Vectorizing SpMV by Exploiting Dynamic Regular Patterns

Xin You†
Beihang University, Beijing, China
youxin2015@buaa.edu.cn

Changxi Liu†
National University of Singapore,
Singapore, Singapore
changxi_liu@u.nus.edu

Hailong Yang‡
Beihang University, Beijing, China
hailong.yang@buaa.edu.cn

Zhongzhi Luan
Beihang University, Beijing, China
07680@buaa.edu.cn

Depei Qian
Beihang University, Beijing, China
depeiq@buaa.edu.cn

ABSTRACT
Modern optimizing compilers can exploit memory access and computation patterns to generate vectorized codes. However, such patterns in irregular programs such as SpMV are unknown until runtime due to the input dependence. Thus, either compiler’s static optimization or profile-guided optimization cannot represent the patterns for any common input, which leads to suboptimal vectorization. To address the above drawback, we propose DynVec, a framework to automatically exploit regular patterns buried deeply inside SpMV programs and apply corresponding optimizations for better vectorization. Due to the ability to represent instruction features and identify regular patterns with effective feature extraction and data re-arranging methods, DynVec can generate highly efficient vectorized codes by replacing gather/scatter/reduction operations with optimized operation groups. We evaluate DynVec on optimizing SpMV with representative sparse matrix datasets. The experiment results show that DynVec achieves significant speedup compared to the state-of-the-art SpMV implementations across a range of platforms.

CCS CONCEPTS
• Computer systems organization → Single instruction, multiple data; • Software and its engineering → Just-in-time compilers; • General and reference → Performance.

KEYWORDS
SpMV, Regular Pattern, Vectorization, Performance Optimization

ACM Reference Format:
Xin You, Changxi Liu, Hailong Yang, Zhongzhi Luan, and Depei Qian. 2022. Vectorizing SpMV by Exploiting Dynamic Regular Patterns. In 51st International Conference on Parallel Processing (ICPP ’22), August 29-September 1, 2022, Bordeaux, France. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3545008.3545042

†Both authors contributed equally to this paper.
‡Corresponding author.

1 INTRODUCTION
Sparse matrix-vector multiplication (SpMV) has been widely adopted in both traditional and emerging research fields such as high-performance computing, big data analysis, and deep learning, which exhibits an increasing demand for higher performance. One major difference between sparse and dense matrix-vector multiplication is whether the data access and computation patterns can be known before runtime. For SpMV, the above patterns are strongly correlated with the input data and can only be known during runtime. Such uncertainty of SpMV presents difficulties such as irregular memory accesses and writing conflicts for compiler optimization.

Meanwhile, the SIMD architecture has been pervasively adopted on modern CPUs, which provides vectorized operations such as reduction, gather and scatter for boosting program performance. To exploit the vectorization capability of the CPU without the burden of programmers, compilers have developed powerful static analyses to vectorize programs automatically. However, due to limited analysis scope and agnostic input, the compiler optimizations for irregular programs (e.g., SpMV) often lead to sub-optimal performance. For instance, when there are memory access patterns with potential write conflicts, the compilers usually give up on vectorizing the computation, sacrificing performance for correctness. The above limitation of static compiler optimization leaves the large performance potential of vectorization unexploited for irregular programs such as SpMV.

One reason for the failure of modern compilers to generate efficient vectorization codes for SpMV is due to the unknown memory access patterns. For instance, when loading the data from discontinuous memory addresses, the compilers always generate gather operations for loading the memory. However, as shown in the cases of Figure 1, replacing the gather operation (Method 1) with (load, permute, blend) operation groups (Method 2) achieves better performance (details refer to Section 2). For the regular program, as shown in Figure 1(a), the compilers can automatically perform the vectorization optimization through static analysis. Whereas for the irregular program, as shown in Figure 1(b), since the memory access patterns can only be recognized during runtime, the compilers generate inefficient codes that load data from memory using gather operations (Method 1), other than optimized codes using (load, permute, blend) operation groups (Method 2). Specifically, we have applied similar optimization (Method 2) to SpMV on SuiteSparse [10] dataset, which achieves more than 2x performance speedup on average on KNL CPU compared to Intel icc compiler. Such a large performance opportunity motivates us to exploit the
The above observations indicate a huge performance opportunity for vectorization of irregular programs such as SpMV that cannot be achieved by compilers using static analysis. However, there are several challenges to realizing the above performance opportunities for SpMV. Firstly, the memory access and computation patterns can only be determined during runtime. Therefore, the detection approach to identify the regular patterns needs to be lightweight without offsetting the optimization potential. Secondly, the memory access and computation patterns can vary significantly across different inputs. Therefore, the optimization approach needs to effectively identify regular patterns regarding each input. Thirdly, due to the diverse SIMD architectures, the code generation approach needs to adapt the regular patterns to the architecture features for better performance.

To address the above challenges, we propose DynVec\footnote{DynVec is open source at https://github.com/buaa-hipo/DynVec-artifact}, a framework for optimizing SpMV programs by exploiting vectorization opportunities on SIMD architectures automatically with just-in-time (JIT) compilation techniques. Specifically, DynVec represents the runtime memory access and computation patterns of SpMV with instruction features derived from feature extraction method. Then, DynVec uses the hashing function to merge loop iterations with the same instruction features and write locations to avoid memory bloat. After that, DynVec re-arranges the data with the data re-arranger for better data locality. Finally, DynVec generates optimized SpMV codes with the code optimizer, which replaces gather/scatter/reduction operations with more efficient operation groups. When evaluating with representative datasets, DynVec can generate more efficient vectorized codes of SpMV on various SIMD architectures compared to the state-of-the-art implementations.

Specifically, this paper makes the following contributions:

- We identify the missing performance opportunities to leverage regular patterns within SpMV programs on SIMD architectures by replacing traditional gather/scatter/reduction operations with more efficient operation groups.
- We implement DynVec, a framework that realizes several optimization techniques such as feature extraction, data re-arranger, and code optimizer to automatically generate efficient vectorized codes.
- We evaluate DynVec with representative datasets on Intel Broadwell, Skylake, and KNL CPUs. The evaluation results show that the SpMV codes automatically generated by DynVec achieve better performance than the state-of-the-art implementations.

