SpMV Runtime Improvements with Program Optimization Techniques on Different Abstraction Levels

Rudolf Berrendorf
Computer Science Department
Bonn-Rhein-Sieg University
Sankt Augustin, Germany
e-mail: rudolf.berrendorf@h-brs.de

Max Weierstall
Computer Science Department
Bonn-Rhein-Sieg University
Sankt Augustin, Germany
e-mail: max.weierstall@h-brs.de

Florian Mannuss
EXPEC Advanced Research Center
Saudi Arabian Oil Company
Dhahran, Saudi Arabia
e-mail: florian.mannuss@aramco.com

Abstract—**The multiplication of a sparse matrix with a dense vector is a performance critical computational kernel in many applications, especially in natural and engineering sciences. To speed up this operation, many optimization techniques have been developed in the past, mainly focusing on the data layout for the sparse matrix. Strongly related to the data layout is the program code for the multiplication. But even for a fixed data layout with an accommodated kernel, there are several alternatives for program optimizations. This paper discusses a spectrum of program optimization techniques on different abstraction layers for six different sparse matrix data format and kernels. At the one end of the spectrum, compiler options can be used that hide from the programmer all optimizations done by the compiler internally. On the other end of the spectrum, a multiplication kernel can be programmed that use highly sophisticated intrinsics on an assembler level that ask for a programmer with a deep understanding of processor architectures. These special instructions can be used to efficiently utilize hardware features in processors like vector units that have the potential to speed up sparse matrix computations. The paper compares the programming effort and required knowledge level for certain program optimizations in relation to the gained runtime improvements.**

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

I. Introduction

This paper is an extended version of the conference paper [1]. Sparse matrices arise from the discretization of partial differential equations and are therefore widely used in many areas of natural and engineering sciences [2], especially in simulations. Examples of applications areas are mechanical deformation, fluid flow and electromagnetic wave propagation. 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 [2]), which are among the most time consuming operations in many simulations. Therefore, much attention has been given to optimize this operation. As SpMV is a memory bandwidth bound operation and as sparse matrices can get very large, much attention has to be given to the management and access to the matrix data [3], [4]. One important point in an optimization discussion is the choice of an appropriate storage format for the sparse matrix. More than 50 storage formats have been published in the past, among them [5]–[10]. The choice of format depends mainly on the given matrix structure (e.g., diagonal, high / low matrix bandwidth, etc.) and the target architecture, e.g., multicore CPU, multiprocessor system, Graphics Processor Unit (GPU). For example, Compressed Sparse Row (CSR) [2] is a rather general storage format for sparse matrices that performs quite well on CPU-based systems and is used in many applications. Strongly related to a storage format is the way how the SpMV operation is actually implemented, i.e., how the data stored in the format is processed. This computational kernel is mainly a traversal over the data structures in a certain way given by the storage format. But even for a fixed storage format like CSR, there are quite different ways how to program the CSR kernel for the SpMV, with opportunities for program optimization on different abstraction levels. This paper deals with the spectrum of opportunities and discusses some alternatives, their programming effort, the required level of expertise and the achieved performance gain.

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 [11]. 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 [12], [13] 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 [14]–[18], as long as all data dependencies are respected [19]. A general problem with many SpMV implementations is, that the SpMV kernel is quite small, often only a single or a few lines of code surrounded by one or two loops, and therefore the basic block is rather small. Fig. 1 shows as an example a simplified and non-optimized basic version of a SpMV operation appropriate for the CSR format.

