Received: 28 November 2017 | Revised: 12 April 2018 | Accepted: 4 May 2018
DOI: 10.1002/cpe.4701

WILEY

SPECIAL ISSUE PAPER

A sparse matrix-vector multiplication method with low preprocessing cost

Barış Aktemur

Department of Computer Science, Özyeğin University, Istanbul, Turkey

Correspondence
Barış Aktemur, Department of Computer Science, Özyeğin University, Cekmekoy, 34794 Istanbul, Turkey.
Email: baris.aktemur@ozyegin.edu.tr

Summary

Sparse matrix-vector multiplication (SpMV) is a crucial operation used for solving many engineering and scientific problems. In general, there is no single SpMV method that gives high performance for all sparse matrices. Even though there exist sparse matrix storage formats and SpMV implementations that yield high efficiency for certain matrix structures, using these methods may entail high preprocessing or format conversion costs. In this work, we present a new SpMV implementation, named CSRLenGoto, that can be utilized by preprocessing the Compressed Sparse Row (CSR) format of a matrix. This preprocessing phase is inexpensive enough for the associated cost to be compensated in just a few repetitions of the SpMV operation. CSRLenGoto is based on complete loop unrolling and gives performance improvements in particular for matrices whose mean row length is low. We parallelized our method by integrating it into a state-of-the-art matrix partitioning approach as the kernel operation. We observed up to 2.46× and on the average 1.29× speedup with respect to Intel MKL's SpMV function for matrices with short- or medium-length rows.

KEYWORDS

compressed sparse row, sparse matrix-vector multiplication, SpMV

1 | INTRODUCTION

Sparse matrices are those matrices that contain a high ratio of zeros. Sparse matrix-vector multiplication (SpMV) is a fundamental operation used frequently in the solutions to many engineering and science problems. In domains such as Krylov subspace problems and iterative solvers, a sparse matrix goes into an SpMV operation tens or hundreds of times. For this reason, SpMV plays a key role in high performance computing. However, it is known that SpMV's performance falls well behind the capacity of modern computers.[1] Hence, optimization of SpMV has been extensively studied (see the works of Langr and Tvrdik,[2] and Filippone et al[3] for comprehensive surveys).

Sparse matrices are stored in formats that provide space savings. Probably, the most popular and the *de facto* standard of these formats is the *Compressed Sparse Row* (CSR) representation[4] (explained in detail in the next section). CSR is a general-purpose sparse matrix storage format, taking no parameters such as a block size or the cache line length, and oblivious to architectural details of the target computer. However, it cannot give the best performance for all sparse matrix types. For this reason, new storage formats are developed to address performance shortcomings,[2] sometimes targetting only a limited class of sparse matrices. When a new format is proposed, there is a crucial cost that needs to be taken into account: the cost of converting the matrix data from CSR format to the new format. If this conversion is too costly with respect to the SpMV operation, the format would find a very limited use in practice. Furthermore, in addition to the time overhead of format conversion, one has to consider space overheads too because if the matrix is going to be used in other operations after SpMV, the CSR representation cannot be thrown away; the matrix data has to be kept in both the CSR format and the new format. In other words, there may be a need to keep two copies of a matrix in memory. For these reasons, there has been a recent interest to use the CSR format directly or after an inexpensive preprocessing step.[5-10]

Abbreviations: CSR, Compressed Sparse Row; MKL, Intel's Math Kernel Library; SpMV, sparse matrix-vector multiplication.

In this work, we investigate improving the efficiency of SpMV based on complete loop unrolling and using a variation of CSR as the storage format. The experimental evaluation of our approach shows that significant speedups can be obtained, particularly for matrices that have short- and medium-length rows, by paying a low preprocessing cost. The **contributions** made by our work are as follows:

- We present a variation of the CSR matrix storage format, named *CSRLen*.
- We present an SpMV implementation, named *CSRLenGoto*, using the CSRLen format and based on complete loop unrolling. *CSRLenGoto* gives substantial speedups with respect to CSR. Our current implementation is for the X64_64 CPU architecture.
- Matrix data in CSR format can be converted to the *CSRLen* format very quickly (on average, in duration equivalent to 0.11-0.25× of the baseline SpMV execution). This way, the preprocessing cost can be amortized in only a few iterations of the SpMV for most of the matrices.
- *CSRLenGoto* gives speedup for matrices that have low or medium mean row length and in particular for matrices with short rows (eg, mean row length smaller than 8).
- Our method can be parallelized straightforwardly using existing matrix partitioning methods. As an example, we integrated our method as the kernel method of the merge-based SpMV approach[5] and obtained up to 2.46× performance with respect to Intel's Math Kernel Library (MKL) on an 8-core Intel Xeon CPU.
- Our code is publicly available at https://github.com/aktemur/CSRLenGoto.

This paper is organized as follows. Section 2 gives background information about the SpMV problem. Section 3 presents our proposed approach. Sections 4 and 5 evaluate the performance in single-threaded and multi-threaded settings, respectively. Section 6 compares and contrasts our approach to the existing work. Finally, Section 7 gives our conclusions.

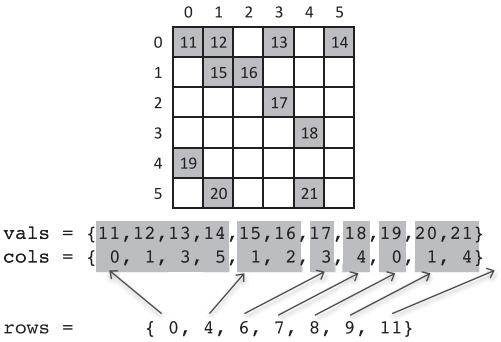
2 | BACKGROUND

In the CSR (*Compressed Sparse Row*) format, a matrix is represented by three arrays that we will call `vals`, `cols`, and `rows`. In the `vals` array, the nonzero elements of the matrix are stored according to the row-major ordering. The `cols` array stores the column indices of the nonzero elements, in the same order as the `vals` array. The `rows` array stores, for each row, the index of the first element of the row in the other two arrays. A sample matrix, its CSR representation, and the SpMV code for the CSR format are given in Figure 1. In the code, `N` is the number of rows of the matrix; `v` is the input vector; `w` is the output vector. SpMV calculates the following expression: $w \leftarrow w + M \cdot v$.

SpMV CSR code is notorious for having poor performance. There are several reasons behind this. SpMV is a memory-bound computation.[11] The accesses to the input vector `v` are indirect and irregular; this causes underutilization of the cache and poor instruction-level parallelism.[1,12] Because sparse matrices usually have short rows, the trip count of the inner loop is low. For this reason, loop-related costs stay relatively high, and the branch-predictor of the CPU does not help much.[13] As an attempt to remedy these performance problems, we can try **loop unrolling**—one of the standard optimization transformations performed by compilers. For instance, if we unroll the inner loop in Figure 1 for 4 times, we obtain the code in Figure 2. Here, we need a second inner loop to handle the left-over elements in case the number of elements in the row is not an exact multiple of the unrolling factor 4. We will refer to the code obtained by unrolling the inner loop of the CSR code *k* times as CSR_k. So, the original code given in Figure 1 is CSR_1. The necessity for a second inner loop in CSR_k brings runtime overhead especially for short rows seen frequently in sparse matrices, and CSR_k does not yield the desired performance increase.