The remainder of this paper is organized as follows. Section 2 motivates our work by observations from real experiment results. Section 3 presents the design overview of DynVec. Section 4, 5, and 6 describes the implementation details of feature extraction, data re-arranger, and code optimization, respectively. Section 7 presents the evaluation results. We describe the related work in Section 8. We conclude this paper in Section 9.

2 MOTIVATION

For convenience, the descriptions of variables and operations referred in this paper are listed in Table 1 and Table 2. To understand the optimization opportunities, we take the gather/scatter operation...
Table 1: Descriptions of variables referred in this paper.

| Variable | Description |
|----------|-------------|
| N        | The vector length of the target SIMD architecture with the specified precision (e.g., for AVX512 double precision, N = 8). |
| T        | Data access order, which includes Increment Order (Inc), Equal Order (Eq), and Other Order (Other). |
| N_R      | The number of load/reduction instructions required to replace. |
| S(t)     | Permutation Address for t* permutation of load/store/reduction. |
| M(t)     | Mask used for t* blend instruction. |
| M_s      | Mask used for maskscatter instruction. |
| D        | Data array (mutable) |
| Idx      | Access array (immutable) |
| Idx^n    | Re-arranged access array, which is generated by Data Re-arranger from immutable access array Idx. |

operations widely used in irregular programs such as SpMV for illustration, which are frequently generated by existing compilers to speed up programs on SIMD architectures. However, we observe that replacing the gather operation with (load, permute, blend) operation groups (gather optimization) achieves better performance in several cases as shown in Figure 3(i)(ii). Similar performance improvement is also observed by replacing scatter operation with (permute, store) operation groups (scatter optimization), as shown in Figure 3(iii). For brevity, we only illustrate the observed performance opportunity for gather operation in the following discussion. We use the term LPB to represent the (load, permute, blend) operation group. To show the performance opportunity with gather optimization quantitatively, we evaluate the performance of the synthesized micro-benchmark (experiment details in Appendix A) on Intel Broadwell, Skylake, and KNL CPUs. The hardware/software details are listed in Section 7.1. Specifically, we manipulate the data arrays with different sizes (ranging from 32 to 8M elements) and corresponding access arrays, which enables replacing gather operations with 1/2/4/8 LPB. We repeat each run 1,000 times and report the average performance in Figure 3.

As shown in Figure 3(i), the performance improvement using gather optimization varies across the different array sizes. The gather optimization is more effective when the array size is small and less LPB are needed to replace the gather operations. The performance benefit becomes even larger when evaluating single precision floating-point values, as shown in Figure 3(ii). As the regular patterns within irregular programs such as SpMV are often small and fragmented, the gather optimization has great potential to improve the performance with better vectorization. Besides, as many applications (e.g., quantized neural networks) utilize low-precision computation, the performance benefit of gather optimization on single precision floating-point precision can reveal more performance opportunities.

In addition, the performance potential of the gather and scatter optimizations for parallel implementations using multi-threading is shown in Figure 4. The parallel implementations utilize OpenMP to run on the entire chip of Broadwell (14 threads), Skylake (12 threads), and KNL (64 threads) platforms, respectively. Similar performance speedups are observed as shown in Figure 3 for both gather and scatter optimizations. Specifically, for Broadwell and Skylake platforms, the average performance speedup of using only 1 LPB is even more significant than that of the serial cases (2.58x/1.54x compared to 1.74x/1.27x on Broadwell, 3.03x/1.67x compared to 1.72x/1.23x on Skylake for single/double precision). For the KNL platform, the average performance speedups are similar to serial cases (1.94x/1.31x compared to 1.93x/1.64x for single/double precision). For double-precision cases on KNL, although enabling multi-threading may increase the memory bandwidth contention (e.g., lower speedup), our approach can still improve performance (e.g., positive speedup) by exploring regular patterns. In sum, the experiment results reveal the performance opportunity of gather and scatter optimizations for both serial and parallel programs.

Moreover, the sparsity pattern of the data also affects the performance opportunity when optimizing the gather operations. For

Table 2: Descriptions of operations referred in this paper.

| Operation | Arguments | Description |
|-----------|-----------|-------------|
| gather    | D, Idx    | Gather data according to access indices Idx from data array D to a vector V_G. |
| scatter   | D, Idx, V | Scatter vector V to data array D according to access indices Idx. |
| reduce    | D, Idx, V | Add reduce vector V with data array D according to access indices Idx. |
| (v)load   | D, Idx    | Load data from data array D with index Idx, where vload is the SIMD load instruction. |
| (v)store  | D, Idx, V | Store data to data array D with index Idx, where vstore is the SIMD store instruction. |
| (v)add    | V_i, V_j  | Add two value V_i, V_j, and return the result, where vadd is the SIMD add instruction. |
| permute   | V, I      | The result vector R is re-ordered by indices I from the source vector V, where R[i] = V[I[i]], 0 ≤ i < N. |
| blend     | V_i, V_j, M | The result vector R is selected by mask M from two source vectors V_i and V_j, where R[i] = M[i] ? 0 : V_i [i] - V_j [i], 0 ≤ i < N. |
| vreduction | V        | The SIMD reduction instruction to reduce a vector V into a single value. |
| maskScatter | D, Idx, V, M_s | Scatter instruction with a mask M_s, where the value V[i] is stored only if M_s[i] is set. |

Figure 3: The performance speedup when applying the gather/scatter optimization with different array size on (a) Broadwell, (b) Skylake, and (c) KNL platforms, where DP and SP indicates double and single precision, respectively.
Figure 4: The performance speedup of parallel implementations when applying the \textit{gather/scatter} optimization with different array size on (a) Broadwell (14 threads), (b) Skylake (12 threads), and (c) KNL (64 threads) platforms, where DP and SP indicates double and single precision, respectively.

