Vectorizing Sparse Matrix Computations with Partially-Strided Codelets

Kazem Cheshmi*
University of Toronto
Toronto, Canada
cheshmik@mcmaster.ca[1]

Zachary Cetinic*
University of Toronto
Toronto, Canada
zachary.cetinic@mail.utoronto.ca

Maryam Mehri Dehnavi
University of Toronto
Toronto, Canada
mmehride@cs.toronto.edu

***Abstract*—The compact data structures and irregular computation patterns in sparse matrix computations introduce challenges to vectorizing these codes. Available approaches primarily vectorize strided computation regions of a sparse code. In this work, we propose a locality-based codelet mining (LCM) algorithm that efficiently searches for strided and partially strided regions in sparse matrix computations for vectorization. We also present a classification of partially strided codelets and a differentiation-based approach to generate codelets from memory accesses in the sparse computation. LCM is implemented as an inspector-executor framework called LCM I/E that generates vectorized code for the sparse matrix-vector multiplication (SpMV), sparse matrix times dense matrix (SpMM), and sparse triangular solver (SpTRSV). LCM I/E outperforms the MKL library with an average speedup of 1.67×, 4.1×, and 1.75× for SpMV, SpTRSV, and SpMM, respectively. It is also faster than the state-of-the-art inspector-executor framework Sympiler [1] for the SpTRSV kernel with an average speedup of 1.9×.**

***Index Terms*—Vectorization, Parallel programming, Polyhedral analysis, Sparse matrix computation**

I. Introduction

Irregular computations, such as in sparse matrix codes, frequently appear in scientific and machine learning problems. The performance of these applications is noticeably improved if their code is vectorized to exploit single instruction multiple data (SIMD) capabilities of the underlying architecture. Vectorization potentially increases opportunities to optimize for locality, further increasing performance. SIMD instructions can efficiently vectorize groups of operations that access consecutive data, i.e. have a strided access pattern. However, vectorization becomes challenging when access patterns are not strided, especially in sparse matrix codes that use compact representation to store the matrix data.

Libraries use domain knowledge, such as kernel or sparsity pattern information to manually find groups of independent operations that can be vectorized efficiently. The operations in a sparse computation can be grouped in different ways for vectorization. One class of prior libraries, such as ELL-Pack [2], DIA [2], and OSKI [3] finds groups of operations that access strided locations in the memory and then vectorizes these groups. We call these groups of operations with strided memory accesses *strided regions*. These libraries map a strided region to a *BLAS* [4] kernel and then call an efficient BLAS implementation to vectorize operations of the strided region. To increase the size of strided regions in sparse matrix codes, data reorganization methods such as new storage formats [2] or padding with additional zero elements [3] are explored. These methods reorganize data to put operations that access strided locations next to each other. Another class of libraries, such as CVR [5] and CSR5 [6] finds groups of operations that only some of their operands access strided memory locations, which are called *partially strided regions*. A common technique to vectorize partially strided regions is to first store operands that are not strided in consecutive locations by using gather/scatter instructions [5], [7] and then vectorize them. Since nonzero elements of sparse matrices are stored consecutively, i.e. are strided, finding partially strided regions is always possible independent of the matrix sparsity pattern.

Compilers automate vectorization of sparse kernels to reduce or eliminate the need to manually optimize per kernel/-pattern, they also provide hardware portability. The automation in compilers is done by defining the space of vectorization as a search problem. They search the sparse kernel computation to find an efficient set of regions that can be vectorized by generating *codelets*. Searching the entire iteration space is expensive, hence prior methods differ based on how they reduce this search space. Methods such as the sparse polyhedral framework (SPF) [8] and Sympiler [1] search for strided codelets inside tiles, i.e. a group of operations in consecutive iterations of a loop. We refer to these methods as *tiling-based* approaches. Tiling-based approaches also add padding to increase opportunities for finding strided codelets. Regular piece-wise methods such as Augustine *et. al.* [9] detect strided regions from anywhere in the iteration space by generating polyhedral models with rectangle shapes. These works do not scale to large matrices (support matrices of up to 10M nonzeros) as their code size becomes large. The common limitation of prior automation approaches is that they only search for strided regions for vectorization.

In this work, we propose a *Locality-based Codelet Mining (LCM)* algorithm and classification of computations based on their strided behaviour, to automatically find strided and partially strided regions in sparse codes. A novel differentiation-based approach is also proposed to generate codelets. LCM mines the memory accesses in the sparse computation in

* Both authors contributed equally to this research.
[1] The author has since moved to McMaster University, Canada.

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

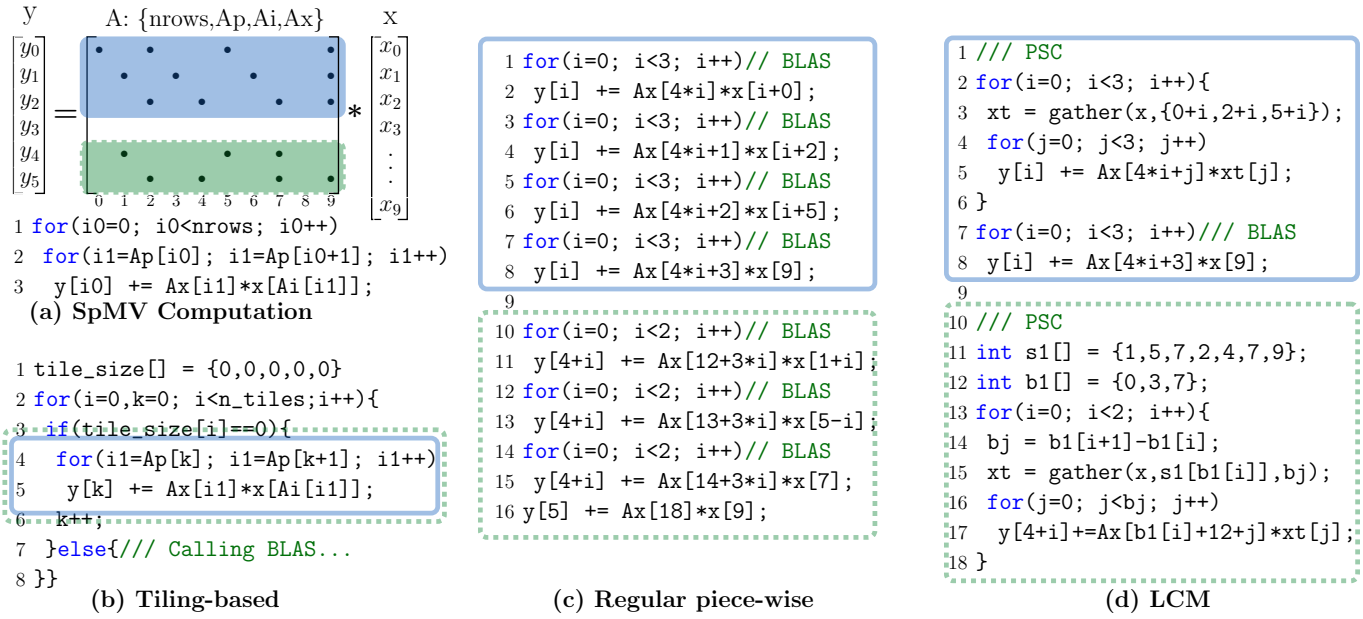

Fig. 1: The codes in Figures 1b–1d show three different approaches to vectorize the SpMV kernel for the two regions in the matrix as shown in Figure 1a. Each approach chooses a different strategy to group operations to be vectorized. The tiling approach searches for consecutive outermost iterations that access strided memory locations but because no outermost iterations with the same access pattern exist. It vectorizes one operation at a time as shown in line 5 in Figure 1b. The regular piece-wise approach searches across all operations and converts the SpMV code to a group of operations with strided memory accesses as shown in Figure 1c, each group is vectorized with BLAS. LCM groups operations that are strided and also operations with partial strides, as shown in Figure 1d.

polynomial time. In a *permutation and partitioning* step, LCM prunes the mining space by reordering operations for locality and as a result, also improves vectorization efficiency; this is because locality between operations increases the number of strided accesses. In a second step, LCM creates the combination of the largest possible BLAS and partially strided codelet (PSC) codelets from the reordered operations to minimize the overall execution time of the sparse kernel. LCM is implemented as an inspector-executor framework and supports multi-threaded execution. Compared to other automation frameworks, LCM has a low inspection time, generates compact code, and is scalable (results are shown for matrices with up to 300 million nonzeros). Our approach outperforms the MKL library with an average speedup of 1.67×, 4.1×, and 1.75× for sparse matrix-vector product (SpMV), sparse lower triangular solver (SpTRSV), and sparse times dense matrix (SpMM) kernels, respectively. It also outperforms Sympiler, an in-house implementation of SPF, and regular piece-wise methods with an average speedup of 1.92×, 4.1×, and 4.6× respectively.

Motivation: We use the example in Figure 1 to compare the codes generated from the automation methods, i.e. tiling-based and regular piece-wise approaches, to the generated code from LCM. The example shows that from the same set of operations in a sparse kernel, different vectorizable codes can be generated, however, the code from LCM is more efficient as it also contains vectorizable codelets for partially strided regions. We do not show the mining strategy in LCM and our PSC classification in this example.

