Program Optimization Strategies to Improve the Performance of SpMV-Operations

Rudolf Berrendorf*, Max Weierstall*, Florian Mannuss†
* Computer Science Department, Bonn-Rhein-Sieg University, Sankt Augustin, Germany
e-mail: rudolf.berrendorf@h-brs.de, max.weierstall@h-brs.de
† EXPEC Advanced Research Center, Saudi Arabian Oil Company, Dhahran, Saudi Arabia
e-mail: florian.mannuss@aramco.com

Abstract—**The SpMV operation – the multiplication of a sparse matrix with a dense vector – is used in many simulations in natural and engineering sciences as a computational kernel. This kernel is quite performance critical as it is used, e.g., in a linear solver many times in a simulation run. Such performance critical kernels of a program may be optimized on certain levels, ranging from using a rather coarse grained and comfortable single compiler optimization switch down to utilizing architecural features by explicitly using special instructions on an assembler level. This paper discusses a selection of such program optimization techniques in this spectrum applied to the SpMV operation. The achievable performance gain as well as the additional programming effort are discussed. It is shown that low effort optimizations can improve the performance of the SpMV operation compared to a basic implementation. But further than that, more complex low level optimizations have a higher impact on the performance, although changing the original program and the readability / maintainability of a program significantly.**

Keywords–Sparse Matrix Vector multiply (SpMV); Single Instruction Multiple Data (SIMD); OpenMP, unrolling; intrinsics.

I. Introduction

Sparse matrices are widely used in many areas of natural and engineering sciences [1], escpecially in simulations. An often used operation on such matrices is the multiplication of a sparse matrix with a dense vector (SpMV). This operation is often the most time consuming operation in iterative solvers (e.g., CG, GMRES [1]), which are the most time consuming operations in many simulations. Therefore, much attention has been given to optimize this operation. One point in an optimization discussion is the choice of an appropriate storage format for the sparse matrix [2]–[6], which depends mainly on the given matrix stucture (e.g., high / low matrix bandwidth, etc.) and the target architecture, e.g., multicore CPU, multiprocessor system, Graphics Processor Unit (GPU). Compressed Sparse Row (CSR) [1] is a general storage format for sparse matrices that performs quite good, especially on CPU-based systems. But even for a fixed storage format like CSR, there are opportunities for program optimization on different abstraction levels.

CPUs and memory systems are optimized for specific workloads in programs. Other than utilizing the memory hierarchy, instruction pipelining and vector units in processors can have a significant influence on a program's performance. For instruction pipelining, large basic blocks are favorable in a program. All recent processors have also some implementation of vector registers and related vector instructions [7] that can significantly speed up computations that exploit this architectural feature. Compilers can optimize code with large basic blocks with much room for optimizations and by vectorizing loops [8]–[12], as long as all data dependencies are respected [13].

Some high level programming models, especially designed for parallel (and therefore resource intensive) computing, have some notations to give hints to a compiler concerning vector operations. For example, OpenMP [14] as the de facto standard for shared memory parallelism has got in the recent revision 4 some annotations to guide a compiler in using vector units in a processor. But vectorizing compilers and directives to give a compiler hints existed already before OpenMP [15].

Other than leaving every optimization to a compiler, there are program optimization techniques known that allow to restucture a program to optimize certain operations (that a compiler may not detect). This restructuring of source code can be done by an expert programmer or by a sophisticated tool [16] [17] [10].

And, in a third way, if for example a compiler is not able to generate fast code because for example complex index expressions exist, a programmer may use vector intrinsics on a rather low abstraction level to program the hardware directly on a more or less assembler level [18]. For some high level language like C / C++, this can be accomplished by using compiler extensions called intrinsics that look like functions calls in the programming language and correspond to one or few assembler instructions.

In a summary, there are certain levels on which a time consuming operation may be speed up. In this paper, the question will be answered for the SpMV operation what the programming effort is needed for an optimization and what performance improvement one can get, if any.

The paper is structured as follows. Section II gives an overview on related work. After that, Section III discusses some program optimization techniques that are used for the investigations in this paper. Section IV describes the test environment for our evaluation. Section V shows and discusses performance results. The paper is summarized in Section VI.

II. Related Work

Compiler writers give hints in user's guides [8] and technical notes [19] how to optimize programs and how to write programs in a way such that a compiler can apply optimizations. Further than that, there exists optimization guides with

Copyright (c) IARIA, 2016.

a detailed description of hardware features that a programmer can use [20] [21].

