SpEQ: Translation of Sparse Codes using Equivalences

AVERY LAIRD, University of Toronto, Canada
BANGTIAN LIU, University of Toronto, Canada
NIKOLAJ BJØRNER, Microsoft Research, USA
MARYAM MEHRI DEHNAVI, University of Toronto, Canada

We present SpEQ, a quick and correct strategy for detecting semantics in sparse codes and enabling automatic translation to high-performance library calls or domain-specific languages (DSLs). When sparse linear algebra codes contain implicit preconditions about how data is stored that hamper direct translation, SpEQ identifies the high-level computation along with storage details and related preconditions. A run-time check guards the translation and ensures that required preconditions are met.

We implement SpEQ using the LLVM framework, the Z3 solver, and `egglog` library [17, 28, 56] and correctly translate sparse linear algebra codes into two high-performance libraries, NVIDIA cuSPARSE and Intel MKL, and OpenMP (OMP) [6, 12, 36]. We evaluate SpEQ on ten diverse benchmarks against two state-of-the-art translation tools. SpEQ achieves geometric mean speedups of 3.25×, 5.09×, and 8.04× on OpenMP, MKL, and cuSPARSE backends, respectively. SpEQ is the only tool that can guarantee the correct translation of sparse computations.

CCS Concepts: • **Software and its engineering** → *Software verification and validation*; **Translator writing systems and compiler generators**.

Additional Key Words and Phrases: Program Analysis, Verification, Equivalence Checking, Equality Saturation

1 INTRODUCTION

Optimized libraries and domain-specific languages (DSLs) enable high performance for a diverse range of applications. By focusing on a specific domain or computation, libraries and DSLs safely apply aggressive optimizations that a general-purpose compiler cannot. Ideally, a user provided program should be automatically translated to a desired library call or DSL to take advantage of such optimizations. However, programmers can write any number of implementations for the same computation, making syntax or structure-directed translation fragile. Codes also often have implicit *preconditions* (assumptions about their input data) that must be derived for a correct translation. In this paper, we focus on detecting preconditions for sparse linear algebra computations, which operate on data stored in compressed tensors. Because sparse codes may have many implementations and complicated preconditions due to compressed tensors, it is difficult to detect their semantics for the purpose of safe translation.

There are two main strategies for detecting code semantics: *idiom matching* and *verified lifting*. Idiom matching searches the user program for specific patterns that express a computation, called idioms, and replaces them with the desired library call [4, 15, 21]. While idiom matching has a low overhead, creating expressive and correct idioms for different computations requires expert knowledge. Instead of searching for specific patterns, verified lifting searches the space of all programs in a target DSL or library until an *equivalent* program is found [2, 3, 10, 24]. While the search can be guided through a user-provided grammar, it requires expert knowledge to derive and is often highly specific to the user program and target domain. Additionally, proving equivalence becomes expensive on programs with unbounded loops, as each loop requires an inductive proof known as an *invariant*. Finally, both idiom matching and verified lifting cannot derive preconditions from input code, leaving sparse codes out of scope.

Authors' addresses: Avery Laird, University of Toronto, Canada, alaird@cs.toronto.edu; Bangtian Liu, bangtian@cs.toronto.edu, University of Toronto, Canada; Nikolaj Bjørner, Microsoft Research, USA, nbjorner@microsoft.com; Maryam Mehri Dehnavi, mmehride@cs.toronto.edu, University of Toronto, Canada.

This work introduces SpEQ (Translation of **Sp**arse Codes using **Eq**uivalences), a program analysis tool that detects the preconditions of sparse linear algebra codes and then safely replaces the input code with high-performance library calls. SpEQ combines *simulation* techniques with *equality saturation* to first identify preconditions for sparse codes and then correctly detect the code's semantics. Given a user input code f in LLVM Intermediate Representation (IR) [28], SpEQ detects any storage formats used and their relevant preconditions. SpEQ uses knowledge of storage formats to construct a dense version of the input code (without preconditions), f', and a relation R between both versions f and f'. By using *stuttering simulation* to prove f and f' are equivalent under R, SpEQ now has a precondition-free program f'. Next, SpEQ uses equality saturation to match f' with a collection of code semantics. The code semantics are described through reference implementations of a computation, given as regular executable code. Any language with an LLVM IR frontend, such as C/C++, Fortran, or Rust, is acceptable. During an offline phase before compilation, SpEQ *abstracts* every reference implementation into a rewrite rule. The goal of equality saturation is to rewrite the input program f' as a function call to a reference implementation. Once the semantics of f' are matched to known reference implementations, a *backend* uses this information to generate a library call or DSL code. Finally, SpEQ generates an executable run-time check for each detected storage format that guards the translated f' to ensure the storage format's preconditions are met at run-time.