For the region highlighted in blue in Figure 1a, the tiling-based approach (code in Figure 1b) can not find tiles and thus has to vectorize one multiply-add operation with a SIMD instruction. As a result, SIMD units are not utilized efficiently. We assume the number of SIMD units is three to simplify our representation. The regular piece-wise approach (code in Figure 1c) finds four regions with strided accesses and vectorizes them with BLAS codelets. Each codelet has three operations and thus utilizes the SIMD units. However, SIMD instructions do not access consecutive nonzeros in `Ax`, degrading spatial locality. Figure 1d shows the code from LCM which includes a partially strided codelet that vectorizes 9 operations (lines 1–6) and thus utilizes the SIMD units. It also iterates over the nonzero elements in `Ax` consecutively, thus enhancing spatial locality. Additionally, for the unstrided accesses to `x`, the LCM generated code reuses column indices {0,2,5}, and uses a gather instruction to store values consecutively in `xt` to further enhance locality. For the set of computations highlighted in green in Figure 1a, LCM generates an even more efficient code compared to other tools because the memory access patterns are less strided compared to the computations highlighted by blue. Tiling-based and regular piece-wise approaches are both unable to use SIMD units efficiently while the code from LCM improves spatial locality, makes better use of SIMD units, and uses gather instructions to load unstrided accesses to `x` as shown in Line 15 in Figure 1d.

$$\mathcal{I} = [i_0, i_1, nrows]\begin{Bmatrix} i_0 \geq 0 \wedge i_0 < nrows \\ i_1 \geq Ap[i_0] \wedge i_1 < Ap[i_0+1] \end{Bmatrix} \quad \begin{matrix} y[*]: & f(i_0,i_1) = i_0 \\ Ax[*]: & g(i_0,i_1) = i_1 \\ x[*]: & h(i_0,i_1) = Ai[i_1] \end{matrix}$$

iteration space — **access functions**

Fig. 2: Polyhedral representation of the SpMV kernel.

II. Partially Strided Codelets

This section introduces partially strided codelets and discusses their classification. We also present a cost model and a differentiation-based PSC detection strategy, both of which are used in the LCM algorithm to find PSCs in sparse matrix computations. We also show how PSCs are vectorized efficiently with vector instructions.

A. Definitions

Polyhedral model and data access functions. A loop nest that contains a set of statements is represented with a polyhedral model through an integer polyhedron set $\mathcal{I}$ and relations f. A statement is made of a data space described with $\mathcal{D}$ which is a disjoint set containing m elements $\mathcal{D}_0, \ldots, \mathcal{D}_m$. An integer polyhedral set $\mathcal{I} = [i_0, ..., i_n]$ is a collection of inequalities that create bounds for each dimension inside $i \in \mathcal{I}$. For each $\mathcal{D}_d \in \mathcal{D}$ a *data access function* f is used to describe how the data space $\mathcal{D}_d$ is accessed by the iteration space of $\mathcal{I}$. In other words, a data access function maps an iteration space to a data space, i.e. $f_{\mathcal{I}\rightarrow\mathcal{D}_d}$. The statement in the SpMV code in Figure 1a has three data spaces `y`, `Ax`, and `x` as well as three data access functions. Figure 2 shows the polyhedron sets for $\mathcal{I} = [i_0, i_1, nrows]$ and also the access functions corresponding to each data space in the SpMV code.

Codelet. A polyhedral model that has a convex integer polyhedron with no flow dependencies, i.e. read after write dependency (RAW) between access functions, is a codelet. A codelet only has one statement type and the operation in that statement is a SIMD-supported operation. Throughout the paper, an operation refers to an instance of a statement. An instance of a statement is each time the statement is executed for a given input.

A list of operations is shown in Figure 3a and its polyhedral representation including iteration space and data access functions are shown in Figure 3b. All operations in this model are independent and are instances of one statement (`y[*] += Ax[*] * x[*]`), with a multiply-add operation which is supported by Intel and AMD SIMD units. Although there is RAW between operations in Figure 3a, they are independent in SIMD units due to associativity of accumulation [9]. These properties satisfy the criteria for being a codelet.

Strided and unstrided data access function. A function that can be expressed with a linear combination of induction variables in $\mathcal{I}$ is a strided access function. If the function cannot be expressed as strided, it is an unstrided function. An access function is represented as $f_{\mathcal{I}\rightarrow\mathcal{D}_d}(\mathcal{I}) = q_d + s_d[i_0] + \ldots + s_d[i_n]$ where s_d has n dimensions and each dimension corresponds to an induction variable. $s_d[i_k]$ shows indices for i_k, and q_d is a constant integer offset. If f is strided with respect to i_k then $s_d[i_k]$ becomes an integer, this integer is the coefficient of i_k in the access function f.

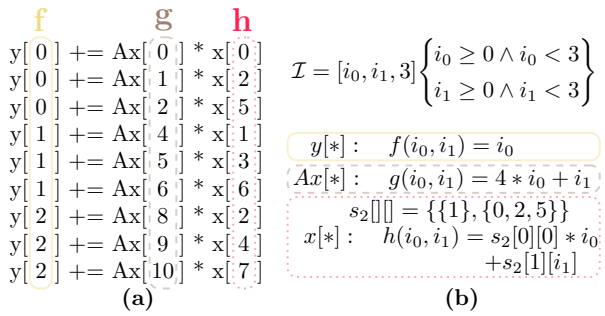

Fig. 3: **(a)** a list of operations. **(b)** the iteration space and access functions of the operation list in part (a).

The data access function f and g in Figure 3b are both strided and can be expressed as a linear combination of $[i_0, i_1]$. Function h is unstrided because it is not linear with respect to i_1. In Figure 3b we show how h is represented using s_2, $s_2[0][0]$ is the coefficient of i_0, $s_2[1]$ is an array of indices for i_1, and q_2 is zero and is not shown.

B. Partially Strided Codelet Classification

An efficient way to vectorize a codelet is to find operations with strided accesses across different iterations. This enables the vectorization of more operations and potentially increases data reuse between different iterations. However, current automation approaches are limited to vectorizing codelets with strided access functions. We call codelets with strided access functions BLAS codelets because an efficient BLAS [4] implementation is used in this work to vectorize these codelets. We also define a novel set of codelets called *Partially Strided Codelets* that have at most $m - 1$ strided access functions and at least one unstrided access function. These codelets can benefit from vectorization because they have one or more strided access functions.

PSCs are classified into different types based on the number of access functions that are strided. For example, in SpMV, SpTRSV, and SpMM that have codelets with three access functions, $m = 3$, two types of PSCs can be defined. The PSC type I codelet is used when two of access functions are strided, and the PSC type II codelet is used when only one access function is strided. For example, the operations in Figure 3b are a PSC I with one unstrided access function h.

C. Partially Strided Codelet Cost Model

We use an empirical cost model to estimate the execution time of codelets. To measure the cost of a codelet, similar to [10], we account for arithmetic operation cost and measure the memory access cost due to load/store operands. Because PSC codelets have unstrided accesses and indirection, we

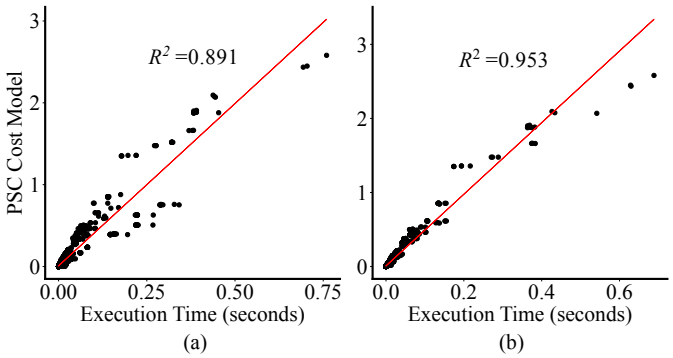

Fig. 4: Correlation between the PSC cost model and execution time for an Intel (a) and an AMD (b) processor.

additionally account for the cost of indexing for load and stores. The cost of a codelet l (C_l) is:

$$cost(C_l) = c_{op} \times \frac{|p|}{V} + c_{st} \times \frac{|s_0 + q_0|}{V} + c_{ld} \times \frac{1}{V} \sum_{d=1}^{m=3} |s_d + q_d| \quad (1)$$

The arithmetic cost of a codelet is obtained by dividing the number of arithmetic operations $|p|$ by the vectorization factor V (because SIMD units execute every V operations at once) and then multiplying the result by c_{op}. c_{op} is the number of cycles for an arithmetic operation type in the target architecture. To measure the memory access cost of a codelet, since the cost of loads differ from the cost of stores on multicore processors, we separately measure the cost of memory accesses from functions that contribute to stores and then add it to the cost of memory accesses from functions that contribute to loads. In this work we use codelets with three access functions, $m = 3$, as shown with f, g, and h in Figure 3. Function f contributes to stores and is expressed with $s_0 + q_0$, this expression is obtained from the access function representation explained in Section II-A. $|s_0+q_0|$ in Equation 1 shows the number of store operands plus the indexing cost, i.e. the size of arrays s_0 and q_0. Then the number of stores is multiplied with c_{st} which is the cost of a store and is divided by V. Functions g and h contribute to loads in a codelet and thus $|s_1+q_1|$ and $|s_2+q_2|$ are added to measure load operands and their indexing cost. The number of loads is then multiplied with c_{ld} which is the cost of a load operation and then divided by V. The arithmetic cost of the codelet in Figure 3 is 9 assuming $c_{op} = V = 1$. The store access cost is 4 assuming $c_{st} = 1$. The load access cost is 25 assuming $c_{ld} = 1$.

Figure 4 shows the correlation between the codelet cost model and the execution time of codelets on an Intel Skylake and an AMD Epyc processor. We vectorize SpMV, SpTRSV, and SpMM computations with different codelets for all matrices in our dataset obtained from SuiteSparse [11]. The x-axis shows the execution time of a kernel for a matrix in seconds. The y-axis shows their corresponding cost, which is computed by adding the cost of all codelets in the sparse kernel for the specific matrix. As shown, the cost model predicts the execution time with a correlation of 0.89 on Intel and 0.95 on AMD. Our cost model shows a good correlation despite not accounting for cache effects. This is because the size of codelets is bounded to be small enough to fit into L1 cache (our heuristic is Section III-B explains this) and thus codelets will not get evicted from cache during execution. Also matrices and vectors are aligned in memory, so the number of misaligned memory accesses remains low.

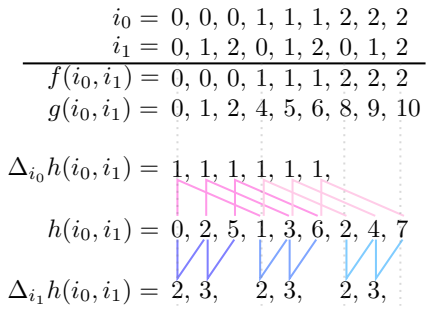

Fig. 5: Taking the derivative of the access function h with respect to i_0 and i_1.