Some high level programming models, especially designed for parallel (and therefore resource intensive) computing, have

```
void SpMV_Basic(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 1. Basic code version for the SpMV operation for the CSR data format (simplified).

some notations to give hints to a compiler concerning the vectorization of code. For example, OpenMP [20] as the de facto standard for shared memory parallelism has got in the recent version 4 some annotations to guide a compiler in using vector units in a processor. But vectorizing compilers and directives to give a compiler hints have a long history and existed already before OpenMP [21]. Such annotations can be used to speed up computational kernels like the SpMV to supply a compiler in a non-standardized way with additional semantic information, if necessary.

One way to optimize a program is to use certain optimization options of a compiler, e.g., a single meta option `-O3` enables already many individual compiler optimizations. Other than leaving all optimizations to a compiler, there are program optimization techniques known that allow to restructure 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], [22], [23].

And, in a fourth 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 [24]. 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 that 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 in more detail 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

There are several dimensions of related work.

Compiler writers give hints in user guides [14], [25], [26] and technical notes [27] how to optimize programs and how to write programs in a way such that a compiler can apply optimizations. Further than that, there exist often optimization guides with a detailed description of hardware features that a programmer can use and should use to get performance [28]–[30].

In [31], Wende discusses the use of SIMD (Single Instruction Multiple Data [12]) functions, i.e., vector intrinsics, to improve the performance on Intel Xeon processors and Xeon Phi coprocessors [32] especially 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.

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 [33]–[36].

There exist various reference implementation of storage formats for sparse matrices where a highly optimized SpMV code uses vector intrinsics. Among them are the formats / implementations CSR5 [6] / [37] and CSB [38] / [39]. As only an implementation exists that uses intrinsics, there is neither a comparism of programming effort in relation to runtime improvement nor an estimation of programming effort. But just to give the reader an estimate how complex such an optimization can be to optimize the SpMV operation for a certain platform, the reference implementation of yaSpMV [40] can be used. The (non-optimized) code for a CPU has 16 lines of C++ code while the highly optimized version for a GPU has approx. 700 lines of rather complex code.

Code optimization is often a multi-dimensional problem, as various architectural features likes register usage, vector units, caches, pipeling among others combined with implementation parameters of storage formats for sparse matrixes (e.g., slice sizes, blocking) influence each other. Different to a manual code optimization done by a programmer using to his knowledge a good combination of such parameters, auto tuning is a way to explore such a large parameter space automatically through extensive offline testing and eventually an additional and faster online testing finding experimentally a good combination of all parameters that fit to the given architecture and given sparse matrix. For the SpMV operation, there exist various implementations that use this technique, e.g. Poski [35], [41], CSX [42] and yaSpMV [40]. The programming effort to generate such an auto tuning framework is very high. The overhead to determine good hardware parameter values out of a large space of possible values can be high, but this is not important as such an exhaustive search has to be done once per system and offline. But also the online runtime overhead can be quite substantial that has to be done once when the non-zero structure of a sparse matrix is known. Often this is only profitable if many SpMV exectutions are done in the following with a fixed matrix structure and the tuning overhead can be compensated with an appropriate runtime improvement for the SpMV operation. Such research is more related to automate code optimization for certain hardware architectures rather than relating optimization techniques to each other.

Additional work was done to select good parameter values in a large parameter space for the SpMV using machine learning techniques. In [43] Lehnert et al. use linear regression, gradient boosting and k Nearest Neighbor techniques to decide at runtime, which matrix format / SpMV kernel should be used for a given matrix and, which thread mapping should be used, dependend on a few statistical parameters describing the matrix structure. In an offline training phase data is gathered in many program runs. While the runtime for these offline tests can be quite substantial (but need to be done only once), the overhead for the *online* decision is quite low (in the order of 10 % of a SpMV operation) for the linear regression / gradient boosting approach while the k Nearest Neighbor approach has a larger runtime overhead that can be in the order of tens to hundreds of SpMV operations. Additional work in using machine learning techniques to improve the SpMV execution was done by Sedaghati et al. [44], [45] using decision trees. Li [7] uses a probalistic approach for that.

III. Program Optimizations

Nowadays, processor and memory architectures are rather complex. Many architectural optimizations 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, hardware prefetching and many others [12], [13]. For all enumerated hardware optimizations it is favorable to have large basic blocks (code without any branch).

A significant performance boost for many applications is the use of vector registers / units that are available in almost all recent processor architectures [28], [46], [47]. These vector architectures follow the well-known SIMD principle [48] 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 to 4 double precision elements that are mostly used in scientific simulations, each 64 bits. Recent announcements show [49], [50] that the vector width will double in the near future with a nominal floating point performance increase of a factor of two. The SpMV operation is eligible to utilize vector units. How efficient such vector units can be utilized depends mainly on the chosen storage format and non-zero structure of the matrix. The CSR format is an example of such a format that can benefit from vector units. Vector units and instruction sets utilizing such units have evolved over time. Newer processor architectures of the Intel family have vector registers and units of size 256 bits and support the AVX (Advanced Vector Extensions) instruction set [51] (processor lines Sandy Bridge EP and Ivy Bridge EP) or the instruction set AVX2 (the most recent Haswell EP and Broadwell EP). Among other things, AVX2 enhances AVX with additional instructions for integer operations and fused multiply add that is of interest with SpMV operations.

The enlargement of basic blocks in a loop body and the use of vector registers are two techniques that can be used to speed up a SpMV operation. There are now certain levels of abstraction on which a programmer may influence these and other optimizations. In the following, in more detail opportunities are discussed a programmer may use to speed up the SpMV operations (and others).

A. Compiler Flags

A simple optimization strategy is to leave any optimization to a compiler. This is the strategy used most often by nearly all programmers. A programmer may specify on a rather coarse scope of one complete source file a general compiler optimization level (i.e., `-O0, -O1, -O2, -O3, -Ofast` as available with most compilers) leaving any detailed decisions and optimization strategies related to that optimization levels solely to the compiler according to the specified optimization level. Such flags are merely meta flags turning on/off a bunch of optimizations or a finer level specific to a compiler. All compiler understand the same meta flags, but the exact meaning of these flags (i.e., which specific optimizations are turned on) is open to a compiler.

An optimization level of `-O0` disables any optimization and is only useful for debugging and should not be used for productions runs. Specifying an optimization level of `-O1` enables basic optimizations that are often sufficient to generate efficient code for programs that have a not too complex program structure. A level of `-O2` instructs many compilers to enable more, advanced and more costly optimization techniques but 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 arithmetic. A level of `-O3` often includes additional optimizations that allows the use of operations that may change (slightly) the meaning of a program; therefore this level has to be chosen with care. Even further is the optimization level `-fast` that enables optimizations that may change the semantic of a program using faster arithmetic or even speculative execution, generate processor specific code (that may not execute on processors of a previous generation), and do interprocedural and link-time optimizations that may take significantly more compile/link time as with other optimization levels.

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 of a specific processor generation. For example, the compiler option `-march=haswell` of the GNU compiler g++ [25] allows the generation of advanced instructions only available on Intel 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 a rather old processor family to allow the compiled program to run on many systems, even older ones.

The PGI compiler [26] 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.

Usually, compilers have flags to generate reports on various levels of details what they could optimize, what not, and in this case why. For example, the Intel compiler generates with the options `-qopt-report=5 -qopt-report-phase=vec` a very detailed report with information concerning the vectorization of code. An example for a report for a simple SpMV implementation with two nested loops is shown in Fig. 2. According

```
LOOP BEGIN at SparseMatrixCSR.cpp(568,3)
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at SparseMatrixCSR.cpp(573,5)
   <Peeled loop for vectorization>
   LOOP END

   LOOP BEGIN at SparseMatrixCSR.cpp(573,5)
      remark #15388: vectorization support: reference this has aligned access   [ SparseMatrixCSR.cpp(575,7) ]
      remark #15389: vectorization support: reference this has unaligned access   [ SparseMatrixCSR.cpp(575,7) ]
      remark #15381: vectorization support: unaligned access used inside loop body
      remark #15305: vectorization support: vector length 2
      remark #15399: vectorization support: unroll factor set to 4
      remark #15309: vectorization support: normalized vectorization overhead 0.533
      remark #15300: LOOP WAS VECTORIZED
      remark #15442: entire loop may be executed in remainder
      remark #15448: unmasked aligned unit stride loads: 1
      remark #15450: unmasked unaligned unit stride loads: 1
      remark #15458: masked indexed (or gather) loads: 1
      remark #15475: --- begin vector loop cost summary ---
      remark #15476: scalar loop cost: 14
      remark #15477: vector loop cost: 7.500
      remark #15478: estimated potential speedup: 1.810
      remark #15488: --- end vector loop cost summary ---
   LOOP END

