On the Use of Vectorization in Production Engineering Workloads

C.T. Vaughan, J. Cook, R.E. Benner, D.C. Dinge, P.T. Lin,
C. Hughes, R.J. Hoekstra, S.D. Hammond
Center for Computing Research
Sandia National Laboratories
Albuquerque, New Mexico, United States
$\{$*ctvaugh, sdhammo*$\}$*@sandia.gov*

Abstract—**Arguably, one of the greatest successes of early Cray supercomputers was the use of highly efficient vector-based computation coupled to a balanced memory subsystem. In addition, efficient scalar throughput helped establish Seymour Cray's designs as the benchmark by which other systems were measured. Recent high-performance processor designs have seen a resurgence in the use of vector-like hardware units. Some in the industry have also argued that use of accelerators such as GPUs will also lead to a greater focus on code being written to be amenable to vector-based computing. For the authors of this paper, a motivation to focus on efficient vectorization is to improve performance of the production ASC Trinity supercomputing platform which comprises approximately 9,000 nodes of dual-socket Haswell processors and 9,500 nodes of Intel Knights Landing sockets – both of which gain much of their computation prowess from the use of vector units in the floating point pipeline.**

In this paper, we describe a study of several modern, production engineering codes which are routinely used at Sandia National Laboratories and other important NNSA computing partner sites, evaluating the levels of utilization for vector units and the performance benefits obtained from vectorized computation. Our results show varying levels of benefit – vectorization is not always faster. Additionally, we show the ratio of vector to integer/logical instructions providing some insight as to why even highly vectorized code does not achieve high levels of performance improvement.

Keywords-**Vectorization, SIMD, HPC, Workload, Analysis**

I. Introduction

Vector processors, sometimes called, single instruction, multiple data (SIMD) processors, have a pedigree which dates back to some of the earliest large-scale computer architectures, including the CDC STAR-100 and the TI-ASC. For many in the high-performance computing industry, however, vector computing is synonymous with the development of Seymour Cray's early designs in the mid 1970s. For the Department of Energy's National Nuclear Security Administration (NNSA), the use of vector processing stretches back to the deployment of the very original Cray-1. The advantage of such technology is the low overhead of fetching and decoding an instruction for execution over multiple operands, thereby increasing the efficiency of the computation.

During the late 1990s and early 2000s, vectorization became a much less critical component of supercomputer performance as the market shifted away from high-performance systems that had been typified by vector-processors, to the use of many more nodes filled with lower-cost commodity sockets. The effect was to see vectorization become much less important to computational throughput as the commodity market was instead being driven by ever increasing clock rates – a period where "the killer micros" all but dominated the market. However, as the clock rate improvements of this period began to slow, hardware designers in the commodity space began to switch back to use vector-based processing, albeit at very limited widths, to increase the performance of multi-media applications such as video decoding and audio processing. Over the last decade, this initial foray back to vector processing has gained momentum and now, wider vector units have re-emerged with much more sophisticated features such as vector-lane masking, gather/scatter capabilities and horizontal operations.

The NNSA's most recent production computing platform – the ASC Trinity supercomputer [1] – deployed as a joint partnership between Los Alamos and Sandia National Laboratories, has made the community return to focus on vectorization in the form of its Haswell [2], [3] and Knights Landing processor [4], [5] partitions. The Intel Haswell sockets offer dual-vector processing units at 256-bit widths but without the ability to mask individual vector lanes unless using explicit logical operations. For the 9,500 nodes in the Intel Knights Landing (KNL) partition of Trinity, vectorization is much more important. KNL offers dual vector-processing units of 512-bits in width with significantly more complex operations being possible. The very limited scalar throughput of the KNL core, when compared to Haswell, places an additional pressure on developers to make as much use of the vector units as possible.

Our early experience with application bring up for KNL showed a number of problems for the use of vector units. During the period of commodity processor use (circa 1990s - 2000s), when many of the NNSA's codes which rewritten to utilize MPI instead of vector-processors, the common programming patterns required to enable automatic vectorization by compiler were often ignored. When combined with the significant growth in code size and complexity resulting from the availability of more powerful systems, this period seems to have resulted in the shift of code to being inherently challenging to vectorize efficiently. This represents a significant challenge for our latest supercomputing platform.