Figure 5: The distribution of \textit{gather} operations on SuiteSparse datasets that can be optimized by replacing with the \textit{(load, permute, blend)} operation groups.

instance, if the sparsity of the data is entirely random, there is hardly a chance of achieving better performance. Fortunately, most of the sparse data exhibit regular patterns to some extent. Figure 5 shows the percentage of sparse datasets that achieves better performance when applying \textit{gather} optimization to the SpMV program. The sparse datasets include 2,700 matrices from SuiteSparse [10]. The x-axis of Figure 5 indicates the number of LPB for replacement, and the y-axis indicates the percentage of the datasets. The legend of the Figure 5 represents the percentage of \textit{gather} operations that can be replaced by LPB. As shown in Figure 5, 18.4% of the datasets exhibit the \textit{gather} optimization opportunities with more than 25% of the \textit{gather} operations to be replaced by 1 LPB. Whereas 46.9% of the datasets contain more than 25% of the \textit{gather} operations that can be replaced with no more than 2 LPB. Moreover, 55.5% of datasets contain more than 75% of the \textit{gather} operations that can be replaced with 4 LPB. It is clear that there is a large performance space by applying \textit{gather} optimization to irregular programs with sparse data. The large performance opportunity motivates this paper to exploit dynamic regular patterns within SpMV programs for efficient code generation using vectorization.

3 DYNVEC: OVERVIEW

We design DynVec to exploit regular patterns within SpMV programs for better vectorization, that includes three important components such as \textit{1) feature extraction, 2) data re-arranger} and \textit{3) code optimizer}. After the above optimizations, DynVec uses LLVM JIT compilation to generate optimized SpMV executables for better vectorization. The design overview of DynVec is shown in Figure 6. Users only need to describe the SpMV computation using a lambda expression with its input data, and DynVec interprets the lambda expression and automatically generates an efficient implementation for a particular SIMD architecture. The runtime data of a particular program execution can be classified as \textit{mutable} data and \textit{immutable} data. The \textit{immutable} data is annotated by user (using \textit{const} keyword) to ensure it is unchanged during runtime, and it will be used to generate information to guide the optimization. Note that the value of the \textit{immutable} data is still unknown until runtime. \textit{DynVec} first interprets the lambda expression and generates the \textit{expression tree}. The \textit{expression tree} describes the computation process without concerning the specific optimizations. It captures operations such as \textit{gather, scatter and reduction} derived from the lambda expression with a left-to-right top-down parser.

Based on the \textit{expression tree}, \textit{Feature Extraction} extracts the computation patterns within the SIMD length to generate the \textit{Feature Table}. Specifically, as shown in Figure 7(a), we inspect the immutable data of \textit{gather/scatter/reduction} operations, and store the extracted patterns into the \textit{Feature Table}, which captures the data access and computation patterns of irregular programs. Each column of the \textit{Feature Table} is the computation process for one loop iteration, and the row represents the corresponding operations in the post-order-traversed \textit{expression tree}. In Figure 7(a), each column of the \textit{Feature Table} is denoted as \textit{iter}_k, where \textit{k} is the \textit{k}-th iteration, and each iteration processes data within the vector length of the target SIMD architecture. Thus, each element in the \textit{Feature Table} describes the \textit{instruction feature} at the current iteration, which includes data access order \textit{T} of a particular operation, number of operations for replacement \textit{N}_R (e.g., two \textit{loads} to replace
one gather), and permutation address $S$. Specifically, we consider two access/computation patterns are different when their instruction features are different, which results in different optimal code generation. The details on how to perform Feature Extraction and build Feature Table are presented in Section 4.

The Data Re-arranger (Figure 7 (b)) processes the Feature Table to merge the columns with the same instruction features as well as the columns with the same write locations. Specifically, the hash value of each column in the Feature Table is generated using hash combine functions. DynVec merges the columns with the same hash value to generate a hash map, which reduces the memory footprint during code generation. After merging columns with the same hash values, the DynVec continues to process the hash map to merge columns with the same write locations. Then, the Data Re-arranger re-organizes the immutable data of each gather/scatter/reduction operation into $p$ key-value pairs, where $p$ equals to the size of the hash map. Each key-value pair uses the hash value as its key (e.g., $k_1$ to $k_p$). For each key, DynVec groups the indices of merged iterations into tuples as its value. These re-arranged indices are packed as the new immutable data of each operation (e.g., $Idx_{gather}^R$), which represents a specific computation pattern. The details of the data re-arranger are described in Section 5.

Then, the Code Optimizer can generate optimized codes for each identified computation pattern with re-arranged immutable data. In general, it performs three kinds of optimizations, including gather optimization that replaces the gather with the operation groups (load, permute, blend), scatter optimization that replaces the scatter operations with the operation groups (permute, store), and reduction optimization that replaces the reduction operations with the operation groups (permute, blend, vadd). Thus, depending on the memory access and computation patterns (represented by the Feature Table) identified during runtime, the Code Optimizer applies the gather/scatter/reduction optimizations accordingly. The details of the code optimizations are described in Section 6. Eventually, DynVec utilizes the LLVM JIT compiler to dynamically generate the deployable function pointer for runtime execution.

## 4 FEATURE EXTRACTION

In this section, we introduce how DynVec extracts the instruction features such as data access order, number of operations to replace, and permutation address from specific immutable data.

### 4.1 Data Access Order

To represent the data access pattern, we classify the data access order within a specific vector length into several categories, including Increment Order, Equal Order and Other Order. The Increment Order indicates the values of the access array $Idx$ within a vector length $N$ is continuous and incremental, whereas the Equal Order indicates all values of access array $Idx$ within a vector length $N$ are the same. The rest of cases are all classified as Other Order, which needs further analysis to reveal the regular patterns. Since the data access with Increment Order and Equal Order exhibits clear regular patterns, these data accesses can be fulfilled with a single memory operation. Besides, Increment Order and Equal Order also reveal the well-supported regular computation patterns on SIMD architecture. Specifically, the reduction operations with Increment Order can be implemented with vadd operations due to non write conflicts, whereas the reduction operations with Equal Order can be implemented with vreduce operations supported by the underlying SIMD architecture.

### 4.2 Number of Operations to Replace

The number of operations to replace $NR$ differs for gather/scatter and reduction operations. Since the method of determining $NR$ for gather and scatter operations is similar, we only present the method for gather and reduction operations as follows.