   LOOP BEGIN at SparseMatrixCSR.cpp(573,5)
   <Remainder loop for vectorization>
   LOOP END
LOOP END
```

Figure 2. Example of a detailed Vectorization Report as given by the Intel Compiler.

to this report, the inner loop was vectorized and the outer loop not. The way how the information is presented makes clear that such a report should only be used by an experienced programmer who is aware of the meaning of the terms used in the report and the consequences that a programmer should take.

B. Loop Unrolling

The SpMV operation is an operation that often contributes to a large portion (e.g., 50 %) of the runtime of a simulation that itself can run for hours, days or even weeks. Therefore performance aware programmers are willing to spend time to speed up such computational kernels if a compiler would not be able to do so.

Unfortunately, the non-optimized SpMV has for many matrix storage formats a rather small loop body of the innermost loop (one or a few lines of code) as, for example, shown in Fig. 1. This means that all these hardware optimizations described in the introduction of this section cannot be utilized efficiently if a compiler cannot handle this by itself, i.e., enlarging the innermost loop body to a larger basic blocks.

A well-known technique called loop unrolling [16], [17] enlarges the basic block of a loop body. This can be favorable if the loop body is rather small (as for most formats 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 index values in registers.

Loop unrolling can be realized manually by a programmer (which is often tedious and error-prone), by using appropriate directives / annotations in a code that instruct a compiler to unroll a loop by a certain factor, or internally by a compiler without a programmer's intervention. Explicit loop unrolling using directives will be used as one of the optimization techniques discussed later, which could be profitable if a compiler is not able to enlarge a basic block by himself. An example for an unrolled loop using such directives is given in Fig. 3.

```
void SpMV_Unroll(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
#pragma unroll(4)
    for(int j=rowStart[i]; j<rowStart[i+1]; j++) {
      u[i] += values[j] * v[columnIndex[j]];
    }
  }
}
```

Figure 3. Example of an explicitly unrolled loop using a directive.

C. Language Directives for Vectorization

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 cannot 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

```
void SpMV_SIMD(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
    double s = 0;
#pragma omp simd reduction(+:s)
    for(int j=rowStart[i]; j<rowStart[i+1]; j++) {
      u[i] += values[j] * v[columnIndex[j]];
    }
  u[i] = s;
  }
}
```

Figure 4. Example of the use of a OpenMP SIMD directive.

```
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 5. Example code for the use of compiler intrinsics.

information. Dependent on the programming language or compiler this may be done in different ways.

An example is OpenMP [20] 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 Fig. 4. Here, the pragma tells the compiler to vectorize the inner loop and to handle the variable `s` as a reduction variable with a special treatment (this is necessary due to the loop carried data dependence on `s`).

The `simd` directive requests an OpenMP compiler 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.
- `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 performance results for these alternatives.

As explained already before, a programmer can be guided for using such directives in an appropriate way by looking at a compiler report that tells whether a piece of code could be vectorized or not (see Fig. 2 for an example). If code could not be vectorized, the reason for that is also given. But this statement must be restricted as the output is often presented in a way that most times only an experienced programmer that understands how a compiler works internally can understand this information in all details.

To use the vector directives in our code, it was necessary with appropriate directives to assure the compiler that the vectors used were aligned (and which must be the case). Additionally, it was necessary (taken from the output of the vectorization report) to copy values of C++ member variables to block local variables. Otherwise, no vectorization of the code took place.

D. 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 the 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 [24] are available with some widely used compilers, e.g., GNU g++ [25], Intel compiler icpc [14]. 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 instructions where one has to manage vector registers and vector instructions directly. Also at most (or better exactly) 4 double values have to be handled in parallel for recent vector units. This means that the code has usually an additional loop that iterates in an appropriate way over blocks of 4 consecutive double values. Needless to say that such an intrinsic code looks quite different to an original code in a high-level programming language.

The example in Fig. 5 shows how to add 4 values using vector intrinsics (this is a small sub problem of a SpMV operation). `__m128d` and `__256d` are special vector types that must be used 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++.

E. How to Choose the Right Program Optimization Strategy?

As already explained, performance aware programmers are willing to write rather complex code if an operation like the SpMV that contributes to a large portion of the runtime of a program can be speed up. The 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 unless options are chosen that may influence the semantics of a program. If this approach produces the most efficient code, this should be the way to go.

The next possibility is to leave many things to the compiler but to give additional hints to the compiler using pragmas / directives at a finer granularity (e.g., with the scope of one loop). A compiler bases its decision concerning vectorizability (and many other optimizations) on data dependency information [19]. When a compiler cannot decide if a part of a program is optimizable / vectorizable, the opportunities that a hardware architecture gives to dispose cannot be utilized. But giving additional hints to a compiler, a programmer needs experience and expert knowledge how a compiler works and what information it may miss in certain parts of a program. 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 but the required level of expertise is high.

The last option is to allow a programmer direct access to the functionality a hardware provides. This allows to utilize the available functionality in an efficient way. Although this may look for a normal programmer scary, 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 (see the example in Fig.r̃effig:Intrinsics). 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 in their programs has to provide an expert that is aware of all architectural features of hardware generations in use and how to use them properly. Additionally, different code versions have to be maintained.

Comfortability to the programmer is one aspect of consideration. If this would be the only aspect it is clear that the approach that would be used is 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 at this point is whether and if yes how much can a program benefit from optimization techniques in the spectrum discussed above? Or is an optimizing compiler able to deliver the same (or even better) performance? And what is the programming effort in relation to a possible gain in performance?

The discussion and the following evaluation is done for a sequential program version. We have seen, that the results discussed in the following sections are transferable to parallel programming models like OpenMP [20] as well. But additional problems have to be handled as well, like processor locality, thread mapping and load balance. A comparism of using different parallel programming models for the SpMV on a GPU can be found, for example, in [52].

IV. Experimental Setup

To answer these questions raised above the rather small SpMV program kernel was used. As this kernel has only (dependent on the matrix storage format and the optimization technique) few lines of code, the influence of the optimization techniques could be clearly seen.

TABLE I. SYSTEMS USED.

system name	SB	HW
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

Sparse matrices can get in production runs very large (for example, up to 10^9 rows). They are stored is an appropriate storage format that takes advantage of the sparsity and the non-zero structure of a sparse matrix. A proper SpMV kernel code is needed that fits the storage format. We used our own implementation of the SpMV operation in C++ using storage formats of different code complexity. Beside the rather simple structured storage formats COO and CSR [2], we used also more sophisticated storage formats with more complex SpMV kernels, namely BRO-ELL [53], SELL-C-σ [5], VBL [54] and ESB [55]. An example for a basic implementation of the simple CSR format we use in our subsequent comparism was already shown in Fig. 1 (in a rather simplified and compact version).