Dedicated to the SpMV operation many papers were published describing performance related program optimizations of various types (e.g., register blocking) that were applied in [22]–[25].

In [26], Wende discusses the use of SIMD functions (i.e., vector intrinsics) to impove the performance on Intel Xeon processors and Xeon Phi coprocessors [27] escpecially for branching and conditional functions calls. He found that for this special application scenario there are only rare situations with a performance improvement by using vector intrinsics. This was mostly the case if the ratio of arithmetic operations to control logic is low.

III. Program Optimizations

Nowadays, processor and memory architecures are rather complex. Many architectural optimitations have been done in last decade's processor architectures that may improve the performance of programs significantly. Such architectural improvements include multi scalarity, out-of-order execution, pipelining and many others [7]. For all enumerated hardware optimizations it is favorable to have large basic blocks (code without any branches). Unfortunately, the SpMV has a rather small loop body for many storage formats. A well-known technique called loop unrolling [10] [11] enlarges the basic block of a loop body.

A significant performance boost for many applications is the use of vector registers / units that are available in almost all recent processor architectures [28]–[30]. These vector architectures follow the well-known SIMD principle [31] that one instruction is applied to several operands at the same time. For a vector width of n data elements, this may result in a speedup of up to n. Recent processors eligible in High Performance Computing (HPC) have a vector register / unit width of up to 256 bits, corresponding 4 double precision elements, each 64 bits. Recent announcements show [32] that the vector width will double in the near future with a nominal floating point performance increase of a factor of two.

The enlargement of basic blocks in a loop body and the use of vector registers can be used to speed up an SpMV operation. There are now certain levels of abstraction on which a programmer may influence these (and other) optimizations.

A. Compiler Flags

A simple strategy is to leave any optimization to a compiler. A programmer may specify on a rather coarse level some general compiler optimization level (i.e., `-O2, -O3`) leaving any decisions and optimization strategies solely to the compiler according to the specified optimization level. Specifying an optimization level of 2 instructs many compilers to enable many optimization techniques that have no influence on the semantics of a program, i.e., no optimizations are applied that may change the meaning of a program as for example using a faster floating point artithmetic.

Additionally on a finer level, special compiler options can be used to include certain optimization techniques or to utilize certain architectural features. An example for that is to allow the generation of code that utilizes the latest additions in the instruction set. For example, the compiler option `-march=haswell` of the GNU compiler g++ [33] allows the generation of advanced instructions only available on Haswell processors. Alternatives would be for the previous generations of Intel processors `-march=ivybridge` or `-march=sandybridge`. The code may be no longer executable on processors of generations previous to the one specified. Other compilers have the same possibilities but with a different syntax of such an option. Without the specification of such an architectural option the compiler generates code with an instruction set that corresponds by default to rather old processors to allow the compiled program to run on many systems, even older ones.

The PGI compiler [34] offers an option to instruct the compiler to generate vector code utilizing vectors of a specific size. For example the option `-tp=haswell -Mvect=simd:256` directs the compiler to generate code for Haswell processor, i.e., utilizing the Advanced Vector Extensions 2 (AVX2) instruction extensions, and to work with vectors of up to 256 bits.

```
#pragma omp simd reduction(+:s)
for(int i==; i<n; i++)
    s += a[i];
```

Figure 1. Example code for the use of an OpenMP pragma.

B. Language Directives

Sometimes, a compiler may not be able to recognize that certain optimization techniques could be applied to a code sequence. For example, this may be the case because the compiler can not know at compile time the value of certain variables, the alignment of variables or cannot exclude data dependencies because of complex index expressions. But, if a programmer can assure that for example a certain variable is always larger than 100 the compiler could optimize this program code. There exist program annotations for exactly these situations to tell a compiler some additional semantic information. Dependend on the programming language this may be done in different ways.

An example is OpenMP [14] where in the fourth version of this standard certain extensions were added that allow a programmer to specify (among parallelism, which is the main focus of OpenMP) that certain parts of a program should be vectorized by the compiler, including hints how to do that or assumptions that a compiler can rely on at that point of the program.

A small example for that is the piece of code shown in Figure 1. Here, the pragma tells the compiler to vectorize the loop and to handle the variable `s` as a reduction variable with a special treatement (this is necessary due to the loop carried data dependence on `s`).