3 | OUR PROPOSED APPROACH

In this section, we present *CSRLenGoto*—the new SpMV method we are proposing; but before we do that, we will discuss an intermediary method that we name *CSRGoto*.




```
for (int i = 0; i < N; i++) {
  double sum = 0.0;
  for (int j = rows[i]; j < rows[i+1]; j++)
    sum += vals[j] * v[cols[j]];
  w[i] += sum;
}
```

FIGURE 1 A sample matrix, its CSR representation, and the SpMV function for the CSR format

```
for (int i = 0; i < N; i++) {
  double sum = 0.0;
  int j = rows[i];
  for (; j < rows[i+1] - 3; j += 4) {
    sum += vals[j] * v[cols[j]];
    sum += vals[j+1] * v[cols[j+1]];
    sum += vals[j+2] * v[cols[j+2]];
    sum += vals[j+3] * v[cols[j+3]];
  }
  for (; j < rows[i+1]; j++)
    sum += vals[j] * v[cols[j]];
  w[i] += sum;
}
```

FIGURE 2 CSR_4 code obtained by unrolling the inner loop of Figure 1 by a factor of 4

```
      // Expected matrix format: CSR
      int j = 0, i = 0, length;
      double sum;
      goto init;
 L_5: sum += vals[j] * v[cols[j]]; j++;
 L_4: sum += vals[j] * v[cols[j]]; j++;
 L_3: sum += vals[j] * v[cols[j]]; j++;
 L_2: sum += vals[j] * v[cols[j]]; j++;
 L_1: sum += vals[j] * v[cols[j]]; j++;
 L_0: w[i] += sum;
      i++;
init: if (i >= N) goto end;
      length = rows[i + 1] - rows[i];
      sum = 0.0;
      goto L_length; // conceptual
 end: ;
```

FIGURE 3 *CSRGoto* code when the maximum row length of the matrix is 5

3.1 | *CSRGoto*

Let us investigate unrolling the inner loop of CSR_1 completely instead of k times, so that we can get rid of the need to have a second inner loop to handle the left-over elements. Because the maximum iteration count of the inner loop of CSR_1 will be equal to the maximum row length of the matrix, we just need to unroll the loop as many times as the number of elements in the longest row. After unrolling, for each row, we have to be able to jump to the appropriate point in the code according to the length of the row. We can do that by putting labels in the code and using a `goto` statement. Let us call this SpMV method *CSRGoto*. We show a *conceptual* code for this method in Figure 3. Note that for this method to work correctly, it is not required that the rows of the matrix have a fixed length; row lengths may vary, but the original inner loop should be unrolled for at least the maximum row length of the matrix.

We said the code in Figure 3 is "conceptual" because the last line is not a legal C statement. Here, we have a programming challenge. How can we express in C the address that we want to jump to? We can use the label addressing operator (`&&`) and the *computed goto* statement* that do not exist in the C standard but are supported by compilers like GCC and Clang:

```
long delta = (&&L_0 - &&L_5) / 5; //distance between each consecutive label
goto *(void*)(&&L_0 - length * delta);
```

However, for this code to be correct, we have to trust the compiler to position the labels equidistantly. In our experiments, we have seen that this is not a valid assumption. Therefore, we decided to implement the SpMV function at the assembly code level, instead of source code. To determine which machine instructions to use, we compiled source codes similar to the one given in Figure 3 using the Clang, GCC, and icc compilers to produce X86_64 assembly code (with the `-O3` optimization level). Upon manual examination of the output of the compilers, we have seen that the statement

```
sum += vals[j] * v[cols[j]]; j++;
```

is transformed to native code similar to

```
movslq (%r9,%rax,4), %rbx   ; rbx ← cols[j]          (4 bytes)
movsd (%r8,%rax,8), %xmm1   ; xmm1 ← vals[j]         (6 bytes)
incq %rax                   ; j++                    (3 bytes)
mulsd (%rdi,%rbx,8), %xmm1  ; xmm1 ← xmm1 * v[rbx]   (5 bytes)
addsd %xmm1, %xmm0          ; sum ← sum + xmm1       (4 bytes).
```

*https://gcc.gnu.org/onlinedocs/gcc-7.1.0/gcc/Labels-as-Values.html

```
1       xor %eax, %eax              ; j ← 0
2       xor %edx, %edx              ; i ← 0
3       movsxd (%r11), %rcx         ; rcx ← rows[0]
4       jmp init
5   L_5: ; sum += vals[j] * v[cols[j]]; j++;
6       movslq (%r9,%rax,4), %rbx
7       movsd (%r8,%rax,8), %xmm1
8       incq %rax
9       mulsd (%rdi,%rbx,8), %xmm1
10      addsd %xmm1, %xmm0
11  L_4: ... ; same as L_5
12  L_3: ... ; same as L_5
13  L_2: ... ; same as L_5
14  L_1: ... ; same as L_5
15  L_0: addsd (%rsi,%rdx,8), %xmm0 ; sum ← sum + w[i]
16      movsd %xmm0, (%rsi,%rdx,8) ; w[i] ← sum
17      incq %rdx                   ; i ← i + 1
18  init:
19      cmp %rdx, N                 ; exit if i ≥ N
20      jge end
21      movslq 4(%r11,%rdx,4),%rbx  ; rbx ← rows[i+1]
22      subq %rbx, %rcx             ; rcx ← -length
23      imul 22, %rcx               ; rcx ← -length * 22
24      leaq -45(%rip), %r10        ; r10 ← L_0
25      addq %rcx, %r10             ; r10 ← L_0 + rcx
26      leaq (%rbx), %rcx           ; rcx ← rbx
27      xorps %xmm0, %xmm0          ; sum ← 0
28      jmp *%r10                   ; goto L_length
29  end:
```

FIGURE 4 X86_64 assembly code for *CSRGoto* when the maximum row length in the matrix is 5

Based on this observation, we write for *CSRGoto* the assembly code shown in Figure 4. A straightforward manual examination of the instructions above reveals that their total length is 22 bytes (4+6+3+5+4). Therefore, the distance between two consecutive labels `L_` j and `L_`$(j + 1)$ will be 22 bytes. Because this is a constant value, no runtime calculation is needed. We use this fact on line 23 in Figure 4.

3.2 | CSRLenGoto

In the *CSRGoto* method, substantial amount of work is performed to calculate the address to jump to (Figure 4, between the `init` and `end` labels). This calculation may be a burden especially for short rows. To remedy, we can pre-compute the work that does not depend on the program counter `%rip`, such as the calculation of `length` values, and keep those values as part of the matrix data. For this, we need to go over the `rows` array of the matrix in a preprocessing phase. We name the matrix data obtained in this way, ie, *CSRLen*, and the SpMV method using this matrix data, ie, *CSRLenGoto*. Converting the data from the CSR format to CSRLen involves processing the `rows` array only and is given in Figure 5. Here, we calculate for each row the distance to land on the label appropriate for the row length. Now that we are preparing a new `rows` array, we can as well get rid of the check for the exit condition (the `cmp` and `jge` instructions). For this, we make an addendum to the very end of the new `rows` array and store the distance between the labels `L_0` and `end`, which is 33 bytes. This value is the sum of the lengths of instructions that appear between these two labels (individual instruction lengths are shown in Figure 6).