We used four different versions for the optimizations for each format. As an information in parentheses the number of lines of code for the CSR format to realize that:

- *normal*: the unmodified version similar to the version in Fig. 1 (9 lines)
- *unroll*: the compiler was told with a directive to unroll a loop four times, similar to the version shown in Fig. 3 (12 lines)
- *simd*: the compiler was told with a directive to vectorize the code / to generate vector instructions similar to the version shown in Fig. 4 (16 lines)
- *intrinsics*: our own implementation using vector intrinsics (68 lines). The code often contains distinctions, 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.

As data sets we used sparse matrices with different properties that may influence the performance of a SpMV operation. For example, the distribution of non-zero values over the matrix and in a row may have an influence on the utilization of vector units and loop unrolling. In total, 111 matrices were used. The matrices are taken form the Florida Sparse Matrix collection [56] and from the Society of Petroleum Engineers (SPE) challenge [57].

We used two compilers in recent versions:

- *g++*: GNU g++ version 5.3.0 [25]
- *icpc*: Intel icpc version 16.0.2 [14]

TABLE II. PARAMETER SPACE IN THE EVALUATION.

parameter	count
processor families	2
compilers	2
matrix formats	6
test matrices	111
SpMV iterationsper run	100
optimization techniques	4

In the compiler options we always specified the target architecture to allow processor specific optimizations. In difference to the program version discussed in [1], additional code was added / changed to enable more compiler optimizations. These changes that were necessary to improve even further the runtime of the SpMV operation are partially discussed in the following chapter. It should be noted, that without these code optimizations that are now already incorporated into the basic version this version would perform substantially slower in certain cases.

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

Table II gives an overview over the parameter space in our evaluation. It follows that for our experimental setup $2 \times 2 \times 6 \times 111 \times 100 \times 4 = 106,560$ SpMV executions were executed.

V. EVALUATION

Each optimization is discussed independently and afterwards an overall comparism is done. For generalizable statements, statistical result values over all 111 matrices are given. Additionally, absolute result values are given for one single matrix, the SPE matrix `spe5Ref_a`, which shows often a similar behavior compared to many other matrices. This matrix is used as a representative for more detailed analysis.

After a discussion of the influence of compiler options, the results for the three optimization techniques unrolling, vector directives and intrinsics are each presented in a common way:

1) a chart showing the percentage of test instances with a runtime improvement,
2) a chart with average speedup values over all matrices.

The charts differentiate between the matrix storage formats.

A. Influence of Compilers and Compiler Levels

Both compilers in use provide comfortable 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 and link time 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.

To evaluate the influence of an optimization level, a basic SpMV kernel version was used for each sparse matrix storage format, but already modified in a way that a compiler can generate efficient code out of this (see remarks above). This leaves many opportunities for a compiler but places also challenges to the compiler to detect how the code looks like (in the sense of optimization potential) and what the best alternative for code generation is.

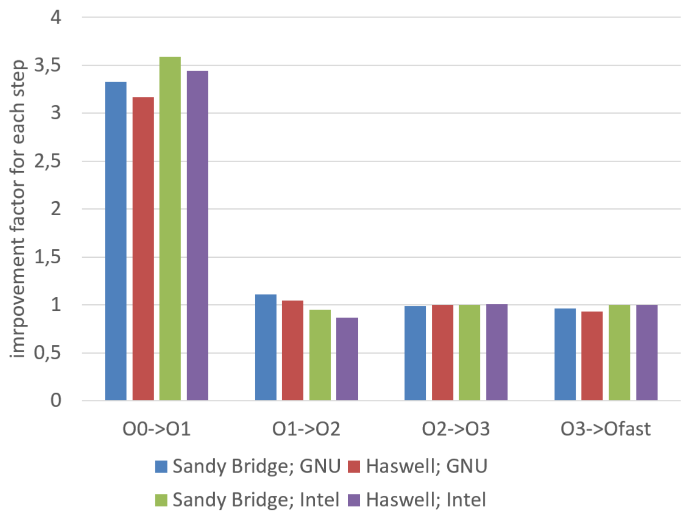

Figure 6. Average runtime improvements for each additional optimization level over all matrices.

Table III shows as a representative example the detailed results for the matrix `spe5Ref_a` on the Sandy Bridge system and Haswell system. The results are transferable to the other matrices and are therefore general statements concerning our SpMV implementation. Both compiler show a significant performance increase going from `-O0` to `-O1`. Again the remark, that the level `-O0` should only used for debugging purposes and not used in any production version. The further transition to `-O2` shows some minor additional performance increase with the GNU compiler and s small peromance degradation with the Intel compiler. But all levels above `-O2` show no or only a small increase in performance. The compiler do already optimizations with `-O2` and even `-O1` that contribute to the best runtime performance. An older version of the Intel compiler that was used in [1] has shown a significant increase in performance on a Haswell system using the compiler option `-fast` that could be attributed to the fact that only with this option architectural features of the Haswell processor generation were utilized. The recent Intel compiler version utilizes the AVX2 units already with lower optimization levels. There are also a few cases where a lower optimization level (sometimes even the level `-O1`) produces a better result than a higher level. And in some of these cases, the difference was quite significant. This can be seen, for example, in Table III with the SELL-C-σ format. Here, compilers do aggressive optimizations that tend to be counter-productive in this cases.

While Table III shows absolute performance numbers for one example matrix and three matrix formats, Fig. 6 shows summarized statistics over all matrices and all 6 matrix formats. In this table, the average improvement factor over all matrices is shown going from one optimization level to the next. A factor higher than 1 means an average increase in performance, a factor lower than 1 means a performance drop. As already seen and discussed with the single example matrix, the transition to level `-O1` shows a huge increase

TABLE III. RUNTIMES IN MILLISECONDS FOR VARIOUS COMPILER OPTIMIZATION LEVELS FOR THE EXAMPLE MATRIX `spe5Ref_a` AND SELECTED FORMATS.

	Intel Sandy Bridge (SB)						Intel Haswell (HW)					
	g++			icpc			g++			icpc		
	CSR	BRO-ELL	SELL-C-σ	CSR	BRO-ELL	SELL-C-σ	CSR	BRO-ELL	SELL-C-σ	CSR	BRO-ELL	SELL-C-σ
-O0	213	737	372	223	769	388	190	603	337	193	624	357
-O1	63	171	121	58	208	118	55	115	107	53	132	105
-O2	57	156	118	53	190	126	61	106	106	53	126	139
-O3	57	156	118	52	191	126	60	105	105	52	126	139
-(O)fast	57	156	118	53	192	125	77	106	129	53	126	138

in performance, as expected. But interestingly, with these compiler versions there is in average no additional increase in performance for higher levels of optimization and even sometimes a small degradation (for example, in the last column of Table III again with the SELL-C-σ format).