The `simd` directive requests to vectorize that part of a program that is in the scope of this directive. For the `simd` directive there are additional clauses beside the shown **`reduction`** clause possible mainly assuring certain program properties. Among them are:

- `aligned` specifies that the specified data objects are aligned to a certain byte boundary.

```
double haddSum( __m256d tmp) {
   // vecA := ( x2 , x1 )
   const __m128d vecA = _mm256_castpd256_pd128(tmp);
   // vecB := ( x4 , x3 )
   const __m128d vecB = _mm256_extractf128_pd(tmp,1);
   // vecC := ( x4+x3 , x2+x1 )
   const __m128d vecC = _mm_hadd_pd(vecA,vecB);
   // vecS := ( x4+x3+x2+x1 , x4+x3+x2+x1)
   const __m128d vecS = _mm_hadd_pd(vecC,vecC);
   // returns x4+x3+x2+x1 as double
   return mm_cvtsd_f64(vecS);
}
```

Figure 2. Example code for the use of compiler intrinsics.

- `safelen` guaranties that n consecutive iterations can be executed in parallel / are independent.
- `linear` tells the compiler that the loop variable has a linear increase.

Similar compiler directives `simd` (vectorize code) and `ivdep` (ignore vector dependencies) outside of the OpenMP standard are known to several compilers with a similar meaning. We have seen no large differences in results for these alternatives.

An (non-OpenMP) directive that many compilers recognize in one or the other notation is the hint to unroll a loop [10]. This can be favorable if the loop body is rather small (as with the SpMV operation) and therefore the instruction pipeline runs soon out of instructions. Additionally, with a larger basic block a compiler may have more opportunities to optimize, e.g., to keep reused values in registers.

C. Vector Intrinsics

A compiler needs to generate special vector instructions to utilize the vector units in a processor. Sometimes a compiler may not be able to detect an appropriate situation because the data dependence analysis in a compiler cannot safely exclude any dependencies. Or a compiler generates sub-optimal code for that situation. In such situations, a programmer may himself “generate“ vector instructions by using so called vector intrinsics.

Vector intrinsics [18] are available with some widely used compilers, e.g., GNU g++ [33], Intel compiler icpc [8]. With these intrinsics a programmer has more or less direct access to vector instructions of the underlying hardware. But please be aware that this functionality is on the level of assembler intructions where one has to manage vector registers and vector instructions directly.

The example in Figure 2 shows how to add 4 values using vector intrinsics. `__m128d` and `__256d` are special vector types and `_mm...` are function calls that correspond to vector instructions. This small example makes it very clear that using intrinsic functionality makes a program hard to read / understand because hardware features are programmed embedded within a high level language like C or C++.

D. How to Choose the Right Program Optimization Strategy?

This spectrum of optimization techniques shown above has consequences for programmers. The first approach (use a compiler switch) leaves any decision and optimization to the compiler. This is a possibility that is quite comfortable for a programmer and does not require any sophisticated skills from a programmer.

The next possibility is to leave many things to the compiler but to give additional hints to the compiler using pragmas / directives. A compiler bases its decision concerning vectorizability (and many other optimizations) on data dependency information [13]. When a compiler cannot decide if a part of a program is optimiziable / vectorizable, the opportunities that a hardware architecture gives to dispose cannot be utilized. But with this approach a programmer needs experience and expert knowledge how a compiler works and what information it may miss in certain program parts. If a programmer assures wrong properties (e.g., safe distance of iterations) a compiler may even generate wrong code. If a programmer uses such directives the programming effort (additional lines of code) is rather small.

The last option is to allow a programmmer direct access to the functionality a hardware provides. This allows to utilize the available funcionality in an efficient way. Performance-aware programmers are used to such things. But this has severe consequences. One point is that the programming level is quite low and the resulting program is therefore hard to write and read. Additionally, programming is now getting platform specific, i.e., a program kernel developed and optimized for an Intel Haswell system is not executable on / not optimized for an older Ivy Bridge / Sandy Bridge system. This means, that any company using such advanced features has to provide an expert that is aware of all technological feature of hardware generations in use and how to use them.

Comfortability to the programmer is one aspect of consideration. If this would be the only aspect it is clear that the approach would be used to leave everything to the compiler. Many simulations in natural science run for hours, days or even weeks. Often a large part of the runtime is executed in rather small parts of the program, computationally intense program kernels like the above mentioned SpMV operation. For such really performance critical parts of a program all possibilities are analyzed that may lead to a decrease in runtime, even on the intrinsic level.