In Figure 5, we create a new array named `newRows`. If the user does not need to keep around the original CSR data, the original `rows` array can be overwritten instead of creating a separate array. In our experiments, to be fair in the measurements, we created a new array. The runtime complexity of the conversion is $O(\mathtt{N})$; the `vals` and `cols` arrays are not modified or processed. Because the size of the `newRows` array is the same as the original one, the size of the matrix data transferred from the memory to the SpMV function will be the same as CSR.

```
int *newRows = new int[N + 1];
for (int i = 0; i < N; i++) {
  int length = rows[i + 1] - rows[i];
  newRows[i] = -length * 22;
}
newRows[N] = 33; // distance between 'L_0' and 'end' in Figure 6
```

FIGURE 5 Converting the CSR format to CSRLen

```
     xor %eax, %eax              ; j ← 0
     xor %edx, %edx              ; i ← 0
     jmp init
L_5: ; sum += vals[j] * v[cols[j]]; j++;
     movslq (%r9,%rax,4), %rbx
     movsd (%r8,%rax,8), %xmm1
     incq %rax
     mulsd (%rdi,%rbx,8), %xmm1
     addsd %xmm1, %xmm0
L_4: ... ; same as L_5
L_3: ... ; same as L_5
L_2: ... ; same as L_5
L_1: ... ; same as L_5
L_0: addsd (%rsi,%rdx,8), %xmm0 ; sum ← sum + w[i]  (5 bytes)
     movsd %xmm0, (%rsi,%rdx,8) ; w[i] ← sum        (5 bytes)
     incq %rdx                  ; i ← i + 1         (3 bytes)
init:
     xorps %xmm0, %xmm0         ; sum ← 0           (3 bytes)
     movslq (%r11,%rdx,4), %rbx ; rbx ← rows[i]     (4 bytes)
     leaq -27(%rip), %r10       ; r10 ← L_0         (7 bytes)
     addq %rbx, %r10            ; r10 ← r10 + rbx   (3 bytes)
     jmp *%r10                  ; goto L_length     (3 bytes)
end:
```

FIGURE 6 X86_64 assembly code for *CSRLenGoto* when the maximum row length in the matrix is 5

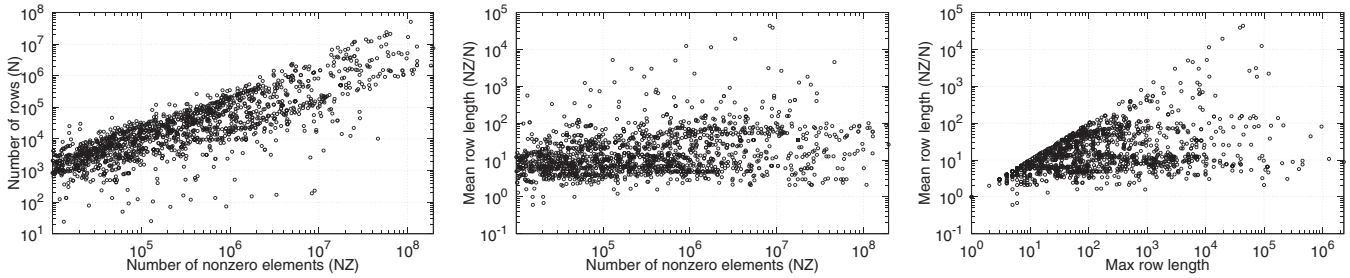


FIGURE 7 Information about the 1878 matrices in our data set that we used for performance evaluation

CSRLenGoto code is given in Figure 6. This code is quite similar to the *CSRGoto* code in Figure 4. The only major difference is that the distance to jump is read from the `rows` array instead of being computed on the fly according to the row length. Remember that the source-level unrolling code in Figure 2 contains one outer loop and two inner loops. The associated overheads prevent performance benefits, especially for short rows. The overheads are minimized in our proposed approach where there exists only one loop obtained via a jump instruction, whose target is dynamically computed for each row. Runtime computation is further eliminated through the preprocessing phase. When code is unrolled at the source-level, the compiler is provided with opportunities for reordering the instructions. We lose this feature because we implemented our approach in assembly; however, that is a compromise we had to make to guarantee that the code sections are equally spaced.

4 | SINGLE-THREADED PERFORMANCE EVALUATION

In this section, we measure and discuss the performance of *CSRLenGoto* in a single-threaded execution environment.

4.1 | Setup

We prepared a data set comprising real-world matrices obtained from the SuiteSparse (formerly known as the University of Florida) matrix collection.[14] Our set contains non-complex-valued matrices that have 10K-200M nonzero elements in the SuiteSparse collection. There are 1878 such matrices. Information about the number of rows, nonzero elements, and row lengths of this set are given in Figure 7.

We ran the tests on two X86_64 computers, one with an Intel CPU and the other with an AMD CPU. The properties of the machines are given in Table 1. The codes were compiled by passing the `-O3` flag[†] to the compiler. Our testbeds were kept unloaded during benchmarking to minimize interference with other processes. When measuring the SpMV duration, we repeatedly invoked the function in a loop and recorded the total elapsed

[†]Vectorization is enabled by default with this flag; however, SpMV implementation for CSR does not benefit from vectorization because of the indirect access to the input vector `v`.

TABLE 1 Machines used in our experiments

CPU (*Micro-Architecture*)	Cache Size (Bytes)			Memory	Compiler
	L1 (I/D)	L2	L3	(GB)	
Intel Xeon E5-1660v4 @ 3.20 Ghz, 8-core, 16 threads (*Broadwell*)	8×(32K/32K)	8×256 K	20 M	32	icc 17.0.0
AMD FX 8350 @ 4.00 Ghz, 8-core, 8 threads (*Piledriver*)	4×64K/8×16 K	4×2 M	8 M	8	gcc 5.4.0

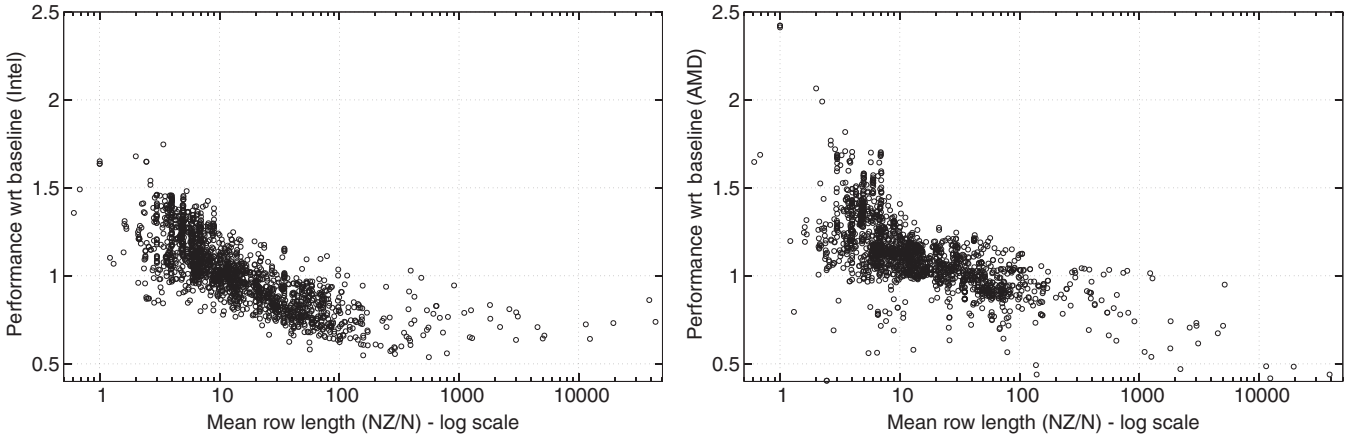


FIGURE 8 *CSRLenGoto*'s performance with respect to the baseline method in the single-threaded setting on Intel (left) and AMD (right)