**Gather Operation** - The $NR$ denotes the number of load operations used to replace a gather operation. The largest value of $NR$ is the vector length $N$ of the SIMD architecture. The algorithm listed in Figure 8(a) estimates $NR$ for the gather operation with particular immutable data $Idx$. Specifically, we first find the smallest source address $Idx_{source}^R$ of the gather operation (line 3). Then we find all other source addresses that falls in the range $Idx_{source}^R + Idx_{source}^R$ + $N$, which will be loaded together by the load operation at $Idx_{source}^R$ (line 7-12). We mark these addresses as loaded (line 12), and repeat until all addresses are marked as loaded (line 2). The estimated value $NR$ equals to the total number of iterations (line 14).

**Reduction Operation** - On the SIMD architecture with vector length of $N$, we need $\log(N)$ (permute, blend, vadd) operation groups at most to complete a SIMD reduction operation. The $NR$ denotes the number of (permute, blend, vadd) operation groups...
Figure 8: The algorithm to estimate $N_{R}$ for (a) gather and (b) reduction operations using access array $Idx$.

Listing 1: The algorithm to calculate the permutation address for reduction operation from access array $Idx$.

required, which ranges 0, 1, 2, ..., $\log N$. We use algorithm listed in Figure 8(b) to estimate $N_{R}$. Specifically, we assume that each target location $Idx_{i}^{R}$ needs to reduce $L_{R}$ values (line 8-13), and the largest $L_{R}$ (denoted as $L_{max}$, line 14) is the maximum number of values to be reduced across all target locations, where $N_{R}$ equals to $\text{ceil}(\log(L_{max} + 1))$ (line 15). Besides, we also calculate the mask $M_{S}$ (line 11) for storing (using maskScatter) reduced value to the first target location $Idx_{i}^{R}$ appeared in the index array $Idx$.

4.3 Permutation Address

For gather operations, the permutation address $S$ is only meaningful when the data access order of this operation is Other Order, because with data access order of Increment Order and Equal Order, gather operations can be optimized with a single load operation without any permute operations. For gather operations with Other Order, the algorithm listed in Figure 8(a) derives the permutation address (line 8), where $S(0)$ indicates the permutation address of the $i^{th}$ load. Therefore, DynVec obtains $N_{R}$ permutation addresses for corresponding $N_{R}$ loads, which is used to permute each loaded value for final blending. Besides, the mask $M^{(i)}$ of $i^{th}$ blend operation is also derived along with permutation address (line 9).

For reduction operation, Listing 1 describes how to calculate the permutation address $S$ and blend mask $M$ from the access array $Idx$. For example, Figure 9(a) demonstrates one of such cases for reduction operation, where $V_{0}$, $V_{3}$, $V_{4}$ and $V_{6}$ are reduced into memory location $I_{0}$ whereas $V_{1}$, $V_{2}$ and $V_{5}$ are reduced into memory location $I_{1}$. The value $V_{5}$ for $I_{1}$ is not reduced within the vector length. Therefore, we need $N_{R} = 2$ permutation addresses (denoted as $S(0)$ and $S(1)$) and corresponding masks for blend operation (denoted

Figure 9: An example of reduction operation (a), and corresponding reduction optimization (b).

Figure 10: An example of inter-iteration re-arrangement that re-organizes operation groups with the same write location (a) before merging and (b) after merging. (c) An example of intra-iteration re-arrangement that re-arranges immutable data within a vector length of gather operations.

5 DATA RE-ARRANGER

After merging loop iterations with same instruction features in Feature Table, the Data Re-arranger re-organizes the immutable data for better data locality by re-arranging the order of iterations (inter-iteration re-arrangement) and data within a vector length of each iteration (intra-iteration re-arrangement). As shown in Figure 10, the inter-iteration re-arrangement generates the re-arranged immutable data according to the order of the iterations in the merged hash map to represent the data of iterations with the same computation pattern to the same write location. Meanwhile, for the intra-iteration re-arrangement, the Data Re-arranger generates the re-arranged data of each operation based on the features within a vector length derived during feature extraction. For gather operation, the re-arranged immutable data $Idx_{i}^{R}$ equals to the load addresses $Idx_{i}^{R}$ derived using Figure 8(a), and the re-arranging method for scatter operation is similar to the gather operation. We do not re-arrange the immutable data for reduction operation to guarantee correctness. In sum, the Data Re-arranger re-arranges immutable data for better locality to facilitate further code optimization.
Figure 11: An example of gather operation (a) and corresponding gather optimization (b).

Figure 10(a) presents an illustrative example of inter-iteration re-arrangement with operation groups writing to the same location. Without merging the operations, two reduction operations (1 2) in Figure 10(a) to the target data array $D$ in the memory are required, which wastes both computation units and memory bandwidth. Figure 10(b) shows the computation pattern after merging the operations, where only one (vadd, reduction) operation group is required (3 4) in Figure 10(b). Although in this case we introduce one extra vector operation (vadd), it is far more efficient than one extra reduction operation. Figure 10(c) demonstrates intra-iteration re-arrangement within a vector length of four, where the data array $D$ is indirectly accessed based on the access array $Idx$. Using algorithm in Figure 8(a), the two access arrays $Idx$ of (0, 3, 1, 2) and (4, 10, 7, 12) are re-arranged as the new access arrays $Idx^{Gather}$ of (0) and (4, 10), which are sufficient to load all values using gather optimization.

6 CODE OPTIMIZER

6.1 Gather/Scatter Operation

Figure 11 illustrates an example of the gather optimization applied based on the derived features of gather operations, assuming the vector length is four, and the bit width of the permute vector is two. In this example, we use two (load, permute, blend) operation groups to replace one gather operation, where the first load reads $D_{0, D_{1}}$ and the second one reads $D_{4, D_{5}}$. Therefore, we can obtain $N_{R} = 2$, and permutation addresses $S^{(0)} = 0011$ and $S^{(1)} = 0011$. Specifically, we first load data $ABC$ and $EFGH$ to registers based on $D_{0}$ and $D_{4}$ of $Idx^{R}$. Then, based on the $S^{(0)}$, $S^{(1)}$, and the loaded values, we obtain $AABB$ and $EEFF$ with permute operation. After that, we apply the blend operation to $AABB$ and $EEFF$ with mask $M^{(0)} = 0110$ in order to obtain $AEEF$. It is clear that with gather optimization, using $NR$ (load, permute, blend) operation groups is sufficient to replace the gather operation.