We present a lightweight strategy for detecting possible storage formats used in a sparse input program. Then, we implement SpEQ using the LLVM framework, Z3 SMT solver, and `egglog` equality saturation library [17, 28, 56]. From a diverse set of benchmarks, SpEQ correctly lifts Sparse Matrix Vector multiplication (SpMV), General Matrix Multiply (GEMM), histograms, and reductions. We implement backends for OpenMP [6], NVIDIA cuSPARSE [12], and Intel MKL [36].

The contributions of this paper are:

- A strategy for (1) detecting storage formats used in sparse linear algebra kernels, and (2) generating run-time checks to verify that storage format preconditions are satisfied.
- A method for applying equality saturation to LLVM IR, along with a generic set of rewrite rules, that enable code semantics to be matched quickly and correctly.
- An implementation of SpEQ with three example backends.

2 RUNNING EXAMPLE

In the running example shown in Figure 1, the input to SpEQ is a code implementing sparse matrix-vector multiplication (SpMV) in the *compressed sparse row* (CSR) storage format. CSR is composed of three arrays: `a` and `col` arrays store the nonzero values and column indices respectively, and the `row` array stores the starting point of each row in the `a` and `col` arrays. As shown in Figure 1a, two for-loops are typically used to implement SpMV based on CSR. The inner loop computes a dot-product for one specific row, and the outer loop iterates over all rows. Consider the scenarios where the `row` array elements are not (non-strictly) increasing, or the `col` array elements are not unique; the resulting computation is no longer an SpMV, and should not be replaced with an SpMV library call. The properties that elements of `row` should be increasing (monotonicity), and that elements of `col` should be unique column indices in a row (periodic monotonicity), are preconditions of the CSR storage format and well-studied in prior works [11, 53]. After detecting that the example code is using the CSR format, SpEQ generates a run-time check (shown in Figure 1d) to guard the running example code and guarantee its preconditions are met. In this example, SpEQ translates the example code into a series of NVIDIA cuSPARSE library calls that perform SpMV on the GPU [12], shown in Figure 1f.