Therefore, the question is whether and if yes how much can a program benefit from programming techniques? Or is an optimizing compiler able to deliver the same (or even better) performance?

IV. Experimental Setup

To answer these questions we used the (rather small) compute intensive SpMV program kernel. We used our own implementation of the SpMV operation in C/C++ using the CSR storage format [1]. The basic implementation we use in our subsequent comparism is shown in Figure 3 (here shown in a rather simplified and compact version).

We used four different versions (as an information in parentheses the number of lines of code to realize that):

- *normal*: the unmodified version similar to the above shown (7 lines)
- *unroll*: the compiler was told with a directive to unroll the loop four times (8 lines)
- *simd*: the compiler was told with a directive to vectorize the code / to generate vector instructions (8 lines)

```
void SparseMatrixCSR::SpMV(Vector &v, Vector &u) {
  // iterate over all rows of the matrix
  for(int i=0; i<nRows; i++)
    // handle all non-zero elements in a row
    for(int j=rowStart[i]; j<rowStart[i+1]; j++)
      u[i] += values[j] * v[columnIndex[j]];
}
```

Figure 3. Basic code version for the SpMV operation (simplified).

TABLE I. SYSTEMS USED.

system name	SB	Haswell
instruction set	AVX	AVX2
architecture	Sandy Bridge EP	Haswell EP
processor (Intel Xeon)	E5-2670	E5-2680 v3
cyle time in GHz (TurboBoost)	2,6	2,5

- *intrinsics*: our own implementation using vector intrinsics (97 lines). The code contains a distinction, which vector instruction set AVX or AVX2 should be used and different intrinsics must be used in some parts of the program, dependent on the instruction set.

To measure performance numbers we use systems of different generations of Intel processors (Intel Sandy Bridge and Haswell). Table I gives an overview of relevant system parameters and systems names. The older Sandy Bridge generation supports only the AVX instruction set, in newer Haswell systems additional features are available in the AVX2 instruction set.

We used matrices as data sets with different properties that may influence performance, e.g., the distribution of non-zero values in a row. In total, 110 matrices were used. The matrices are taken form the Florida Sparse Matrix collection [35] and from the Society of Petroleum Engineers (SPE) challenge [36].

We used two compilers in recent versions:

- *g++*: GNU g++ vesion 5.2.0 [33]
- *icpc*: Intel icpc version 15.0.1 [8]

To filter accidental effects that may happen on any system each measurement was repeated 100 times and the median was taken as the measurement value.

V. EVALUATION

We discuss each optimization independently and summarize with an overall comparism. We give statistical values over all 110 matrices and additionally give a single absolute value for the SPE matrix `spe5Ref_a`, which shows often a similar behaviour compared to many other matrices and is used therefore as a representative.

A. Influence of Compilers and Compiler Levels

Both compilers in use provide compiler switches to turn on certain global optimization levels: `-O0` up to `-O3`. The optimization level 0 should be used for debugging only and not for production runs. The Intel compiler provides further an additional level `-fast` and the GNU compiler the option `-Ofast` to additionally turn on processor specific optimizations as well as interprocedural optimizations for the Intel compiler. But with this option the code eventually runs no longer on processors of previous generations while with the option `-O3` the code is still runnable on all recent systems. The default value for g++ is no optimization, the default for icpc is level 2.

TABLE II. RUNTIMES IN MILLISECONDS FOR VARIOUS COMPILER OPTIMIZATION LEVELS FOR THE EXAMPLE MATRIX `spe5Ref_a`.

	g++		icpc	
optimization option	SB	Haswell	SB	Haswell
-O0	342	295	379	308
-O1	173	139	150	139
-O2	96	87	150	137
-O3	93	82	150	136
-(O)fast	93	82	139	83

TABLE III. EFFECT OF UNROLLING (SEE TEXT FOR EXPLANATION OF ITEMS).

	g++		icpc	
	SB	Haswell	SB	Haswell
% instances better	80	72	22	14
runtime exa. matrix [ms]	90	81	153	162
minimum speedup	0.736	0.790	0.548	0.723
maximum speedup	1.129	1.700	1.293	1.111
average speedup	1.011	1.027	0.967	0.966
standard deviation	0.061	0.101	0.081	0.052

Table II shows as a representative example the results for the matrix `spe5Ref_a` on the Sandy Bridge and Haswell system. The results are transferable to the other matrices and are therefore general statements concerning our SpMV implementation. The performance of the GNU compiler generated code increases with each level while the performance of the Intel compiler is more or less good and nearly the same for all levels other than 0. The option `-fast` with the Intel compiler produces (processor specific) code that is significantly faster than the code of the other optimization levels. It should be noted that these results are specific to this program kernel and results may be different for other program kernels dependend on the source code and for other/future compiler versions.