D. Differentiation Based PSC Detection

In this section, we explain how the *first order partial difference* (FOPD) of the access functions in a group of operations can be used to detect a codelet type.

First Order Partial Differentiation ($FOPD_{\mathcal{I}}$). Given the data access function f with an iteration space of $\mathcal{I} = [i_0, i_1]$, the first order partial difference of f with respect to $i_1 \in \mathcal{I}$ is computed as $FOPD_{i_1} = \frac{\Delta}{\Delta i_1} f(\mathcal{I}) = \Delta_{i_1} f = f(i_0, i_1 + 1) - f(i_0, i_1)$. FOPD shows if accesses to a data space are strided with respect to the induction variable i_1. Figure 5 illustrates the process of computing the FOPD for the access function h given the operation group shown in Figure 3a. For example, the FOPD of h evaluated at $i_0 = 1, i_1 = 1$ is $\Delta_{i_1} h(1, 1) = 3$.

FOPD of access functions are used to distinguish types of codelets by finding strided access functions. Given a codelet with three access functions and the iteration space of $\mathcal{I} = [i_0, i_1, nrows]$, an access function f is strided if its FOPDs with respect to $\mathcal{I}$ are equal in the entire iteration space. In other words, f is strided if the elements in $\Delta_{i_0} f(i_0, i_1)$ are equal to each other and similarly for elements in $\Delta_{i_1} f(i_0, i_1)$. With the strided definition above, all codelet types can be defined per definitions in Section II-B. For example, the first three accesses in function $g(i_0, i_1)$ in Figure 5 are to consecutive locations (0, 1, 2) in `Ax`. The FOPD of the first two accesses, wrt. i_1, is $\text{FOPD}_{i_1}(0, 0) : g(0, 1) - g(0, 0) = 1$ and for the second and third accesses is $\text{FOPD}_{i_1}(0, 1) : g(0, 2) - g(0, 1) = 1$. Similarly the FOPD of the accesses for i_0 are $\text{FOPD}_{i_0}(0, 0) : g(1, 0) - g(0, 0) = 4$ and $\text{FOPD}_{i_0}(1, 0) : g(2, 0) - g(1, 0) = 4$. Since $\text{FOPD}_{i_1}(0, 0)$ and $\text{FOPD}_{i_1}(0, 1)$ are equal, and $\text{FOPD}_{i_0}(0, 0)$ and $\text{FOPD}_{i_0}(1, 0)$ are equal, the function in the iteration space $\mathcal{I} = \{i_0 = 0 \wedge 0 \leq i_1 \leq 3\}$ is strided. Because function f is also strided and h is not strided, the codelet is categorized as a PSC type I.

```
#include <immintrin.h>
void PSCI_MADD(double *Ax, double *x, double *y,
    IterSpace I, Fun f, Fun g, UFun h){
 for(int i0=0; i0<I.n0; i0++){
  auto xt = gather(x, h.s0);
  auto r0 = _mm_setzero_pd();
  for(int i1=0; i1<I.n1; i1+=V){
   Axt = _mm_loadu_pd(Ax, g(i0,i1));
   r0 = _mm_fmadd_pd(Axt, xt, r0);
  }
  y[f(i0)] += hsum_double_avx(r0);
 }
}
```

Listing 1: The parametrized vectorized routine for a PSC-I codelet with the multiply-add operation and with unstrided h.

Op #	$[i_0, i_1]$	f	g	h
0	0 0	y[0] += Ax[0] * x[4]		
1	0 1	y[0] += Ax[3] * x[3]		
2	0 2	y[0] += Ax[4] * x[2]		
3	1 0	y[1] += Ax[5] * y[0]		
4	1 1	y[1] += Ax[6] * y[0]		

(a) access functions and iteration space

$\mathcal{G}$ (i ↓, j →)

i \ j	0	1	2	3	4
0	1	1	1	0	0
1	1	1	1	0	0
2	1	1	1	0	0
3	0	0	0	1	1
4	0	0	0	1	1

(b) dependence information

$final\_list = (C_0, C_1)$

$C_0 = (0, 1, 2)$

$C_1 = (3, 4)$

(c) codelet list

Fig. 6: The inputs (Figures 6a and 6b) and output (Figure 6c) of the PSC mining problem.

E. Parameterized Vectorized Routine

We vectorize a PSC with a parameterized vectorized routine that efficiently uses the SIMD instructions of the target architecture, i.e. x86. The parameterization also allows us to generate concise code invariant to the number of codelets that are mined for a set of operations. The parameterized vectorized routine takes the iteration space and access functions of a codelet and data spaces of the kernel as input and vectorizes all operations in the codelet. Listing 1 shows the parameterized vectorized routine for a PSC-I and the multiply-add operation with an unstrided access function h. For efficient use of SIMD instructions, we implement a separate routine based on strided properties of access functions. For a PSC I codelet, we implement three routines, and in each routine one of f, g, h is unstrided. For PSC II, we use three routines with one of f, g, or h being strided per routine. To vectorize a list of codelets of different types, a switch-case structure is used that selects the parameterized vectorized routine associated with each codelet type.

III. Mining for Partially Strided Codelets

PSC mining creates a list of codelets from access functions and the iteration space of a sparse kernel with the objective to minimize the overall cost of the final codelet list. In this section we first define PSC mining as a problem with its objective and constraints and then propose a locality-based codelet mining (LCM) heuristic to solve it. The extension of LCM to multi-thread parallelism is also discussed.

A. The PSC Mining Problem

This subsection first defines the inputs and output of the PSC mining problem and then demonstrates the objective and its constraints. The inputs to the PSC mining problem are a list of P unique operations represented with three access functions f, g, and h, the iteration domain $\mathcal{I}$, and operation dependence information $\mathcal{G}$. The matrix $\mathcal{G}$ is boolean, and $\mathcal{G}_{ij} = 1$ indicates that executing operation j after operation i does not violate correctness in the sparse computation, and is obtained from the dependence graph of the sparse kernel [1]; The range for i and j is $[0, P-1]$. The output is the list of M mutually disjoint codelets, $final\_list = (C_0, C_1, ..., C_{M-1})$ that covers all P operations in the sparse computation. The objective of the PSC mining algorithm is to find the best list of codelets that covers each operation exactly once and minimizes the below cost:

$$\sum_{k=0}^{M-1} cost(C_k) \tag{2}$$

To ensure correctness for codelets with more than one operation, the following constraint should be satisfied:

$$\forall k = 0 \dots M-1 \quad \forall i, j \in C_k, i \neq j, \quad \mathcal{G}_{ij} = \mathcal{G}_{ji} = 1 \tag{3}$$

The constraint ensures that operations i and j are assigned to a codelet k only if they can execute independently, i.e. $\mathcal{G}_{ij}$ and $\mathcal{G}_{ji}$ is one. Additionally the PSC mining problem ensures that an operation is only mapped to one codelet and is used in that codelet only once. It is possible that a codelet gets assigned only one operation and because in those codelets no dependence is violated $\mathcal{G}_{ij}$ for $i = j$ is always 1.

An example of input operation list to PSC mining is shown in Figure 6a. The dependence information matrix $\mathcal{G}$ for the five operations in Figure 6a is shown in Figure 6b, e.g. $\mathcal{G}_{34}$ and $\mathcal{G}_{43}$ is set to one because operations 3 and 4 are independent. Note that these operations are independent because we are leveraging the associativity property of the add operation. $\mathcal{G}_{30}$, $\mathcal{G}_{31}$, and $\mathcal{G}_{3,2}$ are zero because there is read after write dependence between operation 3 and operations 0–2. The constraint in Equation 3 is satisfied in the final codelet list in Figure 6c, e.g. operations 3 and 4 that are mapped to codelet 1 are independent based on $\mathcal{G}$ while operations 3 and 0 are not mapped to the same codelet.

The solution to the discussed PSC mining problem is NP-hard as it is equivalent to solving a set partitioning problem [12]. The objective of a set partitioning problem is to create non-overlapping subsets that cover all elements in the set and minimizes a total cost over all subsets. Similarly, PSC mining creates a list of codelets from the list of operations in the sparse kernel such that codelets do not overlap and cover all operations in the kernel.

B. Locality-based Codelet Mining Heuristic

We propose a locality-based mining algorithm that mines operations and finds an efficient codelet combination that satisfies the constraints of the PSC mining problem and minimizes the total cost of codelets. The LCM algorithm has two steps; in the first step, LCM permutes operations to increase data reuse and the possibility of creating efficient codelets from

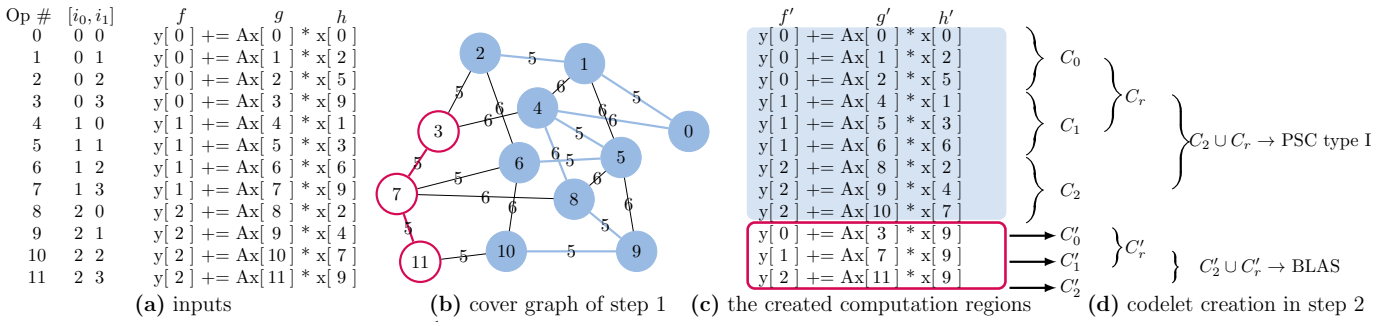

Fig. 7: Figure 7a shows the LCM inputs, i.e. the list of operations, their iteration space, and their access functions that correspond to SpMV operations of the blue part of the matrix in Figure 1a. Figure 7b shows the minimum edge cover problem that first step of LCM solves. Each vertex corresponds to an operation and an undirected edge between operations i and j shows that they can execute in any order without violating correctness while ensuring a strided access. Figure 7c shows the permuted operations corresponding to highlighted edges in Figure 7b. Figure 7d shows the second step of LCM where, for each computation region, it merges initially created small codelets to reduce the total cost of codelets.

that list of operations. This step also groups operations that have the most reuse into *computation regions*. The second step of LCM searches for efficient codelets in consecutive operations of each computation region. The steps in LCM prune the search space of the PSC mining problem by finding codelets in computation regions with high data reuse, i.e. good locality.