In this paper, we present an analysis of several key applications and algorithm domains that are of interest to the NNSA/ASC HPC laboratories. We show a wide variation in the use of vector capabilities for our codes, and, we reveal some of the low-level behavior that explains why we are unable to gain performance from these codes when they are ported to the KNL platform. Although KNL may appear to be an older processor (availability in 2015), it acts as a sentinel for other systems which are in the future, including the Intel Skylake Xeon processor which will offer similar 512-bit wide vectors, and, processors developed by Arm to use its Scalable Vector Extension (SVE) instructions [6]. Vector computing is far from being an uninteresting, historical technology. In fact, the use of data-parallel execution is gaining, driven by the need for much greater efficiency of operations per unit of power. We therefore expect that the use of vector-like capabilities will increase in the build up to Exascale-class computing and that studies like the one reported in this paper, will be necessary to ensure we able to maximize the performance of future HPC platforms.

The remainder of this paper is laid out as follows: Section II opens our paper with some of the motivations that encouraged our study of vectorization in large-scale applications. In Section III we describe our vectorization analysis tools which are utilized to provide the data for this study. Section IV discusses our low-level application analysis. Finally, we conclude the paper in Section V.

II. Motivation - Application Benchmark Runs on Knights Landing

During the early application bring up on the ASC Trinity Knights Landing partition, benchmark runs had demonstrated a mixture of applications which would perform faster than the dual-socket Haswell nodes and then some applications which were much slower. Since the significant bulk of the compute performance of the KNL processors was made available via the dual VPUs, a number of application porting teams benchmarked runs of their codes with and without vectorization enabled. For all runs reported in this paper we use the Intel 17.0.4 compiler update. Platform targeting is delivered via appropriate use of Cray-PE modules. Vectorization is disabled through the explicit setting of the `-no-vec` flag which instructions the compiler to continue to perform optimization but to generate scalar compute instructions. The purpose of these studies was to see if codes were gaining significantly from vectorization by disabling the generation of vector instructions. We have recreated an example set of these runs for a 64-core (single node) run of a range of ASC-relevant applications (shown in Table I). We note that the difference between application execution with and without vectorization enabled is small for all codes, the largest gain is with the CTH hydrodynamics application running in flat-mesh mode (24% improvement in runtime) and the worst is a 14% application slow down (SPARC). We also note that the improvement in application performance is not correlated with either size or implementation language – both small and large applications can benefit from vectorization, as well as those written in C, C++ and Fortran. This correlates with our opinion that code complexity and implementation quality are greater determinants of whether vectorization is useful and motivates us to analyze the execution behavior so that we can report the analysis to application developers and hardware designers.

Given these results, which display a significant range in execution behavior, we attempted to perform a more in-depth analysis of instruction execution behavior.

III. Vectorization Analysis Tools

Our first attempt at performing vectorization analysis used performance counters available on the Knights Landing processor. However, KNL has an extremely limited support for performance counters and it is all but impossible to sufficiently differentiate floating-point vector instructions from other instructions such as wide loads, stores and integer operations that utilize the vector pipeline. Further, we intended to study the enable/disablement of vector lanes using vector masks, while a counter is available on KNL to perform this operation, the results obtained from using it were not fully consistent with our expected results from pathological test cases. We concluded that the use of hardware-only results was not sufficient for a study of this nature.

In order to address these shortcomings we developed an Intel PIN-based [13] tool suite called APEX (*APplication Characterization for EXascale* [14]. The tool performs a custom instrumentation of each instruction in an application, along with a custom identification of basic-block structures (not using those supplied by PIN) as well as an instrumentation of each vector instruction which uses mask registers. The result is a tool which is able to analyze multi-threaded, multi-rank (MPI) applications and perform extremely detailed analysis of compute and memory operations within the binary. Overhead is limited through careful use of atomic operations to increase counts rather than through the generate of locks around critical structures.

For the remainder of this paper we utilize terminology which is reported by our APEX tools:

- **Operation** - is defined to be a mathematical calculation, for instance an additional, multiplication etc. We could each operation performed on a piece of data separately. In the case of compounded operations as performed from the hardware perspective, for instance, a fused-multiply add, we count two operations, the multiplication and then subsequent addition.
- **Instruction** - an instruction is a hardware artifact which is decoded and then issued to perform one or more op-

Table I: Application Execution Times with and without Vectorization Enabled for Example 64-Core Intel Knights Landing Benchmark runs, Lower Time is Better, Lower Ratio is Better.