For better efficiency, the code generation differs when the gather operation exhibits different data access patterns. The detailed code generation of each access pattern is listed in Table 3. For data access patterns of Increment Order and Equal Order, since the index of permute is constant, the indices of load operations for permute can be omitted. Whereas the scatter operation can be replaced by the combination of vload and store operations when the index of gather is continuous and incremental. For Other Order, the code generation for gather varies in terms of the number of operation groups ($N_{R}$) to replace. Considering the gather optimization may lead to negative results when the performance of (load, permute, blend) operation groups cannot outperform a gather operation, we generate optimized codes only when the optimization leads to positive results (based on the empirical study shown in Figure 3). Otherwise, we leave the original gather operations unchanged.

As shown in Table 4, after gather optimization, the size of index data avoided to be loaded is $N - N_{R}$. However, our optimization introduces ($N_{R} - 1) \times N$ extra data to be loaded as well as $N \times log_{2}N + (N_{R} - 1) \times N$ bits to record the additional information. Fortunately, on the cache hierarchy of the modern processor, the number of cache lines consumed by our method is the same as the original gather operation. Besides, the size of the extra data introduced by our method is always smaller than the size of the index data eliminated.

6.2 Reduction operation

To ensure the correctness after optimization, DynVec can be applied to reduction operations that are both associative and commutative,
Table 4: The comparison of the data size before and after the **gather** and **reduction** optimization.

| gather | index | data | additional data |
|--------|-------|------|-----------------|
| original | N     | N    | N x log₂(N) + (N₀ − 1) x N |
| optimized | N₀    | N x N₀ | N₀ x log₂(N₀) + N₀ x 1 |

| reduction | target index | target data | additional data | target data |
|-----------|--------------|--------------|-----------------|--------------|
| original  | N            | N            | N x log₂(N)     | N            |
| optimized | N₀           | N₀ x N₀      | N₀ x log₂(N₀)   | N₀ x log₂(N₀) |

for example add and multiply. Other reduction operations such as minus and division, can be transformed to add or multiply reduction operations with negative variance operations. Currently, DynVec supports the add reduction operation widely used in SpMV programs. Instead of generating code based on the distribution of write locations, we generate optimized reduction operation groups based on N₀. The details of how to generate efficient codes with different instruction features are listed in Table 3. Specifically, DynVec handles the potential write conflicts with reduction optimization. For data access pattern of Other Order, the optimized reduction operation group can be represented as a list of N₀ unrolled operation groups of (permute, blend, vadd). Figure 9(b) presents an example of reduction optimization with Other Order access, where the instruction features in Figure 9(a) are derived by Feature Extraction. Note that each (permute, blend, vadd) operation group reduces two values with the same write addresses in parallel (e.g., V₀, V₂ and V₄, V₆ with write address I₀ are reduced in parallel). For Equal Order, whose N₀ must be equal to log₂(N), we can directly use the reduction operation provided on the target architecture.

With the reduction optimization, we can reduce the number of add operations on the reduction data from N to 1, and the number of reduction operations from N to N₀, where N₀ is less than or equal to log₂(N). Although DynVec introduces additional operations such as permute, it can still speedup the reduction operation when executing N₀ Tpermute operations is faster than executing (N − 1) add operations and (N − N₀) reduction operations. Besides, the reduction optimization eliminates redundant memory loads and stores of the target data by the size of (N − N₀), as shown in Table 4. It also eliminates unnecessary load to the target index by the size of (N − N₀). However, the reduction optimization introduces extra overhead with additional data used by the permute operations (N₀ x log₂(N₀) bits). Thus, the reduction optimization is effective when the introduced data size is less than the data size of the unnecessary load and store eliminated.

7 EVALUATION

7.1 Experiment Setup

We evaluate DynVec with SpMV on platforms equipped with Intel Xeon E5-2680 v4 (Broadwell), Intel Xeon Gold 6126 (Skylake), and Intel Phi 7210 (KNL) CPU, respectively. The Broadwell platform is installed with 64-bit Ubuntu v20.04. The KNL and Skylake platforms are installed with CentOS v7.4 and v7.5, respectively. The compilation toolchain includes icc of OneAPI v2021.1 and LLVM v8.0.0 on all platforms. We compare DynVec with the SpMV implementations from Intel Math Kernel Library (MKL v2021.1), CSR5 [17], and CVR [28], where MKL uses the CSR data layout and the parameters of CSR5 and CVR remain as default. The SpMV implementations of MKL, CSR5, and CVR are compiled with icc (-O3 -xHost). For each run, we execute the SpMV 1,000 times and measure the average execution time. To evaluate SpMV, we select 2,700 matrices from SuiteSparse matrix datasets [10], including both regular and irregular matrices. Specifically, the size of the evaluated matrices (row x col) ranges from 1x2 to 2.1Mx2.1M, the number of non-zero values (nnz) ranges from 1 to 148.8M, and the sparsity (nnz/row) ranges from 0.13 to 555.5.

7.2 Performance of SpMV

The CSR format for SpMV implementation can decrease memory usage and provide more opportunities for compiler optimization (a.k.a., ICC implementation). However, in DynVec, we use COO instead of CSR. This is because the COO utilizes flat storage for non-zero values to compute SpMV and simplifies the lambda expression as well as corresponding analysis without loss of potential regularities. Figure 12 presents the achieved performance of all evaluated matrices with ICC, MKL, CSR5, CVR, and DynVec implementations on each evaluated platform, where the results are sorted by the achieved performance among all evaluated methods. In general, the achieved performance improves when utilizing the latest CPU platforms with wider vector length, especially for DynVec, which achieves the best performance on all evaluated datasets, resulting in 6.11 GFlops/s and 12.44 GFlops/s on Broadwell and Skylake platforms, respectively. Note that although the KNL platform supports AVX512 vector instructions, DynVec only achieves the maximum performance of 1.83 GFlops/s on all evaluated datasets (better than most of the other implementations). The reason can be attributed to the limited memory bandwidth on the KNL platform.