1) Step 1, Permutation and Partitioning: In the first step, LCM permutes operations to increase locality. It also creates partitions from the permuted operations. Since locality results from operations with 0/1 strided memory accesses, LCM looks into the access functions for operations that access spatially close or common memory locations. This is obtained via solving a permutation problem that improves locality through minimizing a *distance* between operation pairs while ensuring the correctness of the computation. Only operation pairs that have at least one strided access are considered as they provide better data reuse. Distance between operations is defined as a measure for common memory accesses and the number of strided accesses between a pair of operations. For a pair of operations i and j, distance is $d_{ij} = (|f(OP[i]) - f(OP[j])| + |g(OP[i]) - g(OP[j])| + |h(OP[i]) - h(OP[j])|) \times u^2$ where $|.|$ is the absolute difference between operations i and j when the difference is either zero or one, otherwise it is a constant, e.g. 4, and u is the number of operation pair accesses for these two operations that do not have 0/1 strided access. OP is the list of integer tuples $(\mathcal{I}_0, ..., \mathcal{I}_{|p|})$ where $OP[j]$ shows the iteration information of operation j, e.g. in Figure 7a $OP[2] = (0, 2)$.

We solve the permutation problem by modeling it as a minimum edge covering graph problem[1]. Vertices are also partitioned as a part of the process. A *cover graph* is first created. In the cover graph each vertex represents an operation and an edge exists between two vertices if their operations do not have RAW dependence and have at least one 0/1 strided access. Each edge in the graph has a weight equal to the distance between the two operations connected via the edge. The selected edge set from the graph problem is used to permute operations. If e_{ij} is in the set, then operation j will follow i in the permuted list of operations. In an additional step, we iterate over the graph with the selected edges and, group vertices that belong to a connected component as a computation region. Figure 7b shows the cover graph representation created for the list of operations in Figure 7a. The graph has 12 vertices that correspond to the SpMV operations in the blue region of the matrix in Figure 1a. Operations 0 and 1 have 0/1 stride in f and g thus, an edge with weight 5 connects them. For brevity, only edges for operations with 0/1 strided accesses in at least two access functions are shown. The graph is also leveraging the associativity property in the add operation in SpMV and hence for example there exists no edge between operations (vertices) 0 and 1. The output of Step 1 is shown with colored edges. Selected edges between operations 3 and 7 indicate that operation 7 should execute after operation 3 the direction is from the operation with small number to the big one. Vertices covered by edges of the same color belong to one computation region. Two computation regions are created in the graph in Figure 7b.

Since solving the minimum edge cover problem on the entire graph is NP-hard [14], LCM uses a greedy algorithm and finds the best possible edge cover amongst vertices visited through a depth-first search. It initially creates a computation region with a *window* of vertices. A window contains a chain of vertices with the size of vectorization factor that all their connecting edges have the same edge weight d_i. Windows w_1 and w_2 are strided if every vertex in w_1 connects to a unique vertex in w_2 via a common weight d_o. From the initial window, LCM does a depth-first search to find its strided windows. The vertices of a window and its strided window vertices are put into the same computation region. The process repeats for any unvisited vertex in the graph.

The permutation step in LCM is shown in Lines 1–12 in Algorithm 1. The graph is created via the *make_cover_graph* function in line 2. To create the edges of the cover graph, obtained by the distance, *make_cover_graph* checks operation

[1]Minimum edge covering problem [13] finds the subset of edges that minimizes total edge cost; the union of edge endpoints covers all vertices.

Algorithm 1: Locality-based Codelet Mining (LCM)

```
   Input  : f, g, h, I, G
   Output: final_list
   /* 1) Permutation and Partitioning                    */
1  comp_region_list ← ∅;
2  CGraph ← make_cover_graph(f, g, h, G);
3  for unvisited v1, v2, v3 ∈ CGraph do
4      stack.push( v1, v2, v3);
5      tmp_cr ← ∅;
6      di, do ← determine_search_window(v1, v2, v3);
7      while stack is not empty do
8          v1', v2', v3' ← stack.pop_three_ops();
9          mark_as_visited(v1', v2', v3');
10         tmp_cr.append(v1', v2', v3');
11         stack.push( expand_window(v1', v2', v3', di, do));
12     comp_region_list.append(tmp_cr);
   /* 2) Codelet Creation                                */
13 for r ∈ comp_region_list do
14     FOPDr ← compute_FOPD(r);
15     S ← create_PSC_II(FOPDr, I);
16     for s ∈ S do
17         Cr ← ∅;
18         while cost(s ∪ Cr) ≤ cost(Cr) + cost(s) do
19             Cr ← s ∪ Cr;
20             s ← S.next();
21         final_list.append(Cr);
```

Algorithm 2: Multi-threaded LCM (MT-LCM)

```
   Input  : f, g, h, I, G, K
   Output: Schedule
1  if G.has_zero() then
2      Partitions ← wavefront_coarsening(G, I, 8 × K);
3  else
4      Partitions ← partition_even(I, 8 × K);
5  pList ← ∅;
6  for I' ∈ Partitions do
7      f', g', h', G' ← get_func_of_iteration_space(f, g, h, I');
8      L' ← LCM(f', g', h', I', G');
9      C' ← Cost(L');
10     pList.append(L', C');
11 Schedule = first_fit_packing(pList, K);
```

pairs. Because checking all operation pairs is expensive, *make_cover_graph* picks pairs from V consecutive operations. V is the vectorization factor. This strategy is effective because operations are initially sorted based on access functions with the most reuse, i.e. g in SpMV and SpTRSV and h in SpMM. The initial window and common weights d_i and d_o are determined via $determine\_search\_window$ in Line 6 in Algorithm 1. Depth first search is conducted in lines 7–11 and function $expand\_window$ in Line 11 returns the strided windows v'_1, v'_2, and v'_3 w.r.t d_i and d_o. When all reachable vertices are visited, i.e. the stack is empty, the current computation region is added the list of regions in Line 12 and the process repeats to create another computation region. Operations in a window are placed consecutively in the $comp\_region\_list$ list to ensure efficient use of SIMD units. During this process each vertex is visited twice and each edge is visited once thus the computation complexity of step 1 is $O(2 \times P + E)$ where E is the number of edges.

For the graph in Figure 7b and a vectorization factor of three, a computation region is initialized with a window {0,1,2}. The region is expanded with a strided window {4,5,6} where $d_i = 5$ and $d_o = 6$. This region is enlarged by adding the strided window {8,9,10}. The final computation region will be of size 9, and then the algorithm repeats to create the other computation region containing {3,7,11}.

2) Step 2, Codelet Creation: In the second step, for each computation region, LCM fits the best combination of PSC/BLAS possible. As discussed in Section II-C, different codelet combinations can be created for a computation region with a given order of operations. For example, for the blue computation region in Figure 7c, one possibility is to create three BLAS codelets as shown in Figure 1c with an indexing cost of 15, another possibility is to create three PSCs-II with the cost of 18. LCM chooses the codelet set with the minimum cost of 7, which is the PSC-I shown in Figure 3 (arithmetic operation cost and loading/storing operand cost are the same for all three possibilities).

To solve the codelet creation problem, LCM first creates a list of PSC-II codelets (S) that covers all operations in a computation region, ($create\_PSC\_II$ in Line 15). Then the codelets in S will be visited in order and merged to reduce the total cost of the output, $final\_list$. The codelet list $final\_list$ is created to minimize Equation 2 while prioritizing writes over reads. This is because write accesses are more costly compared to reads. The algorithm performs this by grouping all operations that the FOPD of their write access function is constant and zero. To create the final codelet list, in Lines 16–21, LCM applies a merging method based on codelet costs. The first codelet in S is put in a C_r (running codelet) and merged with the following codelets in S, if profitable. Merging two codelets is profitable if the PSC that covers the operations of both codelets, i.e. $cost(s \cup C_r)$, has a cost that is lower than the cost of individual codelets added. If profitable, the algorithm merges C_r with s and checks for the possibility of merging with the next codelet in S. The algorithm also stops merging when the size of the codelet becomes larger than the L1 cache to satisfy the cost model (Section II-C) requirement. If not profitable, the codelet in C_r is put in to the $final\_list$. The process continues until the $final\_list$ is populated with codelets that cover all operations in the kernel, i.e. set S is fully visited. Since each operation is visited once in Step 2, its complexity is $O(P)$.

C. Multi-threaded Extension

LCM takes a set of operations and creates codelets to vectorize computations on a single-core. To extend the algorithm to multiple threads and hence improve scalability, in Algorithm 2, we show the extension of LCM to multiple threads (called MT-LCM). The number of cores K is also an input to MT-LCM. Lines 1–4 create balanced partitions to be executed by each thread and simultaneously minimize synchronization between

threads. LCM is then applied to each partition in lines 6–10. Because the execution time of operations in partitions changes after vectorization, the cost of each partition is also computed in line 9, and in the final stage, partitions are balanced using bin packing to create well-balanced partitions.