Application	Execution Time		Ratio	Domain	Language	Code Size
	Vectorized	Non-Vectorized				(Source Lines)
CTH [7] Flat Mesh	2058.7	2696.8	0.76	Hydrodynamics (Struct.)	Fortran 77/90	O(1M)
CTH [7] AMR Mesh	1019.3	946.2	1.08	Hydrodynamics (Struct.)	Fortran 77/90	O(1M)
miniAMR [8]	582.4	613.0	0.95	Hydrodynamics (Struct.)	C	O(10K)
LULESH [9]	332.1	349.9	0.95	Hydrodynamics (Unstruct.)	C/C++	O(10K)
PARTISn [10]	641.8	699.2	0.92	Transport	Fortran 77/90	O(500K)
LAMMPS (LJ) [11]	395.4	392.0	1.01	Molecular Dynamics	C,C++	O(100K)
SAGE	56.4	57.4	0.98	Hydrodynamics (Unstruct.)	Fortran 77/90	O(100K)
SPARC CFD [12]	2394.9	2108.2	1.14	Computational Fluid Dynamics	C/C++	O(2M)

erations over data. Instructions can be either vectorized, when they may issue operations over multiple lanes of data, or non-vectorized where they execute operations over scalar values. In each case, we count only a single instruction regardless of how many operations are performed.

- **Masked Operations** - a mask is used by the vector units on KNL to disable a specific vector lane from having its operation from being performed. In this case, we do not count the operation in our running total (since no side effect is observable).
- **Packed Instructions** - we use the term "packed" for the issue of an instruction which is both vectorized, and, for which the instruction executes operations on *all* vector lanes. Such instructions represent the most efficient use of the decode/issue pipeline.

IV. Application Analysis

A. Floating-Point Instruction Intensity

Table II: Percentage of Instructions Executed that Perform 64-bit Double Precision Compute Operations

Application	Vectorized	Non-Vectorized
CTH AMR Mesh	3.58	6.03
CTH Flat Mesh	11.46	22.96
MiniAMR	30.33	36.31
LAMMPS	40.80	40.74
LULESH	48.16	49.36
SAGE	10.71	14.17
SPARC	13.22	17.19

Table II shows the percentage of total application instructions which, when executed, perform a double-precision floating-point operation. Smaller applications (MiniAMR, LULESH and LAMMPS) all see significantly higher floating point calculation intensity when compared with the larger application portfolio. We see a perhaps unsurprising correlation that applications which experience a performance gain from the use of vectorization also see an increase in the percentage of instructions which perform compute operations when vectorization is disabled. This reflects the continued observation that even though floating-point operations may not dominate executed instructions (as we see from our results), they continue to be significant determinants of execution time. The outlier in this analysis is the result from the CTH AMR Mesh which sees an increase in percentage of instructions executed when vectorization is disabled but does not experience a performance gain from their use.

Observation: floating-point arithmetic instructions continue to be a minor, and in some cases very small, component of application execution. This creates a concern for architectures that are optimized for machine learning or Exascale-class LINPACK runs, since they direct greater proportions of the processor die to floating-point operations despite these being a small component of execution.

B. Packed Vector Instruction Intensity

Table III: Percentage of Vector Instructions Executed that Perform Packed Operations

Application	Vectorized	Non-Vectorized
CTH AMR Mesh	64.07	0.45
CTH Flat Mesh	77.03	0.22
MiniAMR	32.91	0.00
LAMMPS	0.00	0.00
LULESH	21.25	4.27
SAGE	50.92	0.00
SPARC	57.17	6.12

As stated, packed vector instructions are the most efficient method of decoding and issuing operations, and, wherever possible these are preferred for compute intensive routines. Table III shows our analysis of application instruction streams in this context. Note that for some applications, the non-vectorized components are non-zero despite vectorization being disabled because platform libraries/runtimes (such as libc) are able to perform limited dispatch of system functions based on runtime hardware detection (for instance, calls to memory copy operations).