More detailed statistical results are shown in Figure 13 with the performance speedups of DynVec against the implementations of CSR, MKL, and CSR5 on Broadwell, Skylake, and KNL, respectively. Note that CVR only supports AVX512 intrinsics. Thus, we only evaluate CVR on Skylake and KNL platforms. Specifically, the histogram of Figure 13 presents the distribution of the achieved performance speedup of DynVec among all evaluated matrices against each implementation, where the bars located on the right of the red line (performance speedup larger than 1) indicate DynVec performs better than the compared implementation. Especially, the histogram indicates a more significant performance speedup when the bars are more concentrated to the right. As shown in Figure 13, DynVec
achieves the best performance of 48.6%, 56.1%, and 68.7% of the datasets on Broadwell, Skylake, and KNL, respectively. Compared to CSR, SpMV performs better utilizing DynVec on 48.8%, 66.0% and 92.0% of the datasets, with an average effective speedup\(^2\) of 1.33×, 1.45×, 2.82× on Broadwell, Skylake, and KNL, respectively. Compared to CSR5, DynVec achieves better performance on 97.5%, 79.4%, and 87.1% of the datasets, with an average effective speedup\(^2\) of 4.28×, 3.44×, and 1.85× on Broadwell, Skylake, and KNL, respectively. Compared to CVR, DynVec achieves better performance on 96.5% and 97.0% of the datasets, with an average effective speedup\(^2\) of 3.55× and 2.78× on Skylake and KNL, respectively. Whereas compared to MKL, DynVec performs better on 78.2%, 80.7%, and 77.9% of the datasets, with an average effective speedup\(^2\) of 1.94×, 4.24×, and 2.72× on Broadwell, Skylake, and KNL, respectively. In sum, DynVec exhibits significant performance speedup among most of the datasets on all evaluated platforms. With wider SIMD instructions available, DynVec demonstrates higher performance improvements compared to ICC implementations as well as manually optimized implementations in MKL and CSR5.

The reason why DynVec achieves better performance is that it is able to generate more efficient operation groups (e.g., gather/scatter optimization) and eliminate potential write conflicts (e.g., reduction optimization), which is hard to be exploited in static compilation with ICC and neglected in state-of-the-art implementations such as MKL and CSR5. However, on the datasets where the MKL implementations are better, the reason can be attributed to the split of the writes to the same memory location due to the different computation patterns identified in DynVec, which increases the generated operation groups. Moreover, on Broadwell CPUs, where AVX512 instructions are not supported, the limited vector length diminishes the advantage of DynVec for exploiting regular patterns to optimize. On datasets where CSR5 achieves the best performance, the sparsity patterns of the input matrices are friendly to the specially designed storage format in CSR5.

\(^2\)We use term average effective speedup to indicate the average speedup on the datasets excluding the ones exhibiting performance slowdown.

7.3 Understanding the Performance

To better understand the performance improvement, we utilize the roofline model analysis to derive the attainable performance of SpMV for each matrix in the dataset. Specifically, the attainable performance (Roof) can be calculated as Equation 1, where \(nnz\) indicates the number of non-zeros of a sparse matrix, \(m\) indicates the number of rows, and \(\text{bandwidth} (\text{Bytes})\) is obtained by the same empirical benchmark described in Section 2. Therefore, the Roof of a given sparse matrix can be regarded as the performance upper bound of the SpMV. The ratio of the achievable performance and the attainable performance (Roof) represents the achieved performance efficiency, which is shown in Figure 14. Note that the histogram of DynVec is more concentrated to the right (closer to 1), indicating the implementations using DynVec is more efficient than that of ICC, MKL, CSR5, and CVR on all evaluated platforms. Similarly, the CDF curve of DynVec implementations also exhibits a slower slope than other implementations on all evaluated platforms.

\[
\begin{align*}
\text{Flops} & = 2 \times nnz \\
\text{Bytes} & = nnz \times (8 + 4 + 8) + m \times (8 + 4) + 4 \\
\text{Roof} & = \frac{\text{Flops/Bytes} \times \text{bandwidth (Bytes)}}{}
\end{align*}
\]

For the detailed analysis of the performance improvement with DynVec, we obtain representative performance metrics for SpMV using PAPI on all evaluated platforms. Due to the limited hardware counters, we performed several measurements to collect the reported performance metrics. Each measurement runs 1,000 times, and the average of each performance metric is reported. In our evaluations, DynVec has higher CPI (\(\sim 2x\)) and significantly less total instructions executed (more than 50% less) compared to other methods on all platforms at all sizes of non-zeros. The higher CPI comes from two folds. Firstly, DynVec has more significant stalls (\(\text{STALL\_RATE}\)) than other implementations (up to \(10^3\) more compared to CSR implementation), which can be attributed to higher data/instruction cache misses (\(L1_{ICM\_PER\_INS} / L1_{DCM\_PER\_INS}\)) and penalty of the branch misprediction (\(BR_{MSP\_RATE}\)). Secondly, as DynVec identifies more vectorization opportunities, it generates more vectorized instructions, which leads to higher CPI compared to less vectorized implementations. In sum, although DynVec results in higher CPI, it can exploit the regular patterns and generate more efficient vectorized codes, which contribute to fewer instructions executed and thus better performance.
We draw the box plot by the range of the number of non-zeros via JIT compilation. Therefore, it carries the overhead of JIT compilations varies across different inputs due to the diverse sparsity patterns [11, 12, 16, 17, 28]. Further reducing the JIT overhead requires another research effort.

Figure 15: The overhead of DynVec generating SpMV codes on Broadwell, Skylake, and KNL.

### 7.4 Overhead Analysis

The overhead of DynVec consists of two parts, including code analysis and JIT compilation. To better understand the overhead of DynVec, we present the overhead as the number of iterations that can compensate for the time for code analysis and JIT compilation. Specifically, the overhead (\( n \)) is defined as \( \frac{T_o}{T_{ref} - T_{DynVec}} \), where \( T_o \) indicates the time spent for DynVec analysis and JIT compilation, and \( T_{ref} \) represents the execution time of ICC and DynVec implementation, respectively. Figure 15 illustrates the overhead of DynVec generating SpMV codes on Broadwell, Skylake, and KNL. The immutable data annotated \( \sigma \) indicates the data is immutable across iterations. When dealing with different inputs, the immutable data annotations remain the same. Compared to the polyhedral approaches that require heavy manual effort to design a polyhedron for effective code transformation, DynVec only requires annotating the immutable data and thus relieves the burden of programmers.