MT-LCM initially partitions the operations into M independent partitions in Lines 1–4 in Algorithm 2. It selects M to be larger than K, i.e. $8 \times K$, so that the computations can be balanced via merging partitions. Kernels SpMV and SpMM only contain operations that have write-after-write dependence. Thus, the outermost iterations are evenly divided into partitions. In SpMM, iterations that access the same column can potentially improve temporal reuse. Hence, MT-LCM merges partitions with more than 50% common column access. For kernels that have loop-carried dependencies, such as SpTRSV, its dependence information has RAW dependence and hence at least one zero in $\mathcal{G}$ (Line 1). To create partitions, we use the load-balanced wavefront coarsening (LBC) [15] method to partition operations (line 2 in Algorithm 2). A wavefront is a group of iterations with no RAW. LBC finds independent partitions from a group of wavefronts called coarsened wavefronts and then applies synchronization between the coarsened wavefronts.

MT-LCM uses the PSC cost model along with a bin-packing method to ensure load balance after operations are vectorized. As shown in Lines 6–11 in Algorithm 2, MT-LCM computes access functions of each partition in line 7 and then builds codelets from the functions (Line 8). The partition cost, equal to the total cost of codelets in the partition, is calculated in Line 9. Finally, in Line 11, the vectorized partitions and their costs are passed to a first-fit bin-packing method [16], where it merges every pair of consecutive partitions to create a larger partition equal to a target cost. The target cost is the total cost of all partitions divided by K.

IV. Results

We evaluate the performance of LCM (implemented as an inspector-executor, which we call LCM I/E) using three kernels, SpMV, SpTRSV, and SpMM. LCM I/E is compared to other automation approaches, specifically Sympiler and SPF, which are the tiling-based and the regular piece-wise methods from [9]. We also compare LCM I/E to libraries CSR5 [6] and MKL [17]. Sympiler does not support SpMV and SpMM, and CSR5 and regular piece-wise implementations do not support SpTRSV and SpMM, so these tools are omitted from the respective kernel figures.

A. Experimental Setup

Matrix Dataset: The set of matrices used to evaluate the performance of SpMV and SpMM are obtained from the SuiteSparse repository [11]. For an unbiased selection of matrices, we choose all real matrices with more than 100K non-zero elements (789 total). To ensure the numerical stability of SpTRSV, all symmetric positive definite (SPD) matrices larger than 100K nonzeros (132 total) are selected from SuiteSparse for evaluation. For the SpTRSV, we only use the lower triangular half of the SPD matrices. Since profiling is time consuming, all of our profiling figures and data are conducted on a randomly selected set of 30 matrices in the range of 100k-100 million nonzeros. Throughout the result section, we refer to these 30 matrices as the random subset. Additionally for reproducibility, we separately report

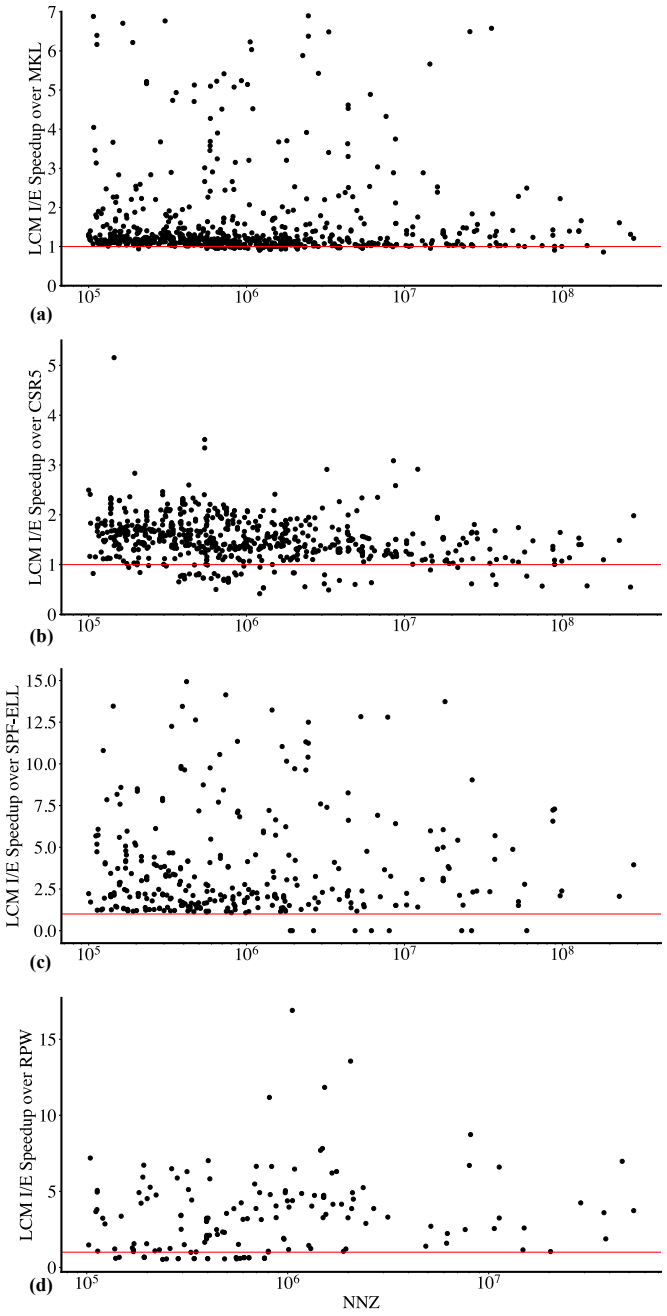

Fig. 8: LCM I/E speedups for SpMV over (a) the best of the multi-threaded SpMV CSR and the parallel MKL, (b) parallel CSR5, (c) SPF-ELL, and (d) Regular piece-wise (RPW). Every point above the red horizontal line represents a matrix where LCM I/E is faster. RPW and SPF-ELL are missing data points because they either time out or become out of memory.

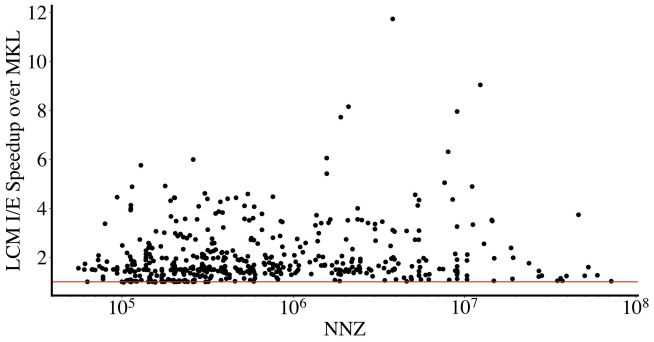

Fig. 9: The LCM I/E speedups for SpMV over the best of the multi-threaded SpMV CSR and MKL SpMV on AMD.

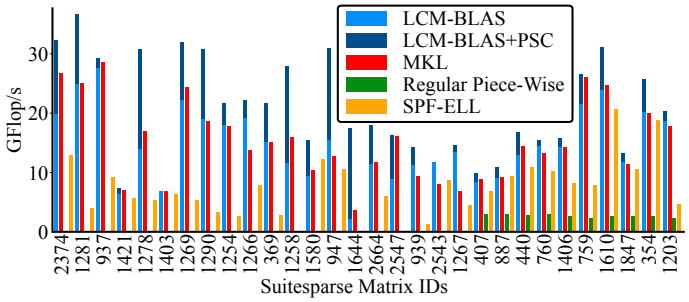

Fig. 10: The performance breakdown of LCM I/E for SpMV using BLAS and PSCs, and its comparison with SPF-ELL and RPW are shown.

the running time in GFlop/s of all tools (as a stacked bar) for these 30 matrices. The test-bed architectures are an Intel(R) Xeon(R) Gold 5115 CPU (2.8GHz, 14080K L3 Cache) with 20 cores and 64GB of main memory and an AMD EPYC 7742 CPU (2.25 GHz, 256MB L3 Cache) with 68 cores and 256GB of main memory. All experiments are conducted on the Intel processor unless otherwise stated. All generated code, implementations of different approaches, and library drivers are compiled with GCC v.11.2 and the `-O3` flag. Each benchmark is executed 50 times per matrix/kernel, and the median value of the runs is reported. MKL 2021.4.0 and latest public version of CSR5 are used for evaluation.

LCM implementation: We implement LCM and the PSC codelets as an inspector-executor framework, called *LCM I/E*[2]. The inspector first creates a set of codelets using the LCM algorithm, and then the executor executes the kernel with the created codelets. With its inspector, LCM I/E creates access functions and the dependence graph and passes them to the MT-LCM algorithm along with the number of cores, which is 20 for the Intel processor. The access functions are created from the matrix sparsity pattern and the kernel code i.e., SpMV, SpTRSV, and SpMM codes in compressed sparse row (CSR) format. For each sparse kernel, LCM I/E generates an inspector and an executor and hence performs code generation and compilation once per kernel. We show the executor time of LCM I/E in all graphs unless otherwise stated.

Regular piece-wise (RPW) methods: To compare LCM with the RPW approaches, we use the work in [9] which is evaluated for SpMV. The code for RPW is not publicly available, thus, we created an in-house inspector-executor implementation of their approach with feedback from the authors. Since RPW methods perform code generation and compilation per matrix, we include this timing as a part of their inspector. The RPW method in [9] supports single-threaded execution, so we also extended it for parallelism, and report the best performance between the two implementations. A timeout of 4 hours was used for all runs (including the code generation, inspection, and execution time). In our figures, RPW does not have a data point for some large matrices because their inspector times out.

Tiling-based approaches: We compare the SpTRSV and SpMV of LCM I/E in order with Sympiler and an implementation of the sparse polyhedral framework. For Sympiler, we use its generated code for the SpTRSV kernel. To compare with SPF, since the code is not publicly available, we use their SpMV implementation from Venkat *et. al.* [19] and use the techniques proposed in ELLPACK [2] to decide when to pad; we call this ELLPACK-based implementation of SPF the SPF-ELL approach. In our figures, SPF-ELL does not have a data point for large matrices, because of padding from ELL, the size of these matrices becomes larger than the system memory.