CTH shows the greatest number of packed instructions issued with up to 77% of its computation being performed using over packed operations. Similarly, SAGE demonstrates high levels of packed instructions. We attribute CTH and SAGE to their use of Fortran which is more amenable to vectorization even in the presence of complex application flow. SPARC is the outlier amongst applications written in C/C++ with up to 57% of its instructions using packed operations. This is attributable to the use of manually vectorized routines (using intrinsics) deep within the low-level kernels of the Trilinos solver framework. In this setting, its inner most compute kernels are optimized on a per-ISA basis to yield high levels of performance. Despite this, it does not show significant gains when vectorization is enabled/disabled. LAMMPS shows no significant change in packed operation counts (both are zero) when vectorization is enabled which reflects the virtually insignificant change in runtime when vectorization is enabled.

Observation: Fortran codes continue to show efficient vector instruction sequences with greater proportions of the vector instructions executed being performed over packed operands.

C. Masked Instruction Intensity

Table IV: Percentage of Vector Instructions Executed that Utilize Vector Masks

Application	**Vectorized**	**Non-Vectorized**
CTH AMR Mesh	6.00	3.76
CTH Flat Mesh	8.18	1.74
MiniAMR	0.49	0.0
LAMMPS	0.0	0.0
LULESH	0.24	0.16
SAGE	3.31	0.0
SPARC	6.13	0.98

One of the most important new features of the AVX512 (512-bit) wide vector instruction set introduced with the KNL processor is the ability of the vector instructions to utilize mask registers to selectively disable vector lanes from computing. These features are useful for several reasons: (1) they enable vectorization even in the presence of complicated control flow (for instance, if-else structures or C-style tertiary operators); (2) they can make the code easier to compile/vectorize since complex flow can be handled and smaller loops/inner loops can be unrolled and completely vectorized with appropriate mask-generation. The downside of mask registers being available is that they can give the impression to the developer that the code is vectorized, when reported by the compiler, but in practice, the lanes are so frequently disabled that only very low-levels of compute efficiency are obtained. In Table IV we show the percentage of vector instructions which utilize masks for execution. For all codes this is relatively low with a maximum of 8.18% in the case of the CTH flat mesh runs.

Observation: vector masking is used relatively infrequently for all codes analyzed in this paper. From initial inspection, it may seem that such features are therefore redundant, however, their availability is often used by the compiler to enable vectorization to be performed since without it the cost-models of code sequences will often direct the compiler to drop back to scalar execution.

D. Average Vector Lane Density when Using Vector Masks

Table V: Density of Enabled Vector Lanes when using Vector Masking

Application	**Vectorized**	**Non-Vectorized**
CTH AMR Mesh	3.23	1.42
CTH Flat Mesh	2.25	0.26
MiniAMR	2.12	1.36
LAMMPS	0.0	0.0
LULESH	3.58	4.26
SAGE	6.61	0.0
SPARC	0.82	1.64

In Table IV we presented the percentage of vector instructions which utilized the masking capabilities of the VPUs in KNL. Table V shows the density of enabled lanes when a mask register is used, for reference the maximum value in KNL is 8 since the vector register is 512-bits wide and each operand occupies 64-bits. SAGE makes the most efficient use of the mask registers with an average density of 6.61 operands per masked-instruction. SPARC is the least efficient at 0.82 operands which implies that some masks disable the vector register entirely during execution to yield less than one operand per masked-instruction. It is possible an algorithm optimization of all zero-detection could be used here to shortcut redundant computation.

Observation: most masked codes make low efficiency use of the VPU when masked-instructions are executed with less than halve the lanes being enabled. SPARC is a particularly low efficiency code for this class of operation with some instructions operating over fully disabled vector registers. Such an occurrence would benefit from short circuiting further execution.

V. Conclusion

Despite being a technology which dates back to the origins of high-performance computing, vectorization continues to be a hardware function of interest in contemporary processor design. Moreover, the restrictions in application code that are required for efficient vector instruction sequences to be generated are of interest as the community progresses towards Exascale-class systems and beyond. Particularly, the annotation of non-overlapping data structures and the promise of operations being side-effect/interference free are likely to be required well into the future.