Whether a compiler can detect in the source code opportunities for optimizations or fail on this can have a significant influence on the runtime of a program, especially when small parts of the source code (few lines of code of a SpMV kernel) contribute to a major part of the program's runtime. This can be seen with the results for the CSR format as given in Table III. In a previous paper [1] large differences in runtime were reported for the Intel compiler with the CSR SpMV kernel switching from optimization level `-O3` to `-fast`. In a deeper analysis of the compiler generated code it was later found that a rather simple overloaded and inlined array access operator `[]` in C++ that was defined to abstract from the concrete realization of the storage format in the code contributed to a significant drop of nearly 50 % in performance for all optimization levels other than `-fast` with the Intel compiler compared to the GNU compiler. The Intel compiler could not handle this piece of code (which is from a programmer's point of view rather simple) with all optimization levels other than `-fast` while the GNU compiler could handle this even for lower optimization levels. After replacing the overloaded operator `[]` with a pointer copy and a normal C++ `[]` operator also the Intel compiler could optimize this, which can be seen in the CSR results given in Table III. Fig. 7 shows the relevant modified code. It should be pointed out that this "optimization" that was necessary because otherwise a compiler could not optimize a code, breaks software abstraction that is very import in software development. With this solution the concrete storage format of a vector is no longer hidden by a class.

In all following discussions, an optimized code (see, for example, the discussion with overloaded array access operator) together with a optimization level `-O3` is used as the default option and the effect of the other optimization techniques are related to these results.

```
void SparseMatrixCSR::SpMV2(const Vector &v, Vector &u) {

  // Faster access to vector data, but loosing abstraction.
  // u[i] and v[i] with an overloaded [] operator
  // prevented compiler optimizations with certain compilers
  // Now copy the values-pointer of the vector class
  // to a local pointer variable.
  const double *const vValues = v.getValues();
  double *const uValues = u.getValues();

  // iterate over all rows of the matrix
  for(int i=0; i<nRows; i++) {
    // handle all non-zero elements in a row
    for(index_t j=rowStart[i]; j<rowStart[i+1]; j++) {
      uValues[i] += values[j] * vValues[columnIndex[j]];
    }
  }
}
```

Figure 7. Loosing software abstraction to handle compiler's disability to optimize code (simplified version shown).

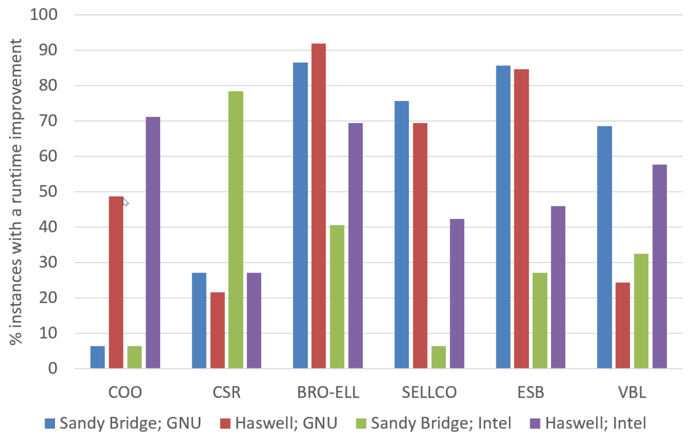

Figure 8. Percentage of Instances that showed a runtime improvement with loop unrooling.

B. *Unrolling Loops*

Different to leaving everything to the compiler with a single compiler option, loop unrolling is used here as the first explicit optimization. It is used to enlarge basic blocks as most SpMV kernels are rather small. This enables a compiler at the basic block level (i.e., without the need to understand complex loop structures) to optimize register usage, a better utilization of functional units and a reduction of the loop overhead for small loop bodies, as is in our case for all matrix formats. As already discussed, this can be used rather comfortable with directives specifying before a loop that this loop should be unrolled, specifying optionally an unroll factor as a parameter. We found out empirically that an unroll factor of 4 performed best.

Fig. 8 shows the percentage of instances that showed a runtime improvement, differentiated by matrix format. Figure 9 shows the average speedup that can be achieved using this technique. As can be seen in the figures, the influence of unrolling on the performance is minimal, all speedups are near to 1. This can be seen also looking at individual speedup results of the formats, which are not shown here. The conclusion is here that both compilers use already techniques that are able to enhance basic blocks for the reasons stated above, e.g. doing

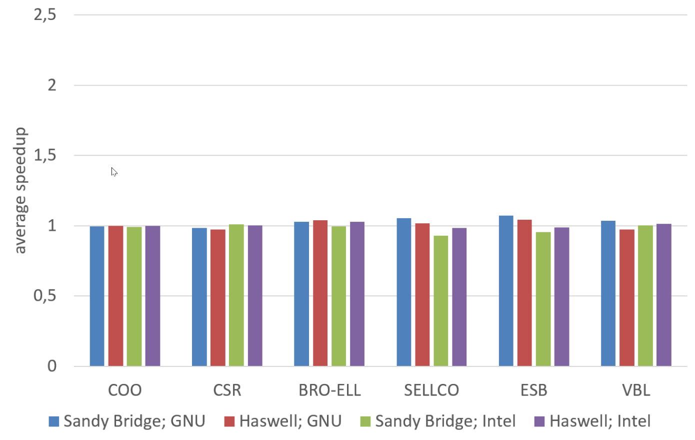

Figure 9. Average speedup with loop unrolling.

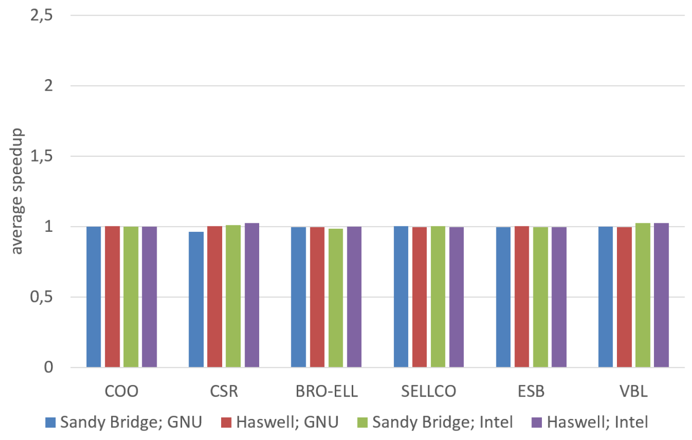

Figure 11. Average speedup with vector directives.