time to avoid possible noise. We set the number of repetitions to a sufficiently large value to keep the total elapsed time reasonably long (eg, $\sim$ 1 second or longer). We divided the elapsed time to the number of repetitions to find the duration taken by a single SpMV invocation. This way, we measured the SpMV time for each matrix for 3 times and recorded the smallest (ie, fastest execution). We used the double-precision floating point type (64-bit `double`) for the nonzero values and integer type (32-bit `int`) for row and column indices. For the CSR_k method, we used the following values for k: $\{1, 4, 8, 16, 32\}$. For *CSRLenGoto*, in addition to the SpMV duration, we also measured the preprocessing time where CSR format is converted to CSRLen. We do not include the cost of preprocessing as part of SpMV time because in a typical "inspector-executor" setting, first, an inspection phase is performed for optimizations, followed by the execution phase where the optimized SpMV function is executed many times in an iteration. Intel's Math Kernel Library (MKL) provides an inspector-executor API[‡] as well. We compare our library against MKL in our multi-threaded performance evaluation (Section 5). We do not evaluate against MKL in this section because MKL is tuned for parallelism and does not perform well in sequential execution.

On each testbed machine, we set the best-performing CSR_k as the baseline method. We determined the best-performing CSR_k as follows: For every matrix, we normalized each CSR_k result with respect to the fastest of the CSR_k methods for that matrix. We observed that on Intel, CSR_1 is almost always the best CSR_k method. Hence, we picked CSR_1 as the baseline on Intel. On AMD, CSR_1 was 8%, CSR_4 was 4%, CSR_8 was 5%, CSR_{16} was 8%, and CSR_{32} was 13% worse than the fastest CSR_k when averaged over all the matrices. Hence, we set CSR_4 as the baseline method on AMD.

We express the performance of the *CSRLenGoto* method with respect to the performance of the baseline method. For this, we divide the duration taken by the baseline method to the time taken by *CSRLenGoto*. Having a ratio smaller than 1 means that *CSRLenGoto* caused a slowdown; a ratio larger than 1 means that we obtained speedup.

4.2 | Results

When examining the results, we noticed that the performance of *CSRLenGoto* is related to the mean row lengths of the matrices. Therefore, we are presenting the performance results by associating them with mean row lengths. Figure 8 gives the results. We can see on both machines that as the mean row length increases, the performance drops down. This observation suggests that we can consider the matrices to be in one of three categories based on their mean row length and the general trend in performance: For matrices with *short rows*, substantial speedups are obtained for the majority. The performance for the matrices with *medium-length rows* is mixed, with results both below and above the baseline. Finally, hardly any speedup is achieved for matrices with *long rows*. This observation naturally raises the next question: Which row length values should we use to draw the boundaries between these three categories? We will soon discuss *where* exactly we can cut the data set, but let us first discuss *why* such a categorization helps.

An SpMV library needs to make runtime decisions about which SpMV method to use among possibly many options for a particular matrix. If the matrix can be put into a category for which a particular SpMV method is known to consistently give good performance, the library can pick that method and proceed with the SpMV operation. The categorization of the matrix is to be made based on install-time information learned on

[‡]https://software.intel.com/en-us/mkl-developer-reference-c-inspector-executor-sparse-blas-routines

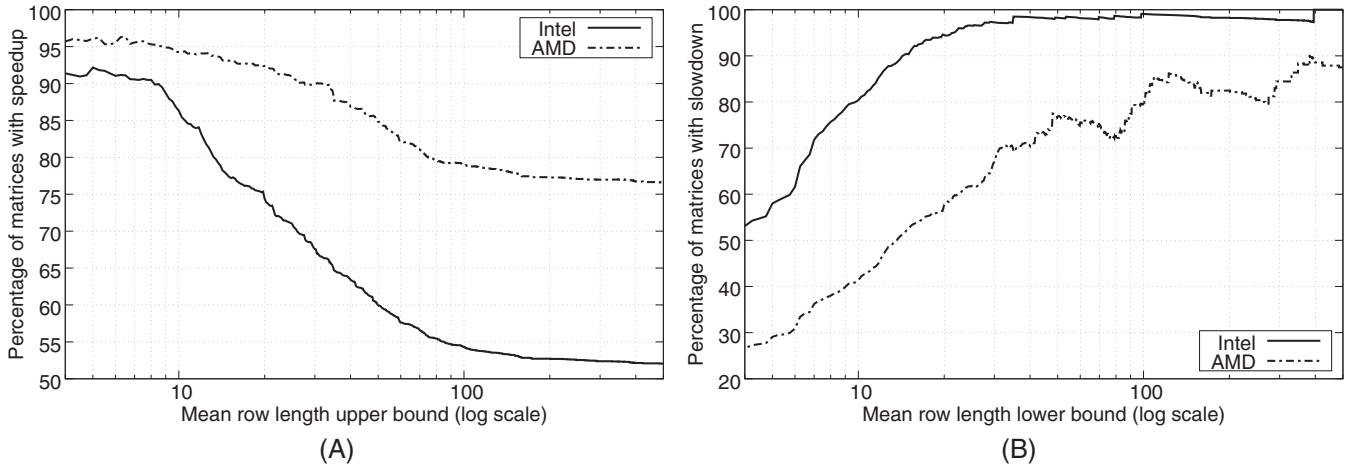


FIGURE 9 Percentage of matrices that give speedup when a certain mean row length value is used as (A) an upper bound to define "matrices with short rows" and (B) a lower bound to define "matrices with long rows"

the target machine and a dynamic analysis of the matrix. This is a typical *auto-tuning* approach that has been successfully applied in the case of SpMV.[15-20] Ideally, the dynamic analysis of the matrix should be as inexpensive as possible. In the case of *CSRLenGoto*, one can quickly check whether the given matrix falls into the category of "short rows" or "long rows". This is just a matter of calculating the mean row length of the matrix via a single division operation (ie, NZ/N) and comparison. If the mean row length is low, the library can decide with high confidence that *CSRLenGoto* will yield speedups and perform the SpMV operation using *CSRLenGoto*. Similarly, if the mean row length is high, the library can predict that *CSRLenGoto* is unlikely to perform better than the baseline and do not attempt using it. When the mean row length is in between, the library may prefer to perform further analysis based on matrix features other than NZ/N (eg, max row length and variation of row lengths) or monitor the first few executions of *CSRLenGoto* to decide whether to fall back to the default method or continue using it.

