los Alamitos, CA, USA, 98-105.
5. S. Maleki, Y. Gao, M.J. Garzarán, T. Wong, and D.A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques* (PACT '11). IEEE Computer Society. Washington, DC, USA, 372-382
6. SIERRA/SM development team. 2014. SIERRA/SM: 4.32 Verification Tests Manual. Sandia National Laboratories, SAND2014-3257, Albuquerque, NM and Livermore, CA.
7. SIERRA/SM development team. 2013. SIERRA/SM: Theory Manual. Sandia National Laboratories, SAND2013-4615, Albuquerque, NM.
8. P. Esterie, M. Gaunard, J. Falcou, J.-T. Lapreste, and B. Rozoy. 2012. Boost. simd: generic programming for portable simdization. In *International Conference on Parallel architectures and compilation techniques*, 431-432.
9. https://software.sandia.gov/mantevo
10. Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, Aniruddha G Shet, George Chrysos, Pradeep Dubey, "Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems Based on Intel Xeon Phi Coprocessor", IPDPS 2013.