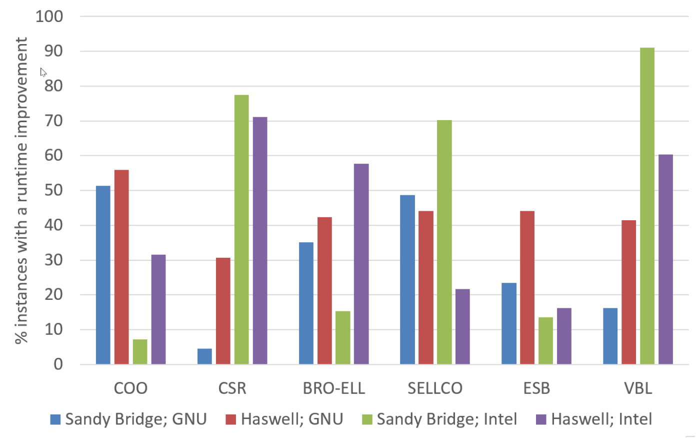

Figure 10. Percentage of Instances that showed a runtime improvement using vector directives.

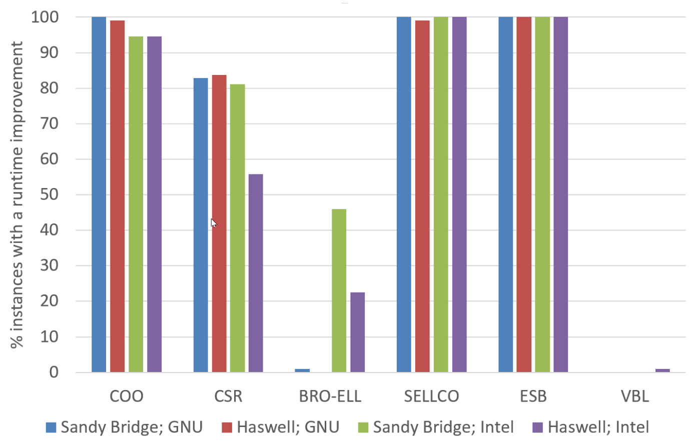

Figure 12. Percentage of Instances that showed a runtime improvement using intrinsics.

unrolling by their own.

C. Using Vector Directives

Vector directives are used to give a compiler additional information and hints. The question was here whether compilers are able to detect themselves opportunities for vectorization by just analyzing the source code or if additional hints are necessary.

As already stated in [1], just specifying a single vector directive is not sufficient to get good performance results. Sometimes this was even counterproductive as a compiler was asked to vectorize but without being able to understand how to do that efficiently. Analyzing all performance information that we gathered we found that several additional information was necessary for the compiler beside the vector directive. Quite important for the Intel compiler was to additionally tell the compiler through additional directives that vector data is aligned at certain byte boundaries (and which has to be assured through appropriate data allocations).

The results were obtained in using OpenMP `simd` directives to explicitly request a vectorization. We found no significant performance difference in using compiler specific vector directives that some compilers know.

Fig. 10 shows the percentage of problem instances that got a performance improvement. Fig. 11 shows the average speedup that was gained in using this directives. Similar to the results for unrolling, there is no real performance gain in using this technique. The compilers are already able to utilize vector units with the modified code, the normal optimization options and by specifying with an additional compiler option for what target architecture code should be generated.

D. Using Intrinsics

While the previous two code optimization techniques are rather high level techniques with few lines of additional code / directives, applying the intrinsic technique is totally different. Here a new program kernel has to be programmed on a rather low abstraction level and explicitly taking a vector length of 4 into account that the hardware units provide. The programming effort is much higher and a deep knowledge of the target processor architecture is essential. For the VBL format, we have not realized a solution with intrinsics and therefore no numbers are shown for that format.

Fig. 12 shows the percentage of problem instances that got a performance improvement. Fig. 13 shows the average speedup that was gained in using this directives, differentiated

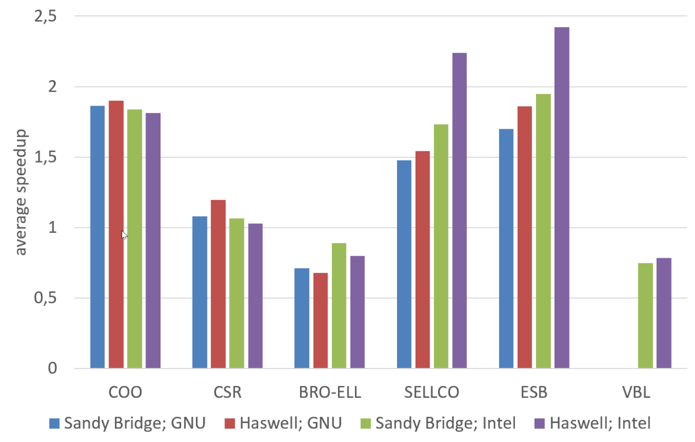

Figure 13. Average speedup with intrinsics.

between the formats. Here, the results are rather different compared to the previous two techniques discussed earlier that have shown no significant performance differences to the compiler's own optimization.

There are three classes of results. In the first class – the formats COO, SELL-C-σ and ESB – a huge performance gain exists when using intrinsics. This gain is up to a factor of nearly 2.5 compared to the pure compiler solution with a high optimization level. Here, an intrinsic programmer was able to find a (much) better performing solution than the compiler. The ESB SpMV intrinsic kernel is rather complex using internally an unrolling technique and utilizing special instructions from the advanced AVX2 instruction set. Also the COO intrinsic kernel is rather complex and uses special AVX2 instructions. Different to that, the SELL-C-σ intrinsic kernel is relative simple using intrinsics only in the innermost loop.

The second class is the format CSR showing no large differences in runtime for the intrinsic solution compared to the compiler generated code.

And in the third class are the formats BRO-ELL and VBL where a performance degradation can be seen for all or nearly all problem instances. Surprisingly (or not?) these are the intrinsic kernels that have the highest complexity / most lines of code (see later Table V for that). Here it may be that the programmer may just be overcharged by the complexity. A compiler applies a strong formal background on code generation and bases its decision on cost estimations. It seems that for very complex code this has advantages.