The next question is what threshold values should we use for the mean row length so a matrix can be categorized as having short/medium/long rows? Figure 8 suggests that these ranges should be machine-specific. Figure 9A depicts an analysis of what percentage of matrices give speedup when a certain mean row length value is used as an upper bound to define "matrices with short rows". For instance, 90% of the matrices whose mean row length is 8.0 or less (767 matrices) on Intel and 27.5 or less (1360 matrices) on AMD give speedup when using *CSRLenGoto*. Similarly, Figure 9B gives an analysis of what percentage of matrices give *slowdown* when a certain mean row length value is used as a *lower* bound to define "matrices with long rows". For instance, among the matrices whose mean row length is 14.5 or more (738 matrices), 90% give slowdown on Intel. So, we can say with high confidence that *CSRLenGoto* will not yield any benefits if the mean row length of a matrix is bigger than 14.5. The situation is better on AMD; for example, among the matrices whose mean row length is 40.0 or more (372 matrices), 70% give slowdown. If we use these boundaries to define the range for matrices with medium-length rows (ie, 8.0-14.5 on Intel, 27.5-40.0 on AMD), then the percentage of matrices that show speedup is 56% on Intel (208 out of 373) and 62% on AMD (90 out of 146). In general, an analysis like this one can be carried on a machine-basis to evaluate the likelihood of *CSRLenGoto* giving speedup for a matrix by just looking at the NZ/N value of the matrix. If an auto-tuning approach is taken, using a decision tree classifier is potentially an effective method for finding the thresholds. In the rest of this paper, we will use the threshold values 8.0 and 14.5 on Intel and 27.5 and 40.0 on AMD, when categorizing matrices as having short/medium/long rows.

It is hard to say precisely why *CSRLenGoto* shows performance relative to the mean row length value, but a reasonable explanation is due to the utilization of the cache. Modern processors include a small cache, called the *micro-op cache*, with the purpose of optimizing the instruction decode time especially for loops with small bodies. For *CSRLenGoto*, as the matrix row length increases, the "loop body" is going to be long, and this diminishes the advantages of the micro-op cache. When the row length is further longer, we begin losing the benefits of the level-1 instruction cache as well. This is why, we believe, *CSRLenGoto* provides better speedup for matrices with shorter rows. The reason we are seeing different speedups on our test machines is likely related to the difference in their cache sizes. The AMD CPU has a larger level-1 instruction cache; hence, better speedups were observed on that CPU for matrices with high row lengths.

4.3 | Preprocessing cost

Recall that to use *CSRLenGoto*, a matrix has to go through a preprocessing phase to convert the matrix data from CSR format to CSRLen. We have measured the cost of conversion relative to the cost of one SpMV function execution using the baseline method. This way, we are attempting to answer the question *"How many SpMV operations could we have executed using the baseline method instead of doing preprocessing for CSRLenGoto?"* The answer to this question is given in the top row in Figure 10. The relative cost of preprocessing tends to be higher for matrices with short rows compared to the matrices with long rows. Yet, the costs are very small, considering that in an expensive SpMV method, preprocessing may take time

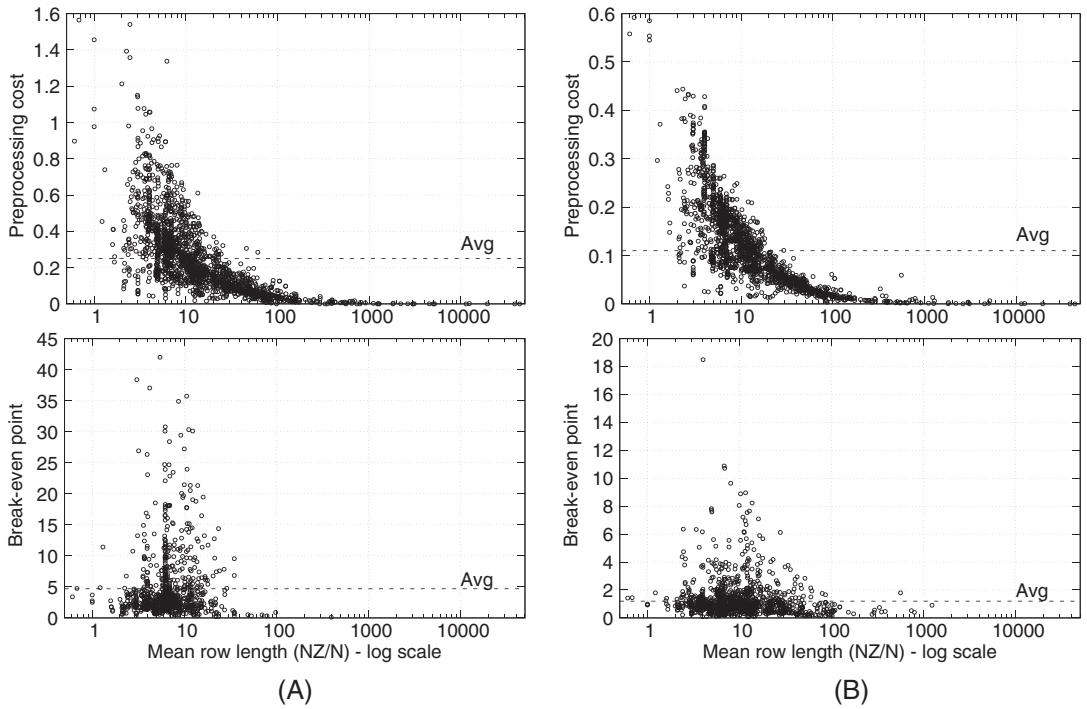


FIGURE 10 Top row: The cost of preprocessing normalized with respect to the cost of one SpMV operation using the baseline method on Intel (left) and AMD (right). Bottom row: Break-even points of matrices for which the performance is at least 1.01× on Intel (left) and AMD (right)

equivalent to tens or hundreds of SpMV invocations.[6,19,21-23] On the average, CSRLen preprocessing time is equivalent to 0.25 SpMV executions on Intel and 0.11 on AMD. The largest cost we have measured is 1.56 on Intel and 0.59 on AMD.

The next question we ask is *"How many times do we have to repeat SpMV for a matrix so that CSRLenGoto compensates its preprocessing cost and becomes quicker than the baseline method?* This question makes sense especially in the context of iterative solvers, where the same matrix is multiplied with some vector many times until a convergence point is reached. In this context, we can afford to pay a preprocessing cost, and if the SpMV operation is repeated for a sufficient number of iterations, we get profits in the overall time spent. The number of iterations needed to start getting profit is called the *break-even point*. The bottom row in Figure 10 shows the break-even points for matrices for which the measured performance is at least 1.01×. There are 961 such matrices on Intel and 1418 on AMD. Most of the break-even points are in the range of just a couple SpMV iterations. The average break-even points are 4.7 and 1.2 for Intel and AMD, respectively; the maximums are 42.0 and 18.5 for Intel and AMD, respectively. This shows that *CSRLenGoto* can be used in iterative solver contexts even when the iteration count is low.

4.4 | Comparison to Yzelman's Sparse Library

Yzelman provides an SpMV benchmark software, called *Sparse Library*, that contains implementation of several SpMV methods.[24] We ran the sequential methods in Sparse Library and measured the performance in order to compare *CSRLenGoto* to a method other than the baseline CSR. We evaluated the following methods: incremental CSR (ICSR); zig-zag CSR (ZZ-CSR); zig-zag incremental CSR (ZZ-ICSR); sparse vector matrix (SVM); Hilbert-ordered triplet schema (HTS); bi-directional incremental CSR (BICSR); hilbert-ordered triplets, backed by BICSR (Hilbert); sparse matrix blocking, backed by Hilbert (Block Hilbert); sparse matrix blocking by bisection, backed by Hilbert (Bisection Hilbert); compressed bi-directional incremental CSR (CBICSR). We again used the 1878-matrix set. We performed the measurements on our Intel machine only.