As the compiler run itself does not take any significant amount of time for any optimization level it is recommandable always to invoke the compiler with a high optimization level. As there are no semantic problems with our code with level 3, we use this level in all following discussions and name it the base case.

B. Unrolling Loops

The next optimization technique we looked at is loop unrolling to enlarge basic blocks. This should optimize register usage and reduce the loop overhead for small loop bodies (as in our case). As already discussed, this can be used rather comfortable with directives specifying before a loop that this loop should be unrolled and giving an unroll factor. We found out empirically that an unroll factor of 4 performed best.

Table III shows results for using unrolling. The first line (% instances better) shows how many of the 110 matrices in percent showed a better performance result using this technique compared to the base case (with the same compiler). The next row (runtime exa. matrix) gives the runtime for the example matrix `spe5Ref_a` in milliseconds to be able to compare results directly. Minimum and maximum give the minimum / maximum speedup value compared to the base value. For example, a value of 0.736 for the minimum speedup means that the worst problem instance shows only 73.6 percent

TABLE IV. EFFECT OF `simd` DIRECTIVES (SEE TEXT ON UNROLLING FOR EXPLANATIONS OF ITEMS).

	g++		icpc	
	SB	Haswell	SB	Haswell
% instances better	1	0	56	73
runtime ex. matrix [ms]	147	158	148	145
minimum speedup	0.584	0.486	0.532	0.535
maximum speedup	1.076	1.000	1.121	1.326
average speedup	0.698	0.602	0.969	1,034
standard deviation	0.095	0.100	0.100	0.147

TABLE V. EFFECT OF INTRINSICS (SEE TEXT ON UNROLLING FOR EXPLANATIONS OF ITEMS).

	g++		icpc	
	SB	Haswell	SB	Haswell
% instances better	93	92	86	91
runtime ex. matrix [ms]	77	66	97	93
minimum speedup	0.805	0.631	0.565	0.622
maximum speedup	1.385	1.545	1.252	1.525
average speedup	1.153	1.184	1.093	1.237
standard deviation	0.107	0.145	0.128	1.196

of the performance of the base case. Average and standard deviation are the average speedup and standard deviation of the speedup over all 110 problem instances. An average of 1 and more means that the used technique performed in average better that the base case.

The results show that for both compilers the performance effect is more or less neglectable. The GNU compiler shows in average a minimal improvement while the Intel compiler shows instead a minimal performance degradation. This can be explained because the compilers themself apply already optimization techniques that make an explicit unrolling unfavorable.

C. Using Vector Directives

Table IV shows results for the use of OpenMP `simd` directives to explicitly request a vectorization. Enabling OpenMP vectorization with the GNU compiler results in a significant performance loss for nearly all problem instances. This may be attributed to the fact that the support for this OpenMP functionality is rather new in the compiler and causes rather performance-worsening code generation compared to the (good) optimization with the compiler level `-O3` of the base case. The Intel compiler shows in average rather similar results compared to the base case.

D. Using Intrinsics

Table V shows results for the use of (processor specific) intrinsics. The best absolute results were achieved for most matrices with this version. Only with few test matrices a small performance degradation could be seen as the matrix structure could not take advantage out of vector processing. For the Intel compiler there is a significant boost in performance as well. But here it is fair to say that the Intel compiler option`-fast` generates also processor specific code like with the intrinsics and is even faster.