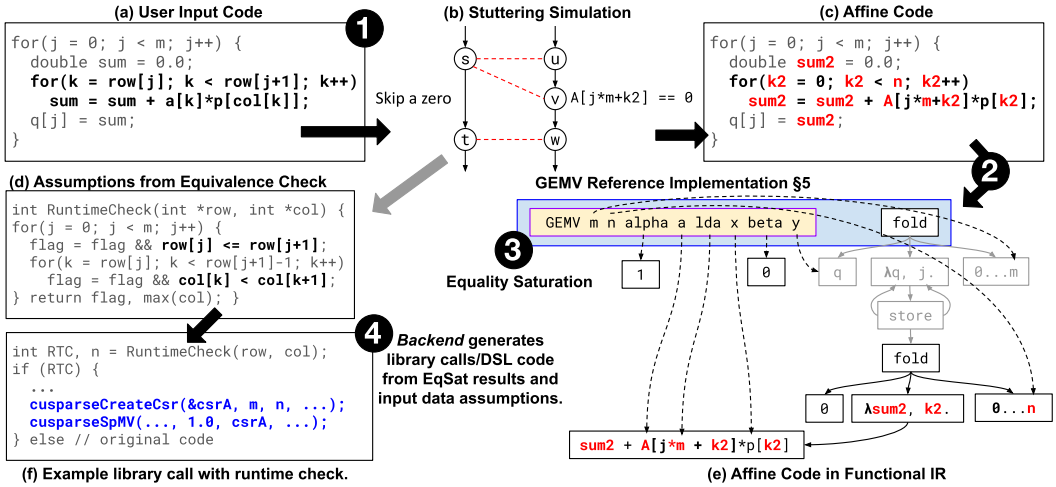


Fig. 1. An example of SpEQ translating a sparse-matrix vector multiplication into a cuSPARSE library call. In step one, SpEQ detects that the example input code (1a) may use compressed-sparse row (CSR) format to represent a sparse matrix (Section 4.1.1). Under this assumption, SpEQ transforms the sparse representation to dense (1c). To ensure the transformation is correct, both codes are translated to transition systems with a simulation relation (red dashed lines) between them (1b). SpEQ verifies that the two systems are equivalent by proving the relation is a *stuttering simulation* (Section 4). During step two, SpEQ converts the affine code into a functional IR and adds it to an *e-graph*. Step three performs equality saturation to find reference implementations (provided during an offline phase) that are equivalent to the input code (Section 5). In this example, the input code can be rewritten to a GEMV reference implementation, with a function signature shown in the yellow box. By rewriting the transformed input code into a GEMV call, the correct arguments for `m`, `n`, `a`, etc., are extracted automatically, along with parameters that do not appear in the input code such as `alpha` and `beta`. In step four, an example cuSPARSE *backend* module takes the GEMV and CSR information as input to generate an SpMV library call (1e). SpEQ uses the relation from step one to automatically generate a run-time check (1d) that verifies all input data assumptions (e.g, CSR) are satisfied (Section 4.5). Otherwise, the original input code is executed.

❶ In step one, SpEQ identifies possible sparse storage formats through a static program analysis (Section 4.1.1). SpEQ uses the results of static analysis to propose that `(row,col,a)` in Figure 1a represent a sparse matrix in CSR format, with `p` and `q` being dense vectors. Based on this assumption, SpEQ constructs a dense implementation of the input code (Section 4.1.2), shown in Figure 1c, along with a relation between the sparse and dense transition systems, shown in 1b.

Figure 1b shows two transition systems; on the left side are two iterations s, t of the inner loop in Figure 1a, and on the right are three iterations u, v, w of the inner loop in Figure 1c. The relation between the two systems is shown as a red dashed line. Both systems are processing three consecutive elements $A_{ij}, 0, A_{ij+1}$ of the sparse input matrix. The left-hand side of Figure 1b shows how the loop in Figure 1a progresses from state $s \rightarrow t$, skipping the zero element between A_{ij} and A_{ij+1}. On the right-hand side, states $u \rightarrow v \rightarrow w$ process elements $A_{ij}, 0, A_{ij+1}$, respectively, without skipping the zero element. If `A[j*m+k2] == 0` at state v, then states v and s are equivalent, shown by the relation between systems (red dashed line). The equivalence check based on this relation only succeeds when certain input data properties are satisfied. For example, two properties are required for this equivalence check to succeed: values in `row` must increase (monotonicity), and values in `col` must strictly increase in each row (periodic monotonicity). These properties relate

to the CSR storage format per [11, 53]. If the equivalence check succeeds, SpEQ knows that the correct storage format was identified during static analysis and generates a run-time check for the format's preconditions (Figure 1d).

❷ During step two, a functional IR (FIR) form of the transformed input code (Section 3) is added to an *e-graph* to search for equivalences with different known computations. The functional IR is important for encoding properties that are difficult to express in LLVM IR, such as multiple memory versions. Inside the functional IR, loops are represented as the **`fold`** operator, which takes four inputs:

$$\langle r_1, \ldots, r_n \rangle = \textbf{fold}\ \langle a_1, \ldots, a_n \rangle\ \ (\lambda\ \langle p_1, \ldots, p_n \rangle\ iv.\ body)\ \ lb\ ub$$

Where $\langle a_1, \ldots, a_n \rangle$ are the initial values for $\langle p_1, \ldots, p_n \rangle$, respectively; the second argument, *combining function*, is the loop body with input variables $\langle p_1, \ldots, p_n \rangle$; and final two arguments are the iteration domain for induction variable iv. The combining function is executed $ub - lb$ times, then the current values of $\langle p_1, \ldots, p_n \rangle$ are returned as $\langle r_1, \ldots, r_n \rangle$, respectively. Figure 1e shows the loop nest in Figure 1c in FIR. For example, the outer loop body stores a value `sum2` to `q[j]` and returns a new version of `q`, represented by the `store` node in Figure 1e. This new version of `q` is used as input to the next execution of the loop body. The value `sum2` is generated by the inner loop and represented by a second **`fold`** function. The input code's functional representation is added to the e-graph in preparation for the equality saturation phase.