In this paper we reported on our experiences with some of the first application ports of NNSA/ASC-relevant codes to the newest production-class supercomputing resource – the Trinity platform housed as Los Alamos National Laboratory. The use of the KNL system partition showed limited early performance improvement when vectorization was enabled. Through the use of a custom binary analysis tool, we have been able to perform a deeper analysis into the application instruction sequences to determine that a limited number of operations in all the codes analyzed are performed using fully dense vector instructions. We evaluated the use of vector masks, one of the novel features of the AVX512 instruction set which debuted in KNL, and showed that applications typically make fairly infrequent use of masked instructions, but that, when they do, the operation density is typically low – less than half of the vector lanes are enabled on average.

Floating point operations continue to be a minor components of our large-scale scientific applications, in part because of complex control flow, address calculation and data movement. Yet, the community continues to utilize the HPL/LINPACK benchmark as a herald of performance. Our results show that the many in the community who question whether HPL is a good determinant of our code performance, may well have an accurate observation to make. Even when we vectorize, and so enable much more efficient compute decode and issue, we fail to find significant gains in application performance. Such hardware structures are built to improve LINPACK and dense-BLAS performance but the results from this paper show we do not currently see the benefits that such designs can bring in real applications. This motivates deeper application work to port our codes to vector architectures – a skill which was largely lost during the period of rapid commodity processor frequency improvements. Such work is likely to bring benefit not only on the systems of today, but as described above, also the systems of tomorrow.

Acknowledgements

Access to the benchmark resources used for this paper are provided by the Application Readiness Testbed (ART) Systems fielded by Sandia's production computing support group. Additional runs and tool development was performed on the NNSA/ASC Advanced Architectures Test Bed platforms, a sequence of hardware deployments by the NNSA/ASC program which provide early access systems to research staff across the NNSA Tri-Lab community. We are grateful to the ART and Test Bed project administration teams who configure and maintain the hardware systems.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

[1] C. T. Vaughan, D. Dinge, P. Lin, S. D. Hammond, J. Cook, C. R. Trott, A. M. Agelastos, D. M. Pase, R. E. Benner, M. Rajan *et al.*, "Early Experiences with Trinity-The First Advanced Technology Platform for the ASC Program," Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), Tech. Rep. SAND2016-3605C, 2016.

[2] T. Jain and T. Agrawal, "The Haswell Microarchitecture - 4th Generation Processor," *International Journal of Computer Science and Information Technologies*, vol. 4, no. 3, pp. 477–480, 2013.

[3] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar *et al.*, "Haswell: The Fourth-Generation Intel Core Processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, 2014.

[4] A. Sodani, "Knights Landing (KNL): 2nd Generation Intel Xeon Phi Processor," in *Hot Chips 27 Symposium (HCS), 2015 IEEE.* IEEE, 2015, pp. 1–24.

[5] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights Landing: Second-Generation Intel Xeon Phi Product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[6] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu *et al.*, "The ARM Scalable Vector Extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[7] J. M. McGlaun, S. Thompson, and M. Elrick, "CTH: a Three-Dimensional Shock Wave Physics Code," *International Journal of Impact Engineering*, vol. 10, no. 1-4, pp. 351–360, 1990.

[8] C. T. Vaughan and R. F. Barrett, "Enabling tractable exploration of the performance of adaptive mesh refinement," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on.* IEEE, 2015, pp. 746–752.

[9] I. Karlin, J. Keasler, and J. Neely, "LULESH 2.0 Updates and Changes," Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep. LLNL-TR-641973, 2013.

[10] R. E. Alcouffe, R. S. Baker, J. A. Dahl, S. A. Turner, and R. Ward, "PARTISN: A Time-Dependent, Parallel Neutral Particle Transport Code System," Los Alamos National Laboratory, Tech. Rep. LA-UR-05-3925, May 2005.

[11] S. Plimpton, P. Crozier, and A. Thompson, "LAMMPS – Large-scale Atomic/Molecular Massively Parallel Simulator," *Sandia National Laboratories*, vol. 18, pp. 43–43, 2007.

[12] M. Howard, A. Bradley, S. W. Bova, J. Overfelt, R. Wagnild, D. Dinzl, M. Hoemmen, and A. Klinvex, "Towards Performance Portability in a Compressible CFD Code," in *23rd AIAA Computational Fluid Dynamics Conference*, 2017, p. 4407.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[14] S. Hammond, "Towards Accurate Application Characterization for Exascale (APEX)," Sandia National Laboratories, New Mexico, United States, Tech. Rep. SAND2015-8051, September 2015.