E. Summary

This section summarizes the results. Table VI summarizes the absolute run times for the various optimizations on the example matrix. Figure 4 shows the percentage of problem

TABLE VI. SUMMARY OF RUN TIMES IN MILLISECONDS FOR THE EXAMPLE MATRIX.

	g++		icpc	
	SB	Haswell	SB	Haswell
-O0	342	295	379	308
-O1	173	139	150	139
-O2	96	87	150	137
-O3	93	82	150	136
-(O)fast	93	82	139	83
unrolling	90	81	153	162
simd	147	158	148	145
intrinsics	77	66	97	93

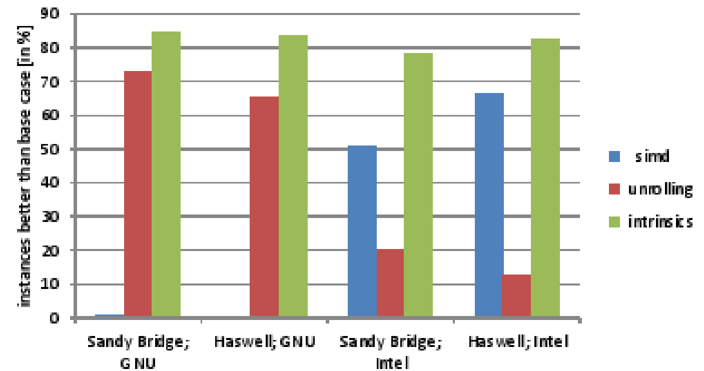

Figure 4. Percentage of instances that performed better than the base case.

instances that got a performance gain when applying an optimization technique.

The first technique used was compiler flags. The additional effort and needed knowledge for using compiler flags is minimal and no code change is necessary. The results were twofold: The Intel compiler uses already optimization techniques with lower optimization levels and has not shown any further improvements with higher optimization levels. Only the option `-fast` (producing processor specific code) resulted in an additional performance boost. For the GNU compiler, there was a steady performance increase with higher optimization levels.

The unrolling technique did not show any major change in runtime. Here, the compilers did already a good job. The programming effort and necessary expertise to use it is quite low.

Using the `simd` compiler directive to ask for vectorization is easy to use but requires a deeper knowledge, e.g., on data dependencies to avoid wrong code generation. The performance reached with the GNU compiler on the beside the directive unmodified code was worse compared to the base case. Using the Intel compiler this directive has no significant influence.

To use intrinsics a deep understanding of a processor architecture and the available instruction set is necessary. Additionally the algorithm may be quite different to the normal version when expressing it on an intrinsic level. The program code is totally different to the original code and quite hard to read and write, at least for a programmer who is not used to intrinsics. But the overall best performance results were reached with intrinsics.

In a nutshell, all optimizations other than `simd` directives with the GNU compiler and loop unrolling with the Intel

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-461-9

compiler had a positive influence for most problem instances. Using intrinsics resulted in 80% and more problem instances for a performance improvement.

VI. Conclusions

SpMV is a time critical operation in many applications. Optimizing this operation is a challenge. There are various opportunities to tackle that problem on a program optimization level, partially dependend on the compiler used.

In this paper, several optimization approaches were described and compared to each other concerning programming effort / required expert knowledge and achieved performance. It was shown that using a simple compiler switch for the highest level of optimization a compiler can often get near to 80% of the best performance reached with any other approach. Unrolling is easy to use but did not show any significant performance effect. Using OpenMP `simd` directives showed only a small performance impact, although some problem instances gained more performance. Using this directives with the GNU compiler is currently not encouraged. Rewriting the program kernel with vector intrinsics means a total rewrite of the code, which is only acceptable for really compute intensive and rather small program kernels. But the performance gain can be substantially increased and this approach resulted in the best performance for the SpMV operation.

Acknowledgements

Jan Ecker, Javed Razzaq and Simon Scholl at Bonn-Rhein-Sieg University helped us in many discussions. We would also like to thank the CMT team at Saudi Aramco EXPEC ARC for their support and input. Especially we want to thank Ali H. Dogru for making this research project possible. We would like to thank the anonymous reviewers for their suggestions and comments.

References

[1] Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd ed. SIAM, 2003.

[2] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiply on modern processors with wide SIMD units," SIAM Journal on Scientific Computing, vol. 26, no. 5, 2014, pp. C401–423.

[3] W. Liu and B. Vinter, "CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in Proc. 29th Intl. Conference on Supercomputing (ICS'15). ACM, 2015, pp. 339–350.

[4] K. Li, W. Yang, and K. Li, "Performance analysis and optimization for SpMV on GPU using probalistic modeling," IEEE Trans. Parallel and Distributed Systems, vol. 26, no. 1, Jan. 2015, pp. 196–205.

[5] W. Liu and B. Vinter, "Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors," preprint arXiv:1504.06474v1, 2015.

[6] J. Wong, E. Kuhl, and E. Darve, "A new sparse matrix vector multiplication GPU algorithm designed for finite element problems," Intl. Journal for Numerical in Engineering, Jan. 2015, pp. 1–35.