B. SpMV Performance

We compare the performance of LCM I/E for SpMV with CSR5, MKL, SPF-ELL, and RPW across all matrices in the dataset as shown in Figure 8. LCM I/E is faster than all four methods for over 91% of the matrices. It is on average 1.6×, 2.1×, 4.1×, and 4.6× faster than MKL, CSR5, SPF-ELL, and RPW respectively. We also compare LCM I/E with its fastest competitor, i.e., MKL for SpMV on the AMD processor as shown in Figure 9. LCM I/E is faster than MKL across all matrices in our dataset with an average speedup of 1.7×.

To demonstrate the effect of partially strided codelets on the performance of LCM I/E, in Figure 10, we use a stacked bar for LCM I/E for the random subset of matrices. The stacks show the GFlop/s from LCM I/E when it only mines for BLAS codelets (LCM I/E+BLAS and refer to LCM I/E BLAS_only), and from the entire LCM I/E algorithm, i.e. Algorithm 1, that mines for BLAS and PSC (shown with LCM I/E or LCM I/E+BLAS+PSC in the figure). As shown, LCM I/E is on average 10× faster than LCM I/E BLAS_only, which demonstrates the importance of using PSC codelets. We also calculate the percentage of operations that are vectorized in the generated code from LCM I/E with PSC I, PSC II, and BLAS codelets, obtained by averaging over all matrices in the random subset. We found that 83% of operations in SpMV are vectorized with PSC codelets.

The PSC codelets improve strided memory accesses through improving data locality in LCM I/E's generated code. Figure 11a shows the relation between LCM I/E's performance and

[2]The code is publicly available from [18]

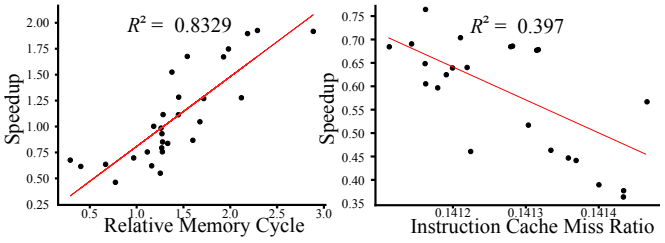

Fig. 11: Figure 11a shows the correlation between LCM I/E speedup over the parallel SpMV CSR and its relative memory cycle to the parallel SpMV. The coefficient of determination or R^2 is 0.83. Figure 11b shows the correlation between instruction cache misses in our implementation of RPW compared to the sequential SpMV CSR where R^2 is 0.4.

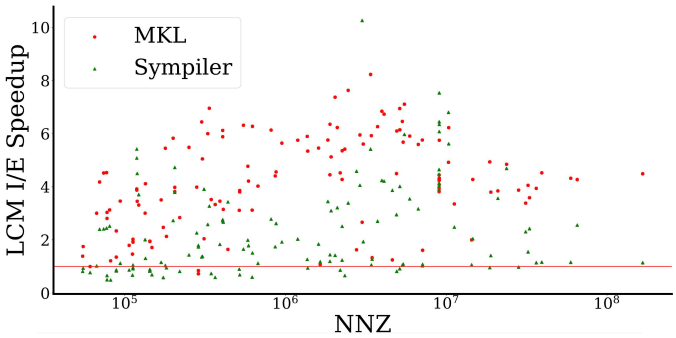

Fig. 12: The LCM I/E speedups over parallel MKL (red circles) and Sympiler (green triangles) for SpTRSV.

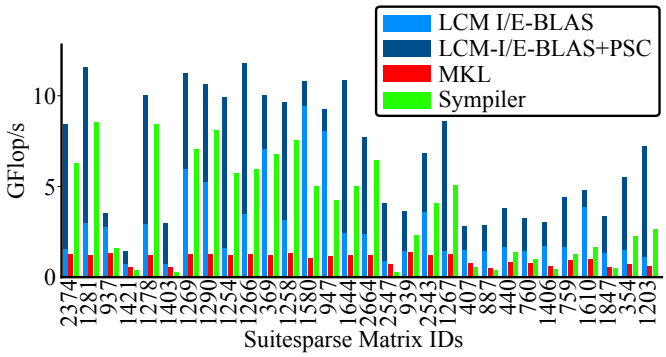

Fig. 13: Performance breakdown of LCM I/E for SpTRSV using BLAS and PSCs, and its comparison with MKL and Sympiler.

the performance of the parallel (OpenMP) version of SpMV CSR for the random subset of matrices. Average memory access latency [20] is used as a measure for locality and is computed by gathering the number of misses and accesses to L1, L2, and LLC caches using the PAPI [21] performance counters. Figure 11a shows the coefficient of determination or R^2 is 0.83, which indicates a good correlation between speedup and the memory access latency.

To further explain why LCM I/E is faster than other tools, we conduct a few experiments and report the resulting average over the random subset of matrices in this paragraph. For MKL, we compute its average memory access latency, which is 4.36x slower than that of LCM I/E, contributing to the worse performance compared to LCM I/E. To compare to CSR5, we count the number of instructions. CSR5 executes 1.73x more instructions compared to LCM I/E. This is potentially due to the overhead of the segmented sum calculations used in their approach to improving vectorization and load balance. To conclude, the SpMV code of LCM I/E is faster than existing implementations because it improves data locality and/or reduces the number of instructions via vectorization.

We compare LCM with the automation approaches, SPF-ELL, and RPW for 30 selected random matrices. Figure 10 compares the performance of LCM I/E with RPW and SPF-ELL. As shown, LCM I/E is faster than RPW and SPF-ELL with an average of 6.32× and 5× respectively. The RPW approach has difficulties in scaling because the size of its output code is often linearly correlated with the number of nonzero elements of the sparse matrix, and thus the performance of its generated code depends on the number of instruction cache misses. Figure 11b illustrates a negative correlation between the regular piece-wise speedup over the sequential baseline SpMV code and the relative instruction cache misses, with $R^2 = 0.397$. Code compaction techniques such as adding loop variables to group small BLAS codelets do not help on large scale due to conditional statement overhead. LCM does not have a code size issue because of using the proposed PSC codelet classification and their parameterized vectorized routines. SPF-ELL is slower than LCM I/E because the number of padded nonzeros in SPF-ELL is often larger than the nonzero elements in the matrix, thus creating significant redundant instructions.

C. SpTRSV Performance

Figure 12 compares the performance of SpTRSV using LCM I/E, MKL, and Sympiler across all SPD matrices in the dataset. On average LCM I/E is faster than MKL, and Sympiler 4.1× and 1.92×, respectively.

We show the performance breakdown of LCM using BLAS and PSCs in Figure 13 for the random subset. As shown, partially strided codelets are the main contributors to the overall performance of the LCM I/E's SpTRSV code. On average, LCM I/E is 3.9× faster compared to when only BLAS codelets are generated. Similar to SpMV, PSCs contribute to optimizing 78% of the operations over the 30 matrices. However, the number of PSC II codelets has increased from 18% in SpMV to 53% in SpTRSV. The number of codelets with more than one strided access function is small, due to the existing dependencies in SpTRSV, thus, more PSC type II codelets are generated. Similar to SpMV, the mined PSCs in LCM I/E improve locality. The correlation coefficient between the speedup and relative memory cycle is 0.67, which is consistent with the trend in SpMV.

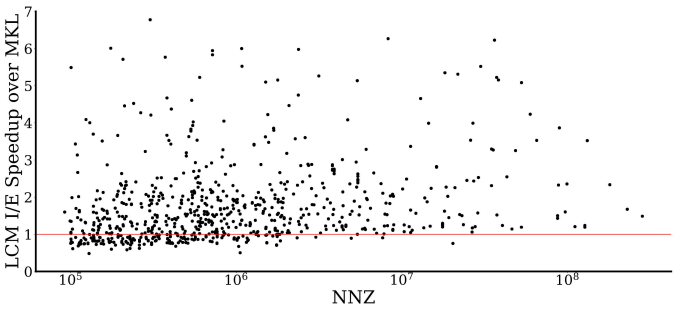

Fig. 14: The LCM I/E speedups for SpMM over the best of the multi-threaded SpMM CSR and MKL SpMM.

The LCM I/E code for SpTRSV is faster than Sympiler and MKL due to multiple factors. While MKL provides an efficient and vectorized implementation for single-threaded SpTRSV executions, it is not optimized to execute on parallel processors; the performance of MKL's parallel code is similar to its serial implementation. Sympiler performs well for matrices that contain row-blocks or can be padded with up to 30% nonzeros to create row-blocks, these row-blocks are converted to BLAS calls and thus improve locality. However, it does not perform well for other matrices.

D. SpMM Performance

The scatter plot for sparse matrix-dense matrix multiplication (SpMM) speedups is shown in Figure 14 where the dense matrix in SpMM has 256 columns. The average speedup for the dataset is 1.75× over the best of MKL and a parallel implementation of SpMM in CSR. LCM I/E is faster than MKL for 87% of matrices up to 6.41×. We also change the number of columns in the dense matrix in SpMM to be 32, 64, and 128 and observe that LCM I/E's average speedup over MKL is 1.23×, 1.54×, and 1.72× respectively for these number of columns. When the number of columns increases in the dense matrix, temporal reuse between operations increases. LCM I/E turns the temporal reuse across iterations into locality and thus outperforms MKL.

E. The Inspector Overhead

We compare the inspection time of LCM I/E to that of other inspector-executors frameworks, i.e. Sympiler and SPF-ELL. We compute the number of executor runs (NER) that amortize the cost of the inspector using $\frac{Inspector\ Time}{Baseline\,Time - \ Executor\ Time}$. The *baseline* time is obtained by running a sequential implementation of the kernel. The RPW inspector would timeout for over 83.3% of matrices because of its large inspection overhead and code compilation time. For small matrices that RPW would execute, more than one million executor runs are needed to amortize the cost of the inspection. LCM I/E's inspection time is on average 0.5 seconds with an average NER of 15 for SpMV and the NER for SPF-ELL is on average 85. The inspection time of LCM I/E and Sympiler are similar, with an average NER of less than 100 for both tools for SpTRSV. The largest matrices in our benchmark are inspected in less than 7 seconds with LCM I/E. Sparse kernels such as SpMV and SpTRSV are typically used in iterative solvers, for example, to compute a residual in each iteration or to apply a preconditioner per iteration. Even with preconditioning, these solvers typically converge to a solution after tens of thousands of iterations [22], [23], [24] and hence inspector-executor frameworks such as LCM I/E and Sympiler lead to noticeable speedups as their inspection time overheads are amortized after a few initial iterations of the solver.