The results are given in Figure 11, where the table on the left lists the evaluated SpMV methods and the number of matrices for which the method was the best performer among the methods listed in the table. In 847 (45.1%) of the matrices, no method was better than the baseline CSR. In other cases, ZZ-CSR dominates the table with a count of 788 (42.0%) matrices. For this reason, we picked ZZ-CSR as the method to compare against *CSRLenGoto*. The result is given in the right-hand side in Figure 11. Here, we see that *CSRLenGoto* again performs very well for matrices with short rows. It should be noted that the zig-zag CSR method needs to reorder the nonzero values and the column indices of the CSR format in its preprocessing phase; *CSRLenGoto*, however, uses these two arrays verbatim.

5 | MULTI-THREADED PERFORMANCE EVALUATION

In this section, we evaluate *CSRLenGoto*'s performance in a multi-threaded environment. SpMV is a highly parallelizable computation; matrix rows can be processed concurrently without having to do synchronization among rows. A typical parallelization approach is to partition the matrix into

Method	Wins	Method	Wins
ICSR	44 (2.3%)	BICSR	9 (0.5%)
ZZ-CSR	788 (42.0%)	Hilbert	48 (2.6%)
ZZ-ICSR	39 (2.1%)	Block Hilbert	1 (0.1%)
SVM	0 (0.0%)	Bisection Hilbert	39 (2.1%)
HTS	51 (2.7%)	CBICSR	12 (0.6%)
Baseline CSR	847 (45.1%)		

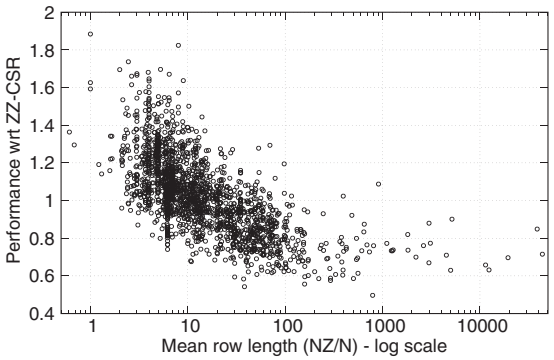


FIGURE 11 The SpMV methods we evaluated from Yzelman's Sparse Library and the number of matrices for which they gave the best performance (left); the performance of *CSRLenGoto* with respect to ZZ-CSR (right)

horizontal "stripes", consisting of rows. Each stripe is then assigned to a thread. In such a row-based decomposition, the elements of a row is never split among different partitions. This decomposition may fail to divide the work equally among threads if the matrix is highly irregular; that is, if there are unusually long or many empty rows in the matrix. The recent *merge-based* method of Merrill and Garland[5] provides strict load balancing between threads by allowing a row at a partition border to be split. Their approach is based on the "merge-path" algorithm, which is an efficient parallelization of the merge operation of two sorted lists, A and B, where each thread is assigned an equal share of the total $|A| + |B|$ steps. The merge-path algorithm is applied to the problem of partitioning CSR matrix data by logically merging the `rows` array, which is sorted, with the natural numbers from 0 to NZ (ie, indices of the `cols` and `vals` arrays). Once the matrix is partitioned, each partition can be processed sequentially by a thread. We implemented a multi-threaded version of *CSRLenGoto* by integrating it into the *merge-based SpMV* code base[25] as the kernel function that is executed sequentially by a thread. In principle, it should be possible to use *CSRLenGoto* with any CSR-based matrix partitioning approach for concurrent execution. We preferred merge-based SpMV because it is state-of-the-art and its code is public.

In our experiment, we used the same data set that contains 1878 matrices with 10000 to 200 million non-complex nonzero values (Figure 7). Our setup is the same as explained in the single-threaded performance evaluation section except the following: We ran the experiment only on our Intel computer (see Table 1 for properties) because the merge-based SpMV code base requires the Intel compiler (icc) and because we wanted to compare parallel *CSRLenGoto* against Intel's Math Kernel Library (MKL) as well. Concurrent execution is obtained via OpenMP pragmas. We ran three methods in this benchmarking:

- MKL's inspector-executor interface: In the inspector phase, we first create an MKL-internal CSR matrix via `mkl_sparse_d_create_csr`, then call `mkl_sparse_set_mv_hint` and `mkl_sparse_optimize`. As the SpMV function, we use `mkl_sparse_d_mv`.
- Merge-based SpMV: In the preprocessing phase, the matrix is partitioned according to the merge-path method. Plain CSR-based SpMV function is executed on each partition. This is Merrill and Garland's original code.
- *CSRLenGoto*: In the preprocessing phase, the matrix is partitioned according to the merge-path method, then the CSR data is converted to CSRLen. *CSRLenGoto*'s SpMV function is executed on each partition.

For all methods, we again timed the preprocessing/inspector phases separately from the SpMV phase. We used 16 threads, which is the number of hardware threads available on the CPU (8 cores, 2 hyper-threads per core). CSR-CSRLen conversion is straightforwardly parallelized via OpenMP compiler directives.

Performance of *CSRLenGoto* with respect to MKL and merge-based methods are given in Figure 12. The overall speedup trend related to the mean row length seems to apply in the multi-threaded context as well. On the average, *CSRLenGoto*'s performance is 1.34× of MKL and 1.33× of merge-based SpMV among matrices with short rows (ie, mean row length 8.0 or less). For matrices with medium-length rows (ie, mean row length between 8.0 and 14.5), the average performances are 1.19× and 1.22× for MKL and merge-based SpMV, respectively. The maximum performance obtained is 2.46× wrt MKL and 2.33× wrt merge-based SpMV. Similar to the sequential execution, slow-down cases dominate when the matrices have long rows. Preprocessing costs with respect to one MKL SpMV function call are shown in Figure 13. On the average, preprocessing takes time equivalent to 0.52, 0.52, and 0.34 SpMV calls for matrices with short rows, medium-length rows, and long rows, respectively. Break-even points are also given in Figure 13; here, we excluded the matrices for which the performance is less than 1.01× (744 matrices). For the remaining 1134 matrices, the average break-even point is 5.08 SpMV iterations.