❸ In step three, SpEQ uses equality saturation to search for equivalences between the input code and a collection of reference implementations. GEMV is one such reference implementation in this example (Figure 9a). During an offline phase before compilation, SpEQ transforms all reference implementations into the functional IR and *abstracts* them into a rewrite rule (Section 5). In addition to rewrite rules derived from reference implementations, SpEQ has a set of generic rewrite rules for common code transformations in the functional IR, such as loop fission/fusion/interchange and sinking/hoisting. If the input code can be transformed to match a reference implementation through a series of rewrite rule applications, it is equivalent to that reference implementation (shown in the blue box). The correct arguments for a call to the reference implementation are extracted through the rewriting process. The result of step three is (1) a list of computations that are equivalent to the input code, and (2) the values from the input code that should be provided as arguments to the computations.

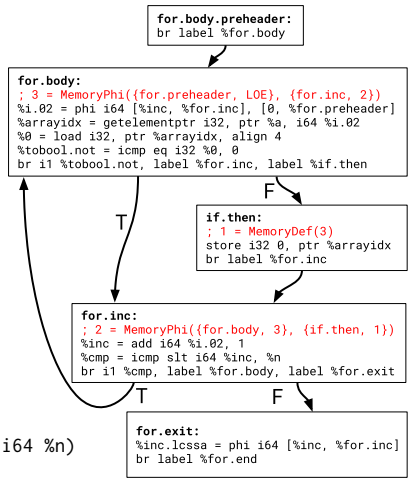
❹ In the final step, a separate *backend* module uses the output from step three and the sparse storage information to generate library calls or DSL code. This example shows the cuSPARSE backend, which generates calls to the cuSPARSE library (Figure 1f). From step three, the high-level computation of the input code is recognized as GEMV. However, the dense matrix `A` is symbolic; the original input code has the sparse matrix represented by `(row, col, a)` as input. Based on this information, the cuSPARSE backend first initializes the sparse matrix through a call to `cusparseCreateCsr`, guarded by the run-time check. The sparse matrix input argument `a` to GEMV also uniquely identifies the computation as SpMV. Therefore, the cuSPARSE backend generates the relevant library call to `cusparseSpMV`.

3 LLVM IR TO FUNCTIONAL IR

This section presents the first phase of SpEQ's pipeline, which converts LLMV IR to a Functional IR (FIR) to enable the stuttering simulation and equality saturation steps. To support multiple languages such as C/C++ and Fortran, SpEQ takes LLVM IR [28] as input. However, encoding LLVM IR directly as a transition system and in an e-graph is challenging due to how store instructions and control flow are represented. As shown in Figure 1, we want store and branch instructions to represent a value; store instructions return the updated array, and branch instructions return the

```
long int fn(int *a, long int n) {
  long int i;
  for (i = 0; i < n; ++i) // for.body
    if (a[i])
      a[i] = 0;
  return i;
}
```

(a) Example input code.



(b) Control flow graph of the loop in 2a.

```
%for.body = λ ptr %a, i64 %i.02 .
  %arrayidx = getelementptr i32, ptr %a, i64 %i.02
  %0 = load i32, ptr %arrayidx
  %tobool.not = icmp eq i32 %0, 0
  %a.1 = store i32 0, ptr %arrayidx
  %a.2 = if %tobool.not then %a.1 else %a
  %inc = add i64 %i.02, 1
(%a.2, %inc) = fold (ptr %a) %for.body Range(i64 0, i64 %n)
```

(c) The loop in Figure 2a in FIR.

Fig. 2. (2a) An example input code with a conditional store to `a[i]`. (2b) The example input code in LLVM IR, with MemorySSA analysis (red text). (2c) The example input code in FIR.

chosen control path. However, in LLVM IR, both store and branch instructions are considered to return no value (Figure 2b). Adding direct support inside LLVM for special memory semantics and branch instructions is a significant engineering effort. Instead, we propose FIR to augment LLVM IR with additional information for implementing the equivalence check and equality saturation phases.