The performance behavior discussed is mostly invariant of the compiler used (with exceptions for the SELL-C-σ and ESB format).

E. Evaluation Summary

This section summarizes the performance results and relates that to the programming effort that was necessary to reach that. Table V shows the programming effort stated in the number of source lines that was necessary to realize the techniques for the different formats. For the unrolling technique one or a few lines of rather simple additional code was sufficient and no further specific knowledge was required from the programmer. For the vector directive solution, specifying the vector directive alone was not sufficient to get reasonable performance. Additionally, certain additional information was necessary to specify (alignment of vector data). But again, only a few additional lines of high-level code were sufficient and this additional code was not very complex. The programming effort for an intrinsic solution is on the other side very high, very complex and the solution looks rather different to all other solutions for that sparse matrix format. For example, the solution for the VBL format (where many complex code lines were produced but no performance gain was achieved with the intrinsics) has many highly adapted small kernels for a selection of different vertical, horizontal or rectangular block sizes, each programmed in a very different way.

Table IV summarizes the absolute run times for the various compiler levels and manual optimizations on the example matrix, differentiated between three of the six formats used.

The first technique used was compiler flags and the additional information on the target architecture to enable processor specific optimizations in the compiler. The additional effort and needed knowledge for using compiler flags is minimal and no code change is necessary. The results show that already with a small optimization level efficient code is generated for the SpMV operation. Using higher optimization levels did not show significant improvements and sometimes even a small performance degradation. Using just a compiler switch is the preferred method almost always to go for most programmers, unless a really compute intensive kernel should be optimized.

The explicit unrolling technique did not show any major change in runtime, neither in a positive nor in a negative direction. Here, the compilers did already a good job in the field if the programmer just specifies a coarse grain optimization level. The programming effort and necessary expertise to use loop unrolling is quite low.

Using the `simd` compiler directive to ask explicitly for vectorization is easy to use but requires a deeper knowledge, e.g., on data dependencies to avoid wrong code generation as the compiler relies on the given information. As explained above, such a directive alone was not sufficient and additional directives on data alignment must be given. Without the additional alignment information, vectorization was in certain cases prohibited or done in a wrong way, which could result even in a severe performance degradation. The performance results gained with this approach (including directives and alignment specification) for our SpMV kernels are very similar to just using a compiler optimization option. Therefore, the additional programming effort is not really necessary. But for other program codes this may be advantageous where a compiler could *not* handle the code itself without additional information.

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. Additionally, for different processor families and even processors versions different code must be developed, which makes program maintenance hard and costly. But for half of the formats very high performance improvements could be reached with this approach. The SpMV kernels that did not perform as well as the pure compiler generated code were the

TABLE IV. RUNTIMES IN MILLISECONDS FOR VARIOUS COMPILER OPTIMIZATION LEVELS AND OPTIMIZATION TECHNIQUES FOR THE EXAMPLE MATRIX `spe5Ref_a` AND SELECTED FORMATS.

	Intel Sandy Bridge (SB)						Intel Haswell (HW)					
	g++			icpc			g++			icpc		
	CSR	BRO-ELL	SELL-C-σ	CSR	BRO-ELL	SELL-C-σ	CSR	BRO-ELL	SELL-C-σ	CSR	BRO-ELL	SELL-C-σ
-O0	213	737	372	223	769	388	190	603	337	193	624	357
-O1	63	171	121	58	208	118	55	115	107	53	132	105
-O2	57	156	118	53	190	126	61	106	106	53	126	139
-O3	57	156	118	52	191	126	60	105	105	52	126	139
-(O)fast	57	156	118	53	192	125	77	106	129	53	126	138
unrolling	56	152	66	52	187	82	63	101	77	52	118	127
directives	59	156	74	184	188	81	61	106	83	178	126	128
intrinsics	50	190	49	50	171	48	51	134	51	52	129	51

TABLE V. NUMBER OF CODE LINES TO REALIZE THE SPMV (INCLUDING CODE FOR A DIRECT VECTOR ACCESS AS EXPLAINED ABOVE).

	COO	CSR	BRO-ELL	SELL-C-σ	VBL	ESB
normal	5	9	45	16	23	37
unrolling	6	12	46	17	25	39
directives	12	16	47	17	31	42
intrinsics	76	68	303	72	738	170

most complex ones where the assumption is that the program complexity was too high for a human programmer (at least in a fixed amount of available time).

VI. Conclusions

SpMV is a time critical operation in many applications. Optimizing this operation is a challenge. One way to optimize the execution of a SpMV operation is to use the right storage format, mostly dependent on the non-zero structure of the matrix and the target architecture. But additionally to the format, there are various opportunities to tackle that problem on a program optimization level, then partially dependent 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. This was done using implementations of six different storage formats for sparse matrix and related SpMV implementations.

It was shown that using a simple compiler switch turning on a certain optimization level in a compiler results in a good performance for the SpMV operation with minimal/no effort. Already the first level of optimization was sufficient to reach that performance. This could only be achieved *after* some code changes were done that prevented otherwise optimizations. An example was the overloaded array access operator where the performance of the Intel compiler generated code dropped to one halve, even with a high optimization level. Replacing the use of that operator with a direct access to an array structure inside a class increased the performance to a normal level in compensation to the fact that an important software abstraction was lost.

The explicit optimization techniques loop unrolling (with alignment specification) and vector directives have shown no performance differences to a pure compiler optimization. The programming effort for these techniques is rather low, the required expertise is very low for loop unrolling and higher when using vector directives as for a wrong specification a compiler may generate code that produces wrong results.

The fourth approach was using intrinsics on an assembler level. The performance results were ambiguous. Three of the six SpMV implementations got a huge performance boost of up to a factor of 2.5 in average compared to the pure compiler solution with a high optimization level. For one format there was no significant difference in performace. And for the two remaining formats there was a performance degradation. The latter formats have the most complex intrinsic realizations. The programming effort and required expertise level for intrinsics is by far the highest. The number of code lines necessary for an intrinsic realization was often near a factor of 10 compared to a native implementation, sometimes even more. And an intrinsic implementation needs to be enhanced or even completely rewritten for a new processor architecture.

In summary, most programmers should rely on the optimizations done by a compiler. Only for rare cases of very compute intensive program kernels like SpMV there should be thought about an intrinsic solution. As long as the complexity of this code does not get too high, a human programmer in a fixed amount of available development time can speed up such computations substantially, which can justify the effort.