V. Related Work

Numerous hand-optimized libraries [25], [26] and implementations [27], [28], [29], [30] exist that optimize the performance of sparse matrix computations for different parallel architectures and also optimize vectorization on a single core. Libraries such as MKL and Eigen as well as implementations in [31], [32], [33], [34], [35] optimize the performance of SpMV on shared memory architectures and improve SIMD vectorizability. A number of library implementations such as [2], [36], [37], [38], [6], [39], [5], [40], [41] reorganize data and computation to increase opportunities for vectorization. A class of these libraries implement and optimize sparse kernels based on available storage formats; for example [42] optimizes SpMV based on ELLPACK. Other libraries work best for matrices arising from specific applications, such as [31], which optimizes SpMV for large matrices from graph analytics or KLU [43] which works best for circuit simulation problems.

Inspector-Executor approaches inspect the unstrided access patterns of sparse matrix computations at run-time to enable the automatic optimization of sparse codes [44], [45], [46], [47], [19], [48], [49], [50], [51]. The index array accesses of the sparse code is analyzed using an inspector and the information is used at runtime to execute the code efficiently. The sparse polyhedral framework [52], [53], [54], [55] uses uninterpreted function symbols to express regular and irregular segments of sparse codes. As a result, it can automatically generate inspector executors at compile time that can resolve data dependencies in sparse computations. These approaches do not generate code that is specialized for the sparsity of the input matrix. Sympiler [1], ParSy [56], and HDagg [57] are amongst the inspector executor frameworks that inspect the matrix sparsity pattern and as a result generate vectorized and parallel code specialized for the input sparsity. Their optimizations for vectorization are based on detecting tiles or row-blocks that primarily exist in matrices obtained from the numerical factorization.

Augustine et al. [9] proposed an approach based on the Trace Reconstruction Engine [58], [59] where polyhedral models are built by inspecting the sequence of addresses being accessed in the sparse matrix-vector multiplication. A follow-up to this work proposes to use program-guided optimization for better vectorization [60]. These approaches lead to generating code that is specialized for the sparsity pattern of the input matrix and improves vectorization in SpMV for strided regions. Their work only supports matrices below 10M

nonzeros because of inspector overheads and because the size of the generated code increases with the matrix size.

VI. Conclusion

In this work, we mine for partially strided codelets to enable the efficient vectorization of computations with strided and partially strided memory accesses in sparse matrix codes. We demonstrate how these codelets increase opportunities for vectorization in sparse codes and also improve data locality in their computation. A novel algorithm called locality-based codelet mining is proposed that efficiently mines for PSCs and as a result, generates highly efficient code for sparse kernels. The performance of the LCM I/E-generated code is compared to state-of-the-art library implementations and automation frameworks for three sparse matrix kernels.

Acknowledgment

We would like to thank the Digital Research Alliance of Canada and Scinet[3] for providing us with critical resources to conduct this research. This work was also supported in part by NSERC Discovery Grants (RGPIN-06516, DGECR00303), the Canada Research Chairs program, Ontario Early Researcher award, the Government of Canada's New Frontiers in Research Fund NFRFE-2019-00503, along with the Extreme Science and Engineering Discovery Environment (XSEDE) [61] which is supported by NSF grant number ACI-1548562.

References

[1] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi, "Sympiler: Transforming sparse matrix codes by decoupling symbolic analysis," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017*. New York, NY, USA: Association for Computing Machinery, Inc, nov 2017, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/3126908.3126936

[2] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[3] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 071.

[4] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[5] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "Cvr: Efficient vectorization of spmv on x86 processors," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 149–162.

[6] W. Liu and B. Vinter, "CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication," *Proceedings of the International Conference on Supercomputing*, vol. 2015-June, pp. 339–350, mar 2015. [Online]. Available: http://arxiv.org/abs/1503.05032

[7] R. Zheng and S. Pai, "Efficient execution of graph algorithms on cpu with SIMD extensions," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 262–276.

[8] M. M. Strout, M. Hall, and C. Olschanowsky, "The sparse polyhedral framework: Composing compiler-generated inspector-executor code," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1921–1934, 2018.

[3] www.alliancecan.ca

[9] T. Augustine, L. N. Pouchet, J. Sarma, and G. Rodríguez, "Generating piecewise-regular code from irregular structures," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, NY, USA: Association for Computing Machinery, jun 2019, pp. 625–639. [Online]. Available: https://dl.acm.org/doi/10.1145/3314221.3314615

[10] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen, "Polyhedral-model guided loop-nest auto-vectorization," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2009, pp. 327–337.

[11] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[12] R. S. Garfinkel and G. L. Nemhauser, "The set-partitioning problem: set covering with equality constraints," *Operations Research*, vol. 17, no. 5, pp. 848–856, 1969.

[13] R. Z. Norman and M. O. Rabin, "An algorithm for a minimum cover of a graph," *Proceedings of the American Mathematical Society*, vol. 10, no. 2, pp. 315–319, 1959.

[14] H. Fallah, A. N. Sadigh, and M. Aslanzadeh, "Covering problem," in *Facility Location*. Springer, 2009, pp. 145–176.

[15] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi, "Parsy: Inspection and transformation of sparse matrix computations for parallelism," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 779–793.

[16] G. Dósa and J. Sgall, "First fit bin packing: A tight analysis," in *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[17] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, *Intel Math Kernel Library*, 05 2014, pp. 167–188.

[18] Cheshmi, Kazem and Cetinic, Zachary and Dehnavi, Maryam Mehri, "Locality-based Codelet Mining (LCM)." [Online]. Available: https://zenodo.org/badge/latestdoi/465357427

[19] A. Venkat, M. Hall, and M. Strout, "Loop and data transformations for sparse matrix code," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, vol. 2015-June. New York, NY, USA: Association for Computing Machinery, jun 2015, pp. 521–532. [Online]. Available: https://dl.acm.org/doi/10.1145/2737924.2738003

[20] J. Hennessy and D. Patterson, *Computer Architecture - A Quantitative Approach*, 01 2007.

[21] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[22] K. Cheshmi, D. M. Kaufman, S. Kamil, and M. M. Dehnavi, "NASOQ: numerically accurate sparsity-oriented qp solver," *ACM Transactions on Graphics (TOG)*, vol. 39, no. 4, pp. 96–1, 2020.

[23] M. Benzi, J. K. Cullum, and M. Tuma, "Robust approximate inverse preconditioning for the conjugate gradient method," *SIAM Journal on Scientific Computing*, vol. 22, no. 4, pp. 1318–1332, 2000.

[24] M. Papadrakakis and N. Bitoulas, "Accuracy and effectiveness of preconditioned conjugate gradient algorithms for large and ill-conditioned problems," *Computer methods in applied mechanics and engineering*, vol. 109, no. 3-4, pp. 219–232, 1993.

[25] *Intel Math Kernel Library. Reference Manual*. Intel Corporation, 2009, santa Clara, USA. ISBN 630813-054US.

[26] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[27] X. S. Li and J. W. Demmel, "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems," *ACM Transactions on Mathematical Software*, vol. 29, no. 2, pp. 110–140, jun 2003. [Online]. Available: https://dl.acm.org/doi/10.1145/779359.779361

[28] S. A. Hutchinson, J. N. Shadid, and R. S. Tuminaro, "Aztec User's Guide Version 1.1," Tech. Rep., 1995.

[29] T. A. Davis and W. Hager, "CHOLMOD: supernodal sparse cholesky factorization and update/downdate," 2005.

[30] T. A. Davis, "Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method," *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 2, pp. 196–199, 2004.

[31] S. Yesil, A. Heidarshenas, A. Morrison, and J. Torrellas, "Speeding up spmv for power-law graph analytics by enhancing locality vectorization,"

in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, vol. 2020-November. IEEE Computer Society, nov 2020, pp. 1–15.

[32] R. Li and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 443–466, 2013.

[33] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. IEEE, 2007, pp. 1–12.

[34] S. Kamin, M. J. Garzarán, B. Aktemur, D. Xu, B. Yılmaz, and Z. Chen, "Optimization by runtime specialization for sparse matrix-vector multiplication," in *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, 2014, pp. 93–102.

[35] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 678–689.

[36] R. W. Vuduc, *Automatic performance tuning of sparse matrix kernels*. University of California, Berkeley, 2003.

[37] R. W. Vuduc and H.-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *International Conference on High Performance Computing and Communications*. Springer, 2005, pp. 807–816.

[38] Y. Li, P. Xie, X. Chen, J. Liu, B. Yang, S. Li, C. Gong, X. Gan, and H. Xu, "VBSF: a new storage format for SIMD sparse matrix–vector multiplication on modern processors," *Journal of Supercomputing*, vol. 76, no. 3, pp. 2063–2081, mar 2020. [Online]. Available: https://doi.org/10.1007/s11227-019-02835-4

[39] W. T. Tang, R. Zhao, M. Lu, Y. Liang, H. P. Huyng, X. Li, and R. S. M. Goh, "Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on intel xeon phi," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 136–145.

[40] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 300–314.

[41] G. Ortega, F. Vázquez, I. García, and E. M. Garzón, "Fastspmm: An efficient library for sparse matrix matrix product on GPUs," *The Computer Journal*, vol. 57, no. 7, pp. 968–979, 2014.

[42] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014.

[43] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: Klu, a direct sparse solver for circuit simulation problems," *ACM Transactions on Mathematical Software (TOMS)*, vol. 37, no. 3, pp. 1–17, 2010.