As irregular programs (e.g., PageRank) for performance optimization.

**Performance potential for parallel programs** - The experiment results shown in Figure 4 have proved the effectiveness of our approach on parallel programs. However, due to implementation complexity (e.g., load balancing), currently DynVec only supports vectorization optimization for serial SpMV programs. Note that even with parallel implementations available, improving the serial performance with optimized ILP still matters since it dominates the single thread/process execution. We leave the extension to parallel programs for future work.

8 RELATED WORK

There are various works to optimize SpMV for its irregular memory access. In addition, there are research works to optimize irregular programs in general. We discuss the related works from the following three aspects.

**Utilizing efficient sparse storage** - Many sparse storage formats have been proposed for SpMV programs for future work.

8 RELATED WORK

There are various works to optimize SpMV for its irregular memory access. In addition, there are research works to optimize irregular programs in general. We discuss the related works from the following three aspects.

**Utilizing efficient sparse storage** - Many sparse storage formats have been proposed for SpMV programs for future work.

8 RELATED WORK

There are various works to optimize SpMV for its irregular memory access. In addition, there are research works to optimize irregular programs in general. We discuss the related works from the following three aspects.

**Improving data locality** - As irregular programs (e.g., SpMV, graph application) suffer from poor data locality, various transformation methods for data layout and loop reordering were proposed to improve the temporal and spatial data reuse, including data clustering [13], full sparse tiling [24], communication avoiding rescheduling [21], and parallel tiling [22]. Especially by grouping data accesses across iterations, the parallel tiling inspector-executor transformations were proposed for coarse-grained parallelism [25].

**Performance benefit** - The performance of SpMV implementations varies across different inputs due to the diverse sparsity patterns [11, 12, 16, 17, 28]. DynVec exhibits the best performance among most of the evaluated datasets, which proves the effectiveness of DynVec to improve SpMV performance in general.

**Annotating immutable data** - The immutable data annotated in DynVec indicates the data is immutable across iterations. When dealing with different inputs, the immutable data annotations remain the same. Compared to the polyhedral approaches that require heavy manual effort to design a polyhedron for effective code transformation, DynVec only requires annotating the immutable data and thus relieves the burden of programmers.

**Applying to other programs** - The performance opportunities revealed in this paper exist in a broader range of programs except for SpMV. For example, replacing the gather/scatter/reduction operations with operation groups can achieve better performance efficiency. Therefore, DynVec can be generalized to apply to other irregular programs (e.g., PageRank) for performance optimization.

**Performance potential for parallel programs** - The experiment results shown in Figure 4 have proved the effectiveness of our approach on parallel programs. However, due to implementation complexity (e.g., load balancing), currently DynVec only supports vectorization optimization for serial SpMV programs. Note that even with parallel implementations available, improving the serial performance with optimized ILP still matters since it dominates the single thread/process execution. We leave the extension to parallel programs for future work.
Optimizing with parallelization - Different parallelization strategies have been proposed to optimize the irregular programs on specific architectures [7, 15] and on distributed memory architectures [4]. Besides, a series of inspector-executor approaches have been proposed to improve parallelization and data locality of irregular programs through polyhedral transformations during compilation [2, 8, 9, 20, 26]. In particular, Augustine et al. [2] proposed to mine regular sub-regions in the irregular data structure and then use compiler-generated inspector-executor code to explore vectorization. One limitation of the above approaches is that it relies on well-designed polyhedrons for effective code transformation during compilation, which is not a trivial task to achieve performance speedup. Whereas DynVec does not rely on the polyhedral framework and complements the polyhedra-based approaches.

9 CONCLUSION
In this paper, we address the limitation of traditional compilers that is unable to exploit the vectorization opportunity for optimizing irregular programs such as SpMV due to the unknown memory access and computation patterns. We propose DynVec, which identifies the regular patterns within SpMV programs with effective feature extraction and data re-arrange methods and then generates more efficient vectorized codes with corresponding optimizations. The experiment results demonstrate the effectiveness of our approach in optimizing the SpMV programs compared to the state-of-the-art implementations, achieving the best performance speedup of 48.6%, 56.1%, and 68.7% on Broadwell, Skylake, and KNL platforms, respectively.

ACKNOWLEDGMENTS
This work was supported by National Key Research and Development Program of China (No. 2020YFB1506703), and National Natural Science Foundation of China (No. 62072018 and 61732002). Hailong Yang is the corresponding author.