3.1 Store Instructions

SpEQ associates a register with each store instruction so that different memory versions are represented explicitly as values. The register names are determined by LLVM's MemorySSA analysis [1], which provides a static single-assignment (SSA) form of memory (shown in Figure 2b in red). However, MemorySSA deliberately provides a coarse-grain heap model, where each store updates a single shared heap. SpEQ uses a fine-grain heap model, where each pointer is assumed to address non-overlapping memory regions (no aliasing). The compiler generates a no-alias check to ensure pointers do not alias at run-time.

Figure 2b shows the control-flow graph (CFG) from input program 2a. The loop conditionally stores a `0` to `a[i]`, shown in the `if.then` block. The analysis from MemorySSA, shown in red, represents the memory update as a `MemoryDef` that creates heap version `1` from heap version `3`. Rather than an update to the entire heap, SpEQ considers this an update to the separate array `a`. Therefore, the value of the store instruction is assigned to register `%a.1`, as shown in Figure 2c.

3.2 Control Flow

Our goal is to represent branching control-flow in LLVM IR as a *value* of the form **`if`** *c* **`then`** *a* **`else`** *b*. If *c* is true then the expression has the value *a*, otherwise it has the value *b*. In LLVM IR, *a* and *b* are not represented directly in the branch instruction but in `phi` and `MemoryPhi` instructions where control-paths merge. Figure 2b shows a branch instruction in `for.body` that creates two paths to `for.inc` and `if.then`. The path along `if.then` modifies `a`, while the other does not. When the two

paths merge at `for.inc`, the `MemoryPhi` instruction chooses the proper value from its predecessors. In this paper we refer to both `phi` and `MemoryPhi` instructions as `phi` *nodes*.

To associate `phi` nodes with the correct branch condition, SpEQ uses the *dominator tree*. The dominator tree is a common compiler analysis used in many scenarios [13, 43]. A block Y *dominates* block X if every path from the entry block to X must pass through Y. For example, `for.body` dominates `for.inc` and `if.then` (Figure 2b). Consider a `phi` node in a block C with two incoming values from blocks B_1 and B_2, respectively. There must be some block A with a branch instruction that created the separate paths to B_1 and B_2. This block A must dominate both B_1 and B_2: it is the *nearest common dominator*. Additionally, A is the block that contains the branch instruction associated with the `phi` node in C. Therefore, SpEQ uses the dominator tree to find the nearest common dominator and the associated branch condition.

In Figure 2b, `for.body` is the nearest common dominator of `if.then` and itself. Therefore, `%tobool.not` is the branch condition associated with `MemoryPhi 2`. The `phi` node is represented as a functional **`if`** that returns `%a.1` or `%a` based on the value of `%tobool.not` (Figure 2c).

In addition to conditional branches, LLVM also controls execution flow with `switch`, function calls, and exceptions. In this paper, we focus on branch instructions only, as it is the most common case in codes we focus on.

3.3 Loops

SpEQ also represents loops as values to provide several benefits for equivalence checking and equality saturation. Recall that SpEQ represents loops using the **`fold`** function:

$$\langle r_1, \ldots, r_s \rangle = \textbf{fold}\ \langle a_1, \ldots, a_n \rangle\ \ (\lambda\ \langle p_1, \ldots, p_n \rangle\ iv.\ body)\ \ lb\ ub$$

Figure 2c shows an example **`fold`** representation of the program in Figure 2a. The return values $\langle r_1, \ldots, r_s \rangle$ are the loop's *live-outs*; registers that are live across a loop boundary, and any modified memory regions. The `%inc` register in Figure 2b, which is used in a return statement outside the loop, is one example. The value of the example loop is the tuple `(%a.2, %inc)` as shown in Figure 2c. To easily retrieve live-outs, SpEQ converts loops into *Loop Closed SSA Form* (LCSSA) [30], which places all live-outs inside the loop exit block.

The initial values $\langle a_1, \ldots, a_n \rangle$ are computed from the `phi` nodes in the loop *header* block (the block that dominates all loop blocks). To guarantee that the header block has a single unique predecessor (the *preheader*), SpEQ also converts loops into *Loop Simplify Form* [31]. Therefore, the initial values for each `phi` node are incoming from the loop preheader. Figure 2b shows how the `MemoryPhi` in block `for.body` (loop header) has the incoming value `LOE` from block `for.preheader` (loop preheader). Although `LOE` represents the entire initial heap state, SpEQ disambiguates this to the initial state of `%a`.