6 | RELATED WORK

Since SpMV is used in many scientific problems, improving its performance implies a wide impact. For this reason, optimizing SpMV has been extensively studied by many researchers (see, eg, other works[1,5,15,21-23,26-28]). Two recent works provide a detailed survey.[2,3]

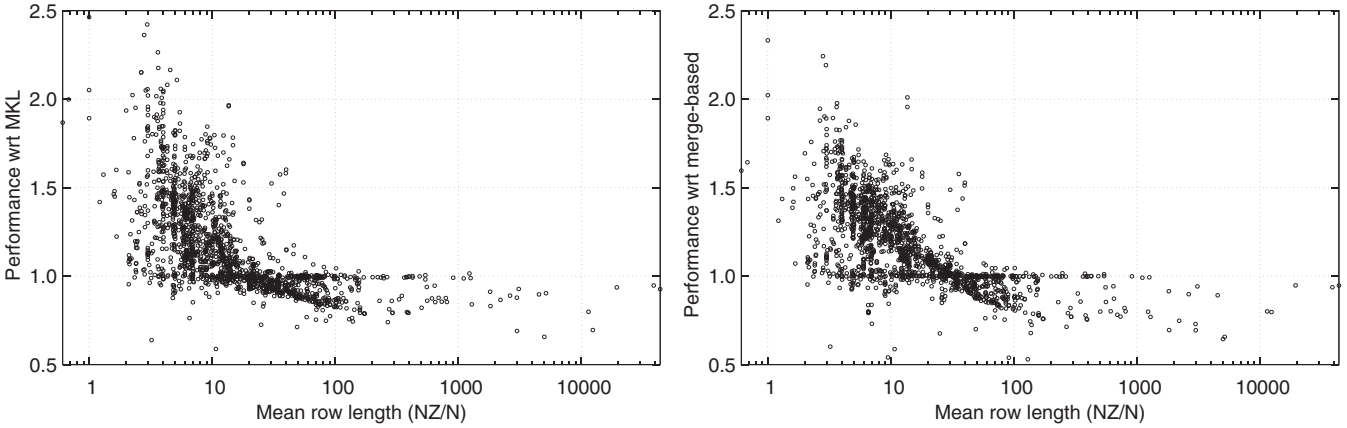


FIGURE 12 Performance of parallelized *CSRLenGoto* with respect to MKL (left) and merge-based SpMV (right)

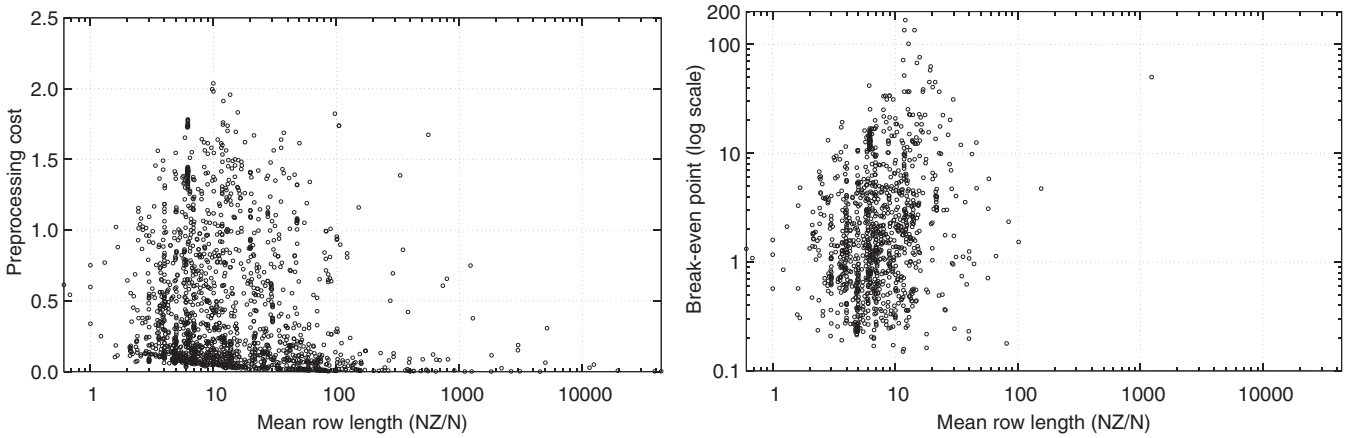


FIGURE 13 Preprocessing costs (left) and break-even points (right) of parallelized *CSRLenGoto* with respect to one MKL SpMV cost

Our motivation in this work has been to use a format similar to CSR so as to avoid heavy preprocessing/inspection costs. Reordering the matrix elements and using a storage format other than CSR may give substantial speedup for some matrices.[13,21-23,28] However, the cost of preprocessing and storage format conversion in these approach can be equivalent to tens or even hundreds of SpMV iterations.[6,19,21-23] Compensating these costs can be possible only if the optimized SpMV routine will be called hundreds or thousands of times. Yet, it is common for the number SpMV iterations in iterative solvers to be less than a hundred. In such context, SpMV optimization approaches that have high preprocessing costs do not give benefits. For this reason, some recent work have focused on optimizing SpMV with little or no preprocessing cost on the GPU,[6,7,9,10] CPU,[8] or both.[5] Among those for the GPU, Ashari et al propose to group matrix rows of the same length together to speed up SpMV.[6] Greathouse and Daga dynamically distribute CSR matrix data to GPU threads and efficiently use GPU's scratchpad memory to obtain improvements.[7] Liu and Schmidt apply no preprocessing to the CSR matrix data in their *LightSpMV* library; they achieve speedup by using GPU's atomic operations, warp shuffle mechanism, and dynamic load balancing.[10] In another GPU-oriented method, Liu et al propose the LSRB-CSR format, where they decompose the matrix data into segments tuned according to the architectural properties and the warp length of the target GPU.[9] Ohshima et al investigate the effect of various OpenMP thread scheduling settings on the CSR-based SpMV performance on CPU.[8] Merrill and Garland propose a row-oriented decomposition technique for the CSR format based on the *merge-path* approach.[5] Their technique promises better load balancing for parallel execution of SpMV on both CPU and GPU, in particular for matrices with irregular shapes.

SpMV is a highly parallelizable computation because the matrix rows can be processed independent from each other. The usual approach for parallelization is to decompose the matrix to as many pieces as the number of threads available on the target platform. There exist row-oriented, one-dimensional decomposition approaches,[5,28,29] and two-dimensional ones.[30-33] After the matrix data is decomposed, each portion is processed using a sequential kernel. Therefore, even though we essentially proposed an improvement of the sequential execution of SpMV, our method can be combined with existing matrix partitioning methods for parallelization. As an example, we integrated our method as the kernel SpMV implementation in Merrill and Garland's state-of-the-art merge-based SpMV approach and showed that our method can provide substantial speed improvement in a multi-threaded setting (see Section 5).

Our method is based on the fundamental idea of loop unrolling. We had previously shown that loop unrolling provides speedup for SpMV,[34] but we had done that for formats that re-arrange matrix data. Another paper that investigates loop unrolling for SpMV is by Mellor-Crummey and Garvin.[13] Here, a sparse matrix representation called LCSR that reorders matrix elements is used. For this reason, converting CSR format to LCSR involves

restructuring both the `vals` and `cols` arrays. The conversion from CSR to our CSRLen format does not process these two arrays; we rebuild the `rows` array only (see Figure 5). *CSRLenGoto* method can be modified to work with a reordered matrix; however, this does not fit the "low-overhead preprocessing" motivation we had put forward. Similarly, in principle it is possible to reduce memory traffic by combining *CSRLenGoto* with data compression approaches such as value compression in the CSR-VI format[27] or other data compression techniques[35]; however, compressing the values or column indices requires a preprocessing phase over the `vals` and `cols` arrays and may incur additional costs such as frequently accessing a hashtable. Therefore, we did not prefer incorporating these compression techniques. Kumahata et al[36] employ loop unrolling to optimize the SpMV portion of the high performance conjugate gradient (HPCG) problem on the K computer. Their approach is based on generating an SpMV loop for each possible row length value. In our approach, it is possible to have a single SpMV function. That single function can be used for any matrix, as long as the maximum row length of the matrix is smaller than the maximum row length assumed when implementing our code.