REFERENCES
[1] Arash Ashari, Naser Sedaghati, John Eisenlohr, and P Sadyayyan. 2014. An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on GPUs. In Proceedings of the 28th ACM international conference on Supercomputing. ACM, Muenchen, Germany, 273–284.
[2] Travis Augustine, Janarthanan Sarma, Louis-Noël Pouchet, and Gabriel Rodriguez. 2019. Generating piecewise-regular code from irregular structures. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, Phoenix, AZ, USA, 625–639.
[3] Satish Balay, Shrirang Abhyankar, Mark Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, W Gropp, et al. 2019. PETSc users manual.
[4] Ayon Barmalik and Rudolf Eigenmann. 2006. Optimizing irregular shared-memory applications for distributed-memory systems. In Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming. ACM, New York, New York, USA, 119–128.
[5] Aydin Buluç and John R Gilbert. 2011. The Combinatorial BLAS: Design, implementation, and applications. The International Journal of High Performance Computing Applications 25, 4 (2011), 496–509.
[6] Aydin Buluç, Samuel Williams, Leonid Oliker, and James Demmel. 2011. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In 2011 IEEE International Parallel & Distributed Processing Symposium. IEEE, Anchorage, Alaska, USA, 721–733.
[7] Daniele Bueno, John A Gunnels, Xinyu Que, Fabio Checconi, Fabrizio Petrini, Tai-Chung Tsau, and Chris Long. 2015. Optimizing sparse linear algebra for large-scale graph analytics. Computer 48, 8 (2015), 26–34.
[8] Kazem Cheshmi, ShoBáKamí, Michelle Mills Stout, and Maryam Mehri Dehnavi. 2017. Simplier: transforming sparse matrix codes by decoupling symbolic analysis.
[9] Kazem Cheshmi, ShoBá Kamí, Michelle Mills Stout, and Maryam Mehri Dehnavi. 2018. ParSy: inspection and transformation of sparse matrix computations for parallelism. In SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE / ACM, Dallas, TX, USA, 779–793.
[10] Timothy A Davis and Yifan Hu. 2014. The University of Florida sparse matrix collection. ACM Transactions on Mathematical Software (TOMS)38, 1 (2011), 1.
[11] Xiang Fei and Youhui Zhang. 2021. ReguZeD: Accelerating Vectorization of SpMV on Intel Processors through 2D-partitioning and Regular Arrangement. In 50th International Conference on Parallel Processing. 1–11.
[12] Constantino Gómiz, Filippo Mantovani, Ertch Focht, and Marc Casas. 2021. Efficiently running SpMV on long vector architectures. In Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 292–303.
[13] Hwansoo Han and Chau-Wen Tseng. 2006. Exploiting locality for irregular scientific codes. IEEE Transactions on Parallel and Distributed Systems 17, 7 (2006), 606–618.
[14] Changwan Hong, Aravind Sukumar-Rajam, Israt Nisa, Kunal Singh, and P Sadyayyan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. ACM, Washington, DC, USA, 300–314.
[15] Peng Jiang and Gagan Agrawal. 2018. Conflict-free vectorization of associative irregular applications with recent SIMD architectural advances. In Proceedings of the 2018 International Symposium on Code Generation and Optimization. ACM, Vösendorf / Vienna, Austria, 175–187.
[16] Chenyang Li, Tian Xia, Wenzhe Zhao, Nanning Zheng, and Pengju Ren. 2021. SpMon: Pursuing optimal vectorization and regular computation pattern in SpMV. In 2021 58th ACM/IEEE Design Automation Conference (DAC). IEEE, 661–666.
[17] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In Proceedings of the 29th ACM on International Conference on Supercomputing. ACM, Newport Beach/Irvine, CA, USA, 339–350.
[18] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In Proceedings of the 27th international ACM conference on International conference on supercomputing. ACM, Eugene, OR, USA, 273–282.
[19] Yucheng Liu, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2012. Distributed graphlab: A framework for machine learning in the cloud.
[20] Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C Davis, Mary Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olshansky, Anand Venkat, and Michelle Mills Stout. 2019. Sparse computation data dependence simplification for efficient compiler-generated inspectors. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, Phoenix, AZ, USA, 594–609.
[21] Marghoobeh Mohiyuddin, Mark Hoemmen, James Demmel, and Katheryn Yelick. 2009. Minimizing communication in sparse matrix solvers. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. ACM, Portland, Oregon, USA, 1–12.
[22] Yuyao Niu, Zhengyang Lu, Haonian Ji, Shuhui Song, Zhaon Jun, and Weifeng Liu. 2022. TitleSpGEMM: a tile method for parallel sparse general matrix-matrix multiplication on GPUs. In Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 90–106.
[23] Mark Stephenson and Saman Amarasinghe. 2005. Predicting unroll factors using supervised classification. In Proceedings of the international symposium on Code generation and optimization. IEEE Computer Society, San Jose, CA, USA, 123–134.
[24] Michelle Mills Stout, Larry Carter, and Jeanne Ferrante. 2001. Rescheduling for locality in sparse matrix computations. In International Conference on Computational Science. Springer, San Francisco, CA, USA, 137–146.
[25] Michelle Mills Stout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kearsel. 2002. Combining performance aspects of irregular gaussian-seidel via sparse tiling. In International Workshop on Languages and Compilers for Parallel Computing. Springer, College Park, MD, USA, 90–110.
[26] Michelle Mills Stout, Mary Hall, and Catherine Olshansky. 2018. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. Proc. IEEE 106, 11 (2018), 1921–1934.
[27] Anand Venkat, Mary Hall, and Michelle Stout. 2015. Loop and data transformations for sparse matrix code. ACM SIGPLAN Notices 50, 6 (2015), 521–532.
[28] Biwei Xie, Jianfeng Zhan, Xu Liu, Wanling Gao, Zhen Jia, Xiuwen He, and Lixin Zhang. 2018. Cvt: Efficient parallelization of sparse matrix-matrix multiplication. In Proceedings of the 2018 International Symposium on Code Generation and Optimization. ACM, Vösendorf / Vienna, Austria, 149–162.
[29] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2019. IA-SpGEMM: an imperative auto-tuning framework for parallel sparse matrix-matrix multiplication. In Proceedings of the ACM International Conference on Supercomputing. ACM, Phoenix, AZ, USA, 94–105.
A PAPER ARTIFACT DESCRIPTION
APPENDIX

Computational Artifacts: Yes

Artifact Description Details
Summarize the experiments reported in the paper and how they were run.

The evaluation scripts and source codes can be downloaded from the link given in Artifact 1. The compared state-of-the-art implementations, including ICC, MKL, CSR5, and CVR are all included in this repository. The evaluation source codes and scripts of motivation experiments are also included in this repository. In summary, one can build all motivation, DynVec and other compared implementations with the build script build.sh and run all evaluations with the script run.sh. The evaluation results are located in log/run_log/spmv/log_spmv_*.

Software Artifact Availability: The source code of DynVec can be downloaded from the following link.

https://github.com/buaa-hipo/DynVec-artifact

Hardware Artifact Availability: Not publicly available.

Data Artifact Availability: The input data used in the evaluation can be downloaded from the following link.

https://1drv.ms/u/s!AsGTYrgSALbmo-cjP_uKT2rtoAp2Iw?e=7ufdpw

Proprietary Artifacts: None.

Author-Created or Modified Artifacts: Artifact 1

Persistent ID: https://github.com/buaa-hipo/DynVec-artifact

Artifact name: Evaluation scripts and source codes of DynVec.

Operating systems and versions: The Broadwell platform is installed with 64-bit Ubuntu v20.04, respectively. The KNL and Skylake platforms are installed with CentOS v7.4 and v7.5.

Compilers and versions: OneAPI v2021.1 and LLVM v8.0.0.

Libraries and versions: Intel Math Kernel Library (MKL v2021.1), CSR5, CVR.

Input datasets and versions: SuiteSparse matrix datasets.