To complete the translation of a loop into a **`fold`**, the combining function is constructed. The input parameters for the combining function are the `phi` nodes from the loop header, including the induction variable *iv*. The iteration domain of the loop is supplied to the **`fold`** through the *lb* and *ub* parameters. The **`fold`** also accepts an optional *stride* argument, which is one by default (we omit the stride in this paper, unless it is not one). Any loop domain with a constant stride that is not modified within the loop body can be described this way. Finally, the combining function *body* is recursively defined as the FIR of the loop's basic blocks: therefore, all instructions in the loop body are guaranteed to be in FIR.

3.4 FIR Requirements

SpEQ performs an initial legality analysis to ensure that the input LLVM IR can be represented in FIR. Loop induction variables must be integers with regular steps (AddRec expressions). This

$$
\begin{array}{rcl}
\textit{Program} & ::= & \text{list } \textit{Loop} \\
\textit{Loop} & ::= & \texttt{fold}\ (\text{list } \textit{Val})\ \textit{Abs}\ \textit{Exp}\ \textit{Exp} \\
\textit{Abs} & ::= & \lambda\ \text{list } \textit{Var}\ \texttt{.}\ \text{list } \textit{Inst} \\
\textit{Inst} & ::= & \text{LLVM IR Instruction} \\
 & | & \texttt{if}\ \textit{Exp}\ \texttt{then}\ \textit{Exp}\ \texttt{else}\ \textit{Exp} \\
 & | & \textit{Loop} \\
\textit{Exp} & ::= & \textit{Val} \mid \textit{Var} \\
\textit{Var} & ::= & \text{LLVM IR Register Name} \\
\textit{Val} & ::= & \text{Constant number}
\end{array}
$$

Fig. 3. The grammar for FIR.

also implies that the induction variable's upper and lower values can be determined through static analysis. Non-memory instructions with side-effects (like IO) are unsupported; however, functions that may modify memory (such as *memset* or user-defined functions) are supported, through the MemorySSA analysis. Finally, a loop nest is representable in FIR only if it and all of it's children are representable.

4 EQUIVALENCE CHECK

This section explains the equivalence check phase, which confirms whether SpEQ has successfully identified all necessary preconditions for safe translation, and if so, returns a precondition-free version of the input code for the equality saturation step. We first explain how storage formats introduce preconditions to sparse linear algebra codes.

Code optimizations are typically *equality preserving*; the semantics of the original program should never be altered. Some optimizations only preserve equality under certain preconditions. For example, $\sqrt{a^2} \mapsto a$ is sound under the precondition that $a \geq 0$. Similarly, sparse computations are the result of optimizing dense computations for sparsity. Any storage format used in a sparse linear algebra computation will represent a valid tensor only under certain preconditions, which are specific to every format. Just as an equivalence check between $\sqrt{a^2}$ and $\mapsto a$ will fail without the sufficiently strong precondition $a \geq 0$, an equivalence check between sparse and dense computations will fail without the preconditions of storage formats. This is why SpMV and GEMV cannot be considered equivalent through typical verified lifting techniques, which do not discover preconditions. On the other hand, idiom matching techniques ignore preconditions entirely, leading to unsafe translation.

SpEQ's goal is to extract the high-level semantics from sparse computation by removing any details related to storage formats. By constructing a *format-agnostic* (dense) version A, and proving it is equal to the sparse input code N under particular preconditions, SpEQ safely extracts the high-level semantics (program A). The equivalence check consists of several stages:

(1) A program analysis identifies storage formats in the input code to create a *format-agnostic* form of the input code and propose preconditions (Section 4.1.1).
(2) SpEQ constructs a *simulation relation* between the input code and its format-agnostic version (Section 4.2).
(3) SpEQ verifies that the simulation relation is sufficient to prove equivalence between the two codes (Section 4.4).