7 | CONCLUSION

We proposed a new SpMV implementation, named *CSRLenGoto*, that relies on complete loop unrolling and computed jump instructions based on row lengths. Our method gives substantial speedup for matrices whose mean row length is not high; for these matrices, we measured up to 2.46× and on the average 1.29× speedup with respect to Intel's MKL. *CSRLenGoto* uses a variation of the CSR sparse matrix storage format, which can be built via a low-overhead preprocessing phase. On the average, the cost of preprocessing is equivalent to half of the time that would be spent in a single execution of the SpMV function. This way, *CSRLenGoto* is able to pay off its preprocessing cost and start bringing speed benefits in only 5.08 iterations of the SpMV function on the average.

ORCID

Barış Aktemur http://orcid.org/0000-0002-1414-9338

REFERENCES

1. Goumas GI, Kourtis K, Anastopoulos N, Karakasis V, Koziris N. Understanding the performance of sparse matrix-vector multiplication. Paper presented at: 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008); 2016; Toulouse, France.
2. Langr D, Tvrdik P. Evaluation criteria for sparse matrix storage formats. *IEEE Trans Parallel Distrib Syst*. 2016;27(2):428-440.
3. Filippone S, Cardellini V, Barbieri D, Fanfarillo A. Sparse matrix-vector multiplication on GPGPUs. *ACM Trans Math Softw*. 2017;43(4).
4. Saad Y. *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA: SIAM; 2003.
5. Merrill D, Garland M. Merge-based parallel sparse matrix-vector multiplication. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16); 2016; Salt Lake City, UT.
6. Ashari A, Sedaghati N, Eisenlohr J, Parthasarathy S, Sadayappan P. Fast sparse matrix-vector multiplication on GPUs for graph applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14); 2014; New Orleans, LA.
7. Greathouse JL, Daga M. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14); 2014; New Orleans, LA.
8. Ohshima S, Katagiri T, Matsumoto M. Performance optimization of SpMV using CRS format by considering OpenMP scheduling on CPUs and MIC. Paper presented at: 2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs; 2014; Aizu-Wakamatsu, Japan.
9. Liu L, Liu M, Wang C, Wang J. LSRB-CSR: A low overhead storage format for SpMV on the GPU systems. Paper presented at: 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS); 2015; Melbourne, Australia.
10. Liu Y, Schmidt B. LightSpMV: faster CUDA-compatible sparse matrix-vector multiplication using compressed sparse rows. *J Signal Process Syst*. 2018;90(1):69-86.
11. Gropp W, Kaushik D, Keyes D, Smith B. Toward realistic performance bounds for implicit CFD codes. In: *Parallel Computational Fluid Dynamics 1999*. New York, NY: Elsevier; 1999.
12. Pichel JC, Heras DB, Cabaleiro JC, Rivera FF. Improving the locality of the sparse matrix-vector product on shared memory multiprocessors. In: Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing; 2004; Coruna, Spain.
13. Mellor-Crummey J, Garvin J. Optimizing Sparse Matrix-Vector Product Computations Using Unroll and Jam. *Int J High Perform Comput Appl*. 2004;18(2):225-236.
14. Davis TA, Hu Y. The University of Florida sparse matrix collection. *ACM Trans Math Softw*. 2011;38(1).
15. Vuduc R, Demmel JW, Yelick KA. OSKI: a library of automatically tuned sparse matrix kernels. *J Phys Conf Ser*. 2005;16:521.
16. Byun J-H, Lin R, Yelick KA, Demmel J. Autotuning Sparse Matrix-Vector Multiplication for Multicore. [Technical Report]. Berkeley, CA: Electrical Engineering and Computer Sciences, University of California, Berkeley; 2012.
17. Li J, Tan G, Chen M, Sun N. SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13); 2013; Seattle, WA.
18. Sedaghati N, Mu T, Pouchet L-N, Parthasarathy S, Sadayappan P. Automatic selection of sparse matrix representation on GPUs. In: Proceedings of the 29th ACM on International Conference on Supercomputing (ICS'15); 2015; Newport Beach, CA.
19. Yılmaz B, Aktemur B, Garzarán MJ, Kamin S, Kiraç F. Autotuning runtime specialization for sparse matrix-vector multiplication. *ACM Trans Archit Code Optim*. 2016;13(1).

20. Elafrou A, Goumas G, Koziris N. Performance analysis and optimization of sparse matrix-vector multiplication on modern multi- and many-core processors. Paper presented at: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW); 2017; Lake Buena Vista, FL.
21. Belgin M, Back G, Ribbens CJ. A library for pattern-based sparse matrix vector multiply. *Int J Parallel Program*. 2011;39(1):62-87.
22. Karakasis V, Gkountouvas T, Kourtis K, et al. An extended compression format for the optimization of sparse matrix-vector multiplication. *IEEE Trans Parallel Distrib Syst*. 2013;24(10):1930-1940.
23. Liu W, Vinter B. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In: Proceedings of the 29th ACM on International Conference on Supercomputing (ICS'15); 2015; Newport Beach, CA.
24. Yzelman AN. Sparse Library. http://albert-jan.yzelman.net/software.php. Accessed November 2017.
25. Merrill D. Merge-based Parallel Sparse Matrix-Vector Multiplication. https://github.com/dumerrill/merge-spmv. Accessed November 2017.
26. Buluç A, Fineman JT, Frigo M, Gilbert JR, Leiserson CE. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In: Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures (SPAA'09); 2009; Calgary, Canada.
27. Kourtis K, Goumas G, Koziris N. Exploiting compression opportunities to improve SpMxV Performance on shared memory systems. *ACM Trans Archit Code Optim*. 2010;7(3).
28. Williams S, Oliker L, Vuduc R, et al. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. *Parallel Comput*. 2009;35(3):178-194.
29. Yang W, Li K, Mo Z, Li K. Performance optimization using partitioned SpMV on GPUs and multicore CPUs. *IEEE Trans Comput*. 2015;64(9):2623-2636.
30. Catalyurek UV, Aykanat C. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans Parallel Distrib Syst*. 1999;10(7):673-693.
31. Martone M. Efficient multithreaded untransposed, transposed or symmetric sparse matrix–vector multiplication with the Recursive Sparse Blocks format. *Parallel Comput*. 2014;40(7):251-270.
32. Yzelman AN, Bisseling RH. Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods. *SIAM J Sci Comput*. 2009;31(4):3128-3154.
33. Yzelman AJN, Roose D. High-level strategies for parallel shared-memory sparse matrix-vector multiplication. *IEEE Trans Parallel Distrib Syst*. 2014;25(1):116-125.
34. Kamin S, Garzarán MJ, Aktemur B, et al. Optimization by runtime specialization for sparse matrix-vector multiplication. In: GPCE; 2014.
35. Willcock J, Lumsdaine A. Accelerating sparse matrix computations via data compression. In: Proceedings of the 20th Annual International Conference on Supercomputing (ICS'06); 2006; Cairns, Australia.
36. Kumahata K, Minami K, Maruyama N. High-performance conjugate gradient performance improvement on the K computer. *Int J High Perform Comput Appl*. 2016;30(1):55-70.

How to cite this article: Aktemur B. A sparse matrix-vector multiplication method with low preprocessing cost. *Concurrency Computat Pract Exper*. 2018;e4701. https://doi.org/10.1002/cpe.4701