[7] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 5th ed. Morgan Kaufmann Publishers, Inc., 2012.

[8] User and Reference Guide for the Intel C++ Compiler 15.0, https://software.intel.com/en-us/compiler_15.0_ug_c ed., Intel Corporation, 2014, retrieved: February 2016.

[9] LLVM Project, The LLVM Compiler Infrastructure, http://llvm.org/, retrieved: February 2016.

[10] R. Allen and K. Kennedy, Optimizing Compilers for Modern Architectures. San Francisco: Morgan Kaufmann, 2002.

[11] K. D. Cooper and L. Torczon, Engineering a Compiler, 2nd ed. Burlington, MA: Morgan Kaufmann, 2012.

[12] S. S. Muchnick, Advanced Compiler Design and Implementation. San Francisco: Morgan Kaufmann, 1997.

[13] U. Banerjee, "An introduction to a formal theory of dependence analysis," The Journal of Supercomputing, vol. 2, 1988, pp. 133–149.

[14] OpenMP Application Program Interface, 4th ed., OpenMP Architecture Review Board, http://www.openmp.org/, Nov. 2015, retrieved: February 2016.

[15] Cray Research Inc., CF90 Commands and Directives Reference Manual, 1995, sR-3901 2.0.

[16] R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, and D. Padua, "Restructuring Fortran programs for Cedar," Concurrency - Practice and Experience, vol. 5, no. 7, Oct. 1993, pp. 553–573.

[17] K. A. Tomko and S. G. Abraham, "Data and program restructuring of irregular applications for cache-coherent multiprocessors," in Proc. ACM Int'l Conf. Supercomputing, Jul. 1994, pp. 214–225.

[18] Intel Intrinsics Guide, https://software.intel.com/sites/landingpage/IntrinsicsGuide/ ed., Intel, 2015, retrieved: February 2016.

[19] M. Corden, Requirements for Vectorizable Loops, Intel, https://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/, 2012, retrieved: February 2016.

[20] Intel® 64 and IA-32 Architectures Optimization Reference Manual, Intel, http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html, Sep. 2014, retrieved: February 2016.

[21] Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture, Intel, http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf, Sep. 2014, retrieved: February 2016.

[22] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Understanding the performance of sparse matrix-vector multiplication," in Proc. 16th Euromicro Intl. Conference on Parallel, Distributed and Network-based Processing (PDP'08), 2008, pp. 283–292.

[23] E. Saule, K. Kaya, and U. V. Catalyrek, "Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi," in Proc. Parallel Processing and Applied Mathematics (PPAM 2013), 2013, pp. 559–570.

[24] R. W. Vuduc, "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, University of California, Berkeley, 2003.

[25] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "Performance modeling and analysis of cache blocking in sparse matrix vector multiply," University of California at Berkeley, EECS Department, Tech. Rep. UCB/CSD-04-1335, 2004.

[26] F. Wende, "SIMD enabled functions on Intel Xeon Phi CPU and Intel Xeon Phi coprocessor," Konrad-Zuse Zentrum für Informationstechnik Berlin, Tech. Rep. ZIB-Report 15-17, Feb. 2015.

[27] G. Chrysos, Intel® Xeon Phi™ Coprocessor – The Architecture, https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner, 2012, retrieved: February 2016.

[28] Intel, Intel® 64 and IA-32 Architectures Optimization Reference Manual, 2014.

[29] AMD64 Architecture Programmers Manual. Advanced Micro Devices, 2013, vol. 3: General-Purpose and System Instructions.

[30] ARM, ARM v8 Architecture Reference Manual, 2010.

[31] M. Flynn, "Some computer organizations and their effectiveness," IEEE Trans. Computers, vol. C-21, 1972, pp. 948–960.

[32] Skylake (microarchitecture), Wikipedia, http://en.wikipedia.org/wiki/Skylake_(microarchitecture), retrieved: February 2016.

[33] GNU GCC, "GCC, the GNU Compiler Collection," retrieved: February 2016. [Online]. Available: https://gcc.gnu.org/

[34] PGI Compilers and Tools, https://www.pgroup.com/, retrieved: February 2016.

[35] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," ACM Trans. Math. Softw., vol. 38, no. 1, Nov. 2010, pp. 1:1–1:25.

[36] SPE Comparative Solution Project, Society of Petroleum Engineers, http://www.spe.org/web/csp/.