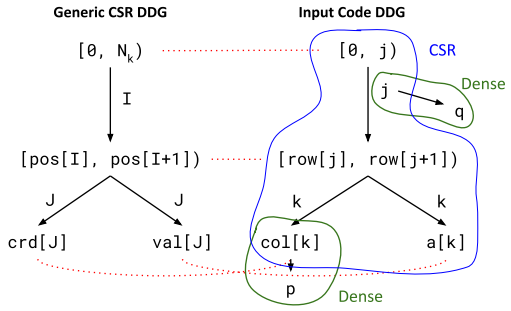


Fig. 4. Data dependence graphs for accessing a CSR matrix (left), and the running example input code (right).

If the equivalence check succeeds, SpEQ uses the result of static analysis to generate an executable run-time check that verifies the preconditions (Section 4.5). We now explain each stage of the equivalence check in order.

4.1 Create Format Agnostic Code

Before performing the equivalence check, SpEQ must identify all preconditions of the sparse input code and propose a precondition-free version. If SpEQ does not capture all necessary preconditions, the equivalence check fails and translation stops. First, we explain the static analysis SpEQ performs to identify storage formats and their preconditions. Next, we explain how SpEQ constructs a mapping between sparse and dense data structures, in order to construct a format-agnostic code version and begin the equivalence check.

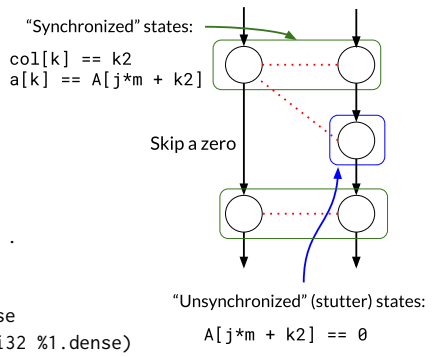
4.1.1 Recognize Storage Formats. The purpose of data compressed formats is to reduce storage requirements and to only store (and compute on) the non-zero elements of a tensor. A wide range of disparate storage formats exists due to diverse sparse patterns that arise from different application areas. Commonly used formats include compressed sparse row (CSR), compressed-sparse column (CSC), and coordinate (COO). Compressed formats are used to retrieve the tensor data; they are not used to express the computation itself. For example, in an implementation of $y = Ax$, A can be stored in any of the compressed formats CSR, CSC, COO, etc.; the computation is still multiplying the sparse matrix with a vector (SpMV), independent of the format chosen. However, prior idiom matching techniques require that a pattern be defined for each possible computation/storage format combination, leading to a large (exponentially growing) number of required patterns. Instead, SpEQ *decouples* computation from storage format details. A key insight is that each storage format induces specific program properties that must exist in the code using that format. SpEQ's strategy for recognizing storage formats is to inspect the data dependence graph (DDG) of input codes for different format characteristics.

Figure 4 shows two DDGs side-by-side. The left-hand side shows a DDG for any code, regardless of computation, that uses the canonical form of CSR (a generic data-flow graph). For example, there must exist an iterator `I`, ranging from `0` to N_k, that indexes a `pos` array. The right-hand side shows the corresponding DDG for the example input code from Figure 1a. Because there is a possible substitution from the left-hand side graph (CSR) to the right-hand side graph (input code), SpEQ considers `(row,col,a)` to describe a sparse matrix stored using the CSR format. We say that the corresponding nodes in the user input code *match* CSR.

In many sparse codes, there is a dependence between different storage formats. For example, in Figure 4, `(row,col,a)` must be identified as a possible sparse matrix to understand `p`'s context as

```
%for = λ double %sum.03, i32 %k.02 .
  %2 = %a[%k.02]
  %3 = %col[%k.02]
  %4 = %p[%3]
  %5 = %2 * %4 + %sum.03
(%5) = fold (double 0.0) %for (i32 %0) (i32 %1)
```

(a) The running example 1a in FIR.



(b) A segment of the joint transition system between 5a and 5c.

```
%for.dense = λ double %sum.03.dense, i32 %k.02.dense .
  %a.dense.elem = %a.dense[%j.05 * m + %k.02.dense]
  %p.elem = %p[%k.02.dense]
  %"5.dense" = %p.elem * %a.dense.elem + %sum.03.dense
(%5.dense) = fold (double 0.0) %for.dense (i32 0), (i32 %1.dense)
crd: (%3 = i32 %k.02.dense)
val: (%2 = double %a.dense.elem)
```

(c) The precondition-free version of 5a in FIR.