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.

References

[1] R. Berrendorf, M. Weierstall, and F. Mannuss, “Program optimization strategies to improve the performance of SpMV-operations,” in Proc. 8th Intl. Conference on Future Computational Technologies and Applications (FUTURE COMPUTING 2016), 2016, pp. 34–40.

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

[3] A. Buluc, S. Williams, L. Oliker, and J. Demmel, “Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication,” in Proc. Intl. Parallel and Distributed Processing Symposium (IPDPS’2011). IEEE, 2011, pp. 721–733.

[4] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, “Optimization of sparse matrix-vector multiplication on emerging multicore platforms,” in Proc. ACM/IEEE Supercomputing 2007 (SC’07). IEEE, 2007, pp. 1–12.

[5] 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.

[6] 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.

[7] 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.

[8] W. Liu and B. Vinter, "Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors," Parallel Computing, vol. 49, 2015, pp. 179–193.

[9] J. Wong, E. Kuhl, and E. Darve, "A new sparse matrix vector multiplication GPU algorithm designed for finite element problems," arXiv.org, vol. abs/1501.00324, 2015, pp. 1–35.

[10] J. Razzaq, R. Berrendorf, S. Hack, M. Weierstall, and F. Mannuss, "Fixed and variable sized block techniques for sparse matrix vector multiplication with general matrix structures," in Proc. Tenth Intl. Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP 2016), 2016, to appear.

[11] J. Razzaq, R. Berrendorf, J. P. Ecker, S. E. Scholl, and F. Mannuss, "Performance characterization of current CPUs and accelerators using micro-benchmarks," Intl. Journal on Advances in Systems and Measurements, vol. 9, no. 1&2, 2016, pp. 77–90.

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

[13] D. A. Patterson and J. L. Hennessy, Computer Organization and Design – The Hardware Software Interface, 5th ed. Morgan Kaufmann, 2014.

[14] Intel C++ Compiler 16.0 User and Reference Guide, https://software.intel.com/en-us/intel-cplusplus-compiler-16.0-user-and-reference-guide ed., Intel Corporation, 2016, retrieved: August 2016.

[15] L. Project, The LLVM Compiler Infrastructure, http://llvm.org/, 2014, retrieved: August 2016.

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

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

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

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

[20] OpenMP Application Program Interface, 4th ed., OpenMP Architecture Review Board, http://www.openmp.org/, Jul. 2015, retrieved: August 2016.

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

[22] 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.

[23] 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.

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

[25] GCC, the GNU Compiler Collection, Free Software Foundation, retrieved: August 2016. [Online]. Available: https://gcc.gnu.org/

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

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

[28] Intel® 64 and IA-32 Architectures Optimization Reference Manual, Intel, http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, Jun. 2016, retrieved: August 2016.

[29] 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, Jun. 2016, retrieved: August 2016.

[30] Software Optimization Guide for AMD Family 15h Processors, AMD, http://support.amd.com/TechDocs/47414_15h_sw_opt_guide.pdf, 2042, retrieved: July 2016.

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

[32] G. Chrysos, Intel® Xeon Phi^{TM} Coprocessor – The Architecture, https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner, 2012, retrieved: November 2014.

[33] 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.

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

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

[36] 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.

[37] CSR5 reference implementation, https://github.com/bhSPARSE/Benchmark_SpMV_using_CSR5.

[38] A. Buluc, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in Proc. 21th Annual Symp. on Parallelism in Algorithms and Architectures (SPAA'09), 2009, pp. 233–244.

[39] CSB reference implementation, http://gauss.cs.ucsb.edu/~aydin/csb/html/index.html.

[40] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaSpMV: yet another SpMV framework on GPUs," in Proc. 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14), 2014, pp. 107–118.

[41] pOSKI: parallel Optimized Sparse Kernel Interface, http://bebop.cs.berkeley.edu/poski/, retrieved: August 2016.

[42] V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, and N. Koziris, "An extended compression format for the optimization of sparse matrix-vector multiplication," IEEE Transactions on Parallel and Distributed Systems, vol. 24, no. 10, Oct. 2013, pp. 1930–1940.

[43] C. Lehnert, R. Berrendorf, J. P. Ecker, and F. Mannuss, "Performance prediction and ranking of spmv kernels on gpu architectures," in Proc. 22th Intl. European Conference on Parallel and Distributed Computing (Euro-Par 2016), 2016, p. to appear.

[44] N. Sedaghati, A. Ashari, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Characterizing dataset dependence for sparse matrix-vector multiplication on GPUs," in Proc. 2nd Workshop on Parallel Programming for Analytics Applications (PPAA'15). ACM, 2015, pp. 17–24.

[45] N. Sedaghati, T. Mu, L.-N. Pouchet, , S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on GPUs," in Proc. 25th Intl. Conference on Supercomputing (ICS 2015). ACM, 2015.

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

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

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

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

[50] J. Jeffers, J. Reinders, and A. Sodani, Intel Xeon Phi Processor High Performance Programming, Knights Landing Edition. Cambridge: Morgan Kaufman Publishers, Inc., 2016.

[51] ISA Extensions, Intel, https://software.intel.com/en-us/isa-extensions/intel-avx, retrieved: May 2016.

[52] J. Ecker, R. Berrendorf, J. Razzaq, S. E. Scholl, and F. Mannuss, "Comparing different programming approaches for SpMV-operations on GPUs," in Proc. 11th International Conference on Parallel Processing and Applied Mathematics (PPAM 2015), vol. 9573. Springer International Publishing, 2016, pp. 537–547.

[53] W. Tang, W. Tan, R. Ray, Y. Wong, W. Chen, S. Kuo, R. Goh, S. Turner, and W. Wong, "Accelerating sparse matrix-vector multiplication on GPUs using bit-representation-optimized schemes," in Proc. Intl. Conference on High Performance Computing, Networking, Storage and Analysis (SC'13). ACM, 2013, article no. 26.

[54] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," in Proc. ACM/IEEE Conference on Supercomputing (SC'99). IEEE, Nov. 1999.

[55] X. Liu, E. Chow, M. Smelyanskiy, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-code processors," in Proc. Intl. Conference on Supercomputing (ICS'13). ACM, 2013, pp. 273–282.

[56] 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.

[57] SPE Comparative Solution Project, Society of Petroleum Engineers, http://www.spe.org/web/csp/, retrieved: February, 2016.