[44] J. Shin, M. W. Hall, J. Chame, C. Chen, P. F. Fischer, and P. D. Hovland, "Speeding up Nek5000 with autotuning and specialization," in *Proceedings of the International Conference on Supercomputing*. New York, New York, USA: ACM Press, 2010, pp. 253–262. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1810085.1810120

[45] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C. K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Annual ACM Symposium on Parallelism in Algorithms and Architectures*. New York, New York, USA: ACM Press, 2011, pp. 117–128. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1989493.1989508

[46] H. L. A. van der Spek and H. A. G. Wijshoff, "Sublimation: Expanding data structures to enable data instance specific optimizations," in *Languages and Compilers for Parallel Computing*, K. Cooper, J. Mellor-Crummey, and V. Sarkar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 106–120.

[47] N. Vasilache, C. Bastoul, and A. Cohen, "Polyhedral code generation in the real world," in *International Conference on Compiler Construction*. Springer, 2006, pp. 185–201.

[48] G. Agrawal, J. Saltz, and R. Das, "Interprocedural partial redundancy elimination and its application to distributed memory compilation," *ACM SIGPLAN Notices*, vol. 30, no. 6, pp. 258–269, 1995.

[49] R. Das, P. Havlak, J. Saltz, and K. Kennedy, "Index array flattening through program transformation," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, 1995, pp. 70–es.

[50] G. Rodríguez and L. Pouchet, "Polyhedral Modeling of Immutable Sparse Matrices," *impact.gforge.inria.fr*. [Online]. Available: http://impact.gforge.inria.fr/impact2018/papers/modeling-immutable-sparsemat.pdf

[51] J. Saltz, K. Crowley, R. Michandaney, and H. Berryman, "Run-time scheduling and execution of loops on message passing machines," *Journal of Parallel and Distributed Computing*, vol. 8, no. 4, pp. 303–312, 1990.

[52] A. LaMielle and M. M. Strout, "Enabling code generation within the sparse polyhedral framework," *Technical report, Technical Report CS-10-102*, 2010.

[53] M. M. Strout, G. Georg, and C. Olschanowsky, "Set and relation manipulation for the sparse polyhedral framework," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2012, pp. 61–75.

[54] M. S. Mohammadi, T. Yuki, K. Cheshmi, E. C. Davis, M. Hall, M. M. Dehnavi, P. Nandy, C. Olschanowsky, A. Venkat, and M. M. Strout, "Sparse computation data dependence simplification for efficient compiler-generated inspectors," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 594–609.

[55] M. S. Mohammadi, K. Cheshmi, M. M. Dehnavi, A. Venkat, T. Yuki, and M. M. Strout, "Extending index-array properties for data dependence analysis," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2018, pp. 78–93.

[56] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi, "ParSy: Inspection and transformation of sparse matrix computations for parallelism," in *Proceedings - International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018*. Institute of Electrical and Electronics Engineers Inc., mar 2019, pp. 779–793.

[57] B. Zarebavani, K. Cheshmi, B. Liu, M. M. Strout, and M. M. Dehnavi, "HDagg: hybrid aggregation of loop-carried dependence iterations in sparse matrix computations," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 1217–1227.

[58] G. Rodríguez, J. M. Andión, M. T. Kandemir, and J. Touriño, "Trace-based affine reconstruction of codes," in *Proceedings of the 14th International Symposium on Code Generation and Optimization, CGO 2016*. New York, NY, USA: Association for Computing Machinery, Inc, feb 2016, pp. 139–149. [Online]. Available: https://dl.acm.org/doi/10.1145/2854038.2854056

[59] A. Ketterlin and P. Clauss, "Prediction and trace compression of data access addresses through nested loop recognition," in *Proceedings of the 2008 CGO - Sixth International Symposium on Code Generation and Optimization*. New York, New York, USA: ACM Press, 2008, pp. 94–103. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1356058.1356071

[60] M. Selva, F. Gruber, D. Sampaio, C. Guillon, L. N. Pouchet, and F. Rastello, "Building a polyhedral representation from an instrumented execution: Making dynamic analyses of nonaffine programs scalable," *ACM Transactions on Architecture and Code Optimization*, vol. 16, no. 4, pp. 1–26, dec 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3363785

[61] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson *et al.*, "Xsede: accelerating scientific discovery," *Computing in science & engineering*, vol. 16, no. 5, pp. 62–74, 2014.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

1 ABSTRACT

We provide details of building the artifact and running the experiments in the "vectorizing sparse matrix codes using partially strided codelets" paper. The paper proposes a new vectorization method, LCM I/E, that inspects the memory access patterns of a sparse computation and vectorizes the code. We provide instructions on how to run LCM I/E and how to compare it with different methods in the paper using the provided singularity image.

2 DEPENDENCIES AND SOFTWARE VERSIONS

This section provides details of the required packages for this artifact. All required packages are installed inside the singularity image(needs singularity v.3.7 or higher). However, the following packages are used inside the artifact.

(1) a C++ compiler (gcc-10), CMake 3.2, and Git for pulling and building the artifact from source code.
(2) MKL Library(v.2021) for comparing its performance.
(3) python3 for plotting figures.
(4) Sympiler code generator for comparison. The latest version is obtained from `https://github.com/sympiler/sympiler.git`
(5) CSR5 is an open-source tool used to compare its performance with other tools. The latest version of CSR5 is obtained from `https://github.com/weifengliu-ssslab/Benchmark_SpMV_using_CSR5.git` and is modified to run on modern Intel processors efficiently.
(6) The main codelet_mining repository, available from `https://zenodo.org/record/6977299`.

3 PLATFORM DETAILS

The architecture is an Intel Xeon(R) Gold processor with 20 cores. We use one node of the Niagara server, which is provided by compute Canada. The artifact should run on other settings as long as dependent packages are installed. The performance plots should be reproducible for Skylake/CascadeLake processors.

4 DATASET

We use matrices from the SuiteSparse matrix repository. All real-typed matrices larger than 100K nonzero elements are selected to compare the performance of sparse matrix-vector multiplication (SpMV) and sparse matrix times dense matrix (SpMM). All SPD matrices larger than 100K are also selected to compare the performance of sparse triangular solver (SpTRSV) across different methods. Two separated scripts (`dl_matrices.py` and `dl_SPD_matrices.py`) are provided in the mining-bench repository to download these two sets of matrices.

5 RUNNING DEMO EXAMPLES

The artifact has a demo for each of the three kernels, i.e. sparse matrix-vector product (SpMV), sparse lower triangular solver (SpTRSV), and sparse matrix times dense matrix (SpMM). A separate demo is also provided for MKL, CSR5, regular-piecewise, and SPF-ELL libraries. All demos will take the matrix file path as their input argument and runs the corresponding kernel/implementation. Other input parameters are explained through calling *–help* switch from the demo. For example, to run the SpMV demo, the following instruction is needed:

```
singularity exec artifact.sif
/source/codelet_mining/build/demo/spmv_demo -matrix
./path/to/matrix.mtx -numerical_operation SPMV
-storage_format CSR
```

6 EXPERIMENT LIST

We provide instructions on how to reproduce Figures 6, 7, 9, 10, and 11 of the submitted draft. These figures compare the performance of LCM I/E (our proposed methods) with MKL, CSR5, regular piece-wise, sparse polyhedral framework, and Sympiler. Figures 7 and 9 additionally provide the effect of different mined codelets. Three significant experiments should be conducted to reproduce the mentioned figures.

(1) The SpMV experiment compares the performance of SpMV across different tools and generates data for Figures 6 and 7.
(2) The SpTRSV experiment compares the performance of SpTRSV across different tools and generates data for Figures 9 and 10.
(3) The SpMM experiment compares the performance of MKL and our framework as shown in Figure 11.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1
Persistent ID: `github.com/cheshmi/mining-bench`
Artifact name: Codelet Mining Instructions

Artifact 2
Persistent ID: `https://github.com/sympiler/sympiler.git`
Artifact name: Sympiler

Artifact 3
Persistent ID: `https://github.com/weifengliu-ssslab/\\Benchmark\_SpMV\_using\_CSR5.git`
Artifact name: CSR5

Artifact 4
Persistent ID: `library://kazem/kazem/artifact22:latest`
Artifact name: Codelet Mining Singularity Image

Artifact 5
Persistent ID: `https://zenodo.org/record/6977299`
Artifact name: Codelet Mining original repository

Reproduction of the artifact with container:

7 RUNNING EXPERIMENTS USING SINGULARITY IMAGE

To run the list of experiments, the singularity images and matrix datasets should be downloaded first, and then provided scripts in the repository should be used to run the experiments. We show all steps to run experiments here:

(1) The mining-bench repository should be cloned: git clone https://github.com/cheshmi/mining-bench.git
cd mining-bench
(2) The singularity image should be pulled to the same directory that the code is cloned using:
singularity pull artifact.sif library://kazem/kazem/artifact22:latest
You can test the image by running the following command from the current directory:
singularity exec artifact.sif /source/codelet_mining/build/demo/spmv_demo –matrix ./LFAT5.mtx –numerical_operation SPMV –storage_format CSR
The output is a set of comma-separated values (CSV) such as matrix specification and execution time of different tools.
(3) The datasets should be downloaded by calling:
python ssgetpy/dl_matrices.py
python ssgetpy/dl_SPD_matrices.py
Matrices are downloaded into the `mm` and `SPD` directories in the current directory (This might take several hours and requires an internet connection).
(4) The SpMV experiment can be executed by emitting: bash run_spmv.sh
For running on compute node: sbatch bash run_spmv.sh
You might need to update scripts with new absolute paths to the dataset and the image file. You will also need to load the singularity module.
(5) SpTRSV experiment can be done by running:
bash run_sptrsv.sh
(6) SpMM experiment can be reproduced by calling:
bash run_spmm.sh
(7) Upon successful completion of experiments, all results should be stored as CSV files under the `./logs/` directory and can be used for plotting. Separated Python scripts are provided to generate each figure. Each experiment calls a python script to plot data in generated CSV files.