Fig. 5. The inner loop of the running example 1a in FIR (5a) and its precondition-free version (5c), with the joint transition system between them (5b).

a dense vector indexed by sparse structure; otherwise, the irregular access on p cannot be analyzed. SpEQ uses a worklist algorithm to find all possible matches until a fixed point is reached (no more matches are found). For example, consider the case where SpEQ attempts to match p with a storage format before a was matched with CSR (Figure 4). Then, no matching format is found for p, and `(row,col,a)` is matched with CSR. On the next iteration of the algorithm, SpEQ succeeds in matching p as a dense 1D vector because `col` is associated with CSR.

SpEQ leverages insights from prior work on generalizing sparse storage formats [11] to search for different formats and their preconditions. SpEQ currently focuses on CSR, CSC, and 1-2 dimensional dense tensors, since they are the formats commonly used in sparse linear algebra codes. Additional or non-standard storage formats can be supported by manually providing their corresponding generic data-flow graph to SpEQ. Next, we describe how sparse storage format information is used to construct a format-agnostic version of the input program.

4.1.2 Transform to Dense. The format-agnostic version is created based on the format recognition step. Figure 5a shows the FIR for the inner loop of the running example (Figure 1a). Figure 5c shows the format-agnostic version of 5a. For clarity, the `load` and `getelementptr` instructions are replaced with C-style array syntax. Each detected storage format is converted into a dense implementation, and a mapping back to the input code is stored at the end of the **`fold`** command. For example, the `col` array stores column indices of the dense matrix. Therefore, the equality `%col[%k.02] == %k.02.dense` must hold. This is stored in the `crd` map, shown in Figure 5c. Additionally, the iteration domain of each **`fold`** is transformed to have the range $[0, m)$, where m is a fresh integer constant. The `crd` and `val` maps relate the sparse data structures in the input code to the dense data structures in the format-agnostic code. The next section explains this relation in more detail.

4.2 Construct Simulation Relation

In previous steps, SpEQ identified storage formats in the sparse input code and their preconditions, then constructed a format-agnostic input code; this section explains how SpEQ performs the equivalence check between the sparse and format-agnostic codes. Intuitively, two programs are equivalent if they reach the same states in the same order. This is known as *simulation* between two programs A and B: when A moves to a new successor state s, system B must have a related state s' [46].

However, some programs that do not satisfy simulation should still be considered equivalent. Sparse computations are an example of such programs. Consider the running example and its format-agnostic version in Figure 5a. When loop 5a skips a zero element, loop 5c does not. Loop 5c has a successor state that does not exist for loop 5a, therefore, they are not similar. SpEQ uses a relaxed version of simulation, called *branching* or *stuttering* simulation [38, 46], to prove equivalence between such loops.

Figure 5b shows an example stuttering simulation between loop 5a and loop 5c. Related states are shown with red dashed lines. States are related in two possible ways. One possibility is that the states are *synchronized*, as shown by the green box in Figure 5b. This represents the case when both programs are in the same state and operating on the same input data (for example, a sparse and dense program processing nonzero elements). Synchronization conditions are stored in the `crd` and `val` maps. The `crd` map stores relations between index functions of sparse structures and dense structures in each program. The `val` map stores relations between the input data of each program. For example, the maps in loop 5c have the following equalities:

```
%col[%k.02] == %k.02.dense
  %a[%k.02] == %a.dense
```

Under the assumption that `(row,col,a)` represent a sparse matrix, these conditions relate equivalent program states.

States can also be *unsynchronized*, rather than synchronized. The blue box in Figure 5b shows an example unsynchronized state. Unsynchronized states occur from stutter transition, where there is no corresponding state in the other transition system. For example, when a sparse code skips a zero, the dense version has an extra unsynchronized state that processes the zero. Such states are related only if the dense matrix contains a zero during the stutter transition; therefore, SpEQ relates unsynchronized states by setting elements in the `val` map to zero. Figure 5b shows an example where `A[j*m+k2] == 0`.

At this stage, SpEQ has a relation between sparse and dense programs, along with a sufficiently flexible notion of equivalence: stuttering simulation. However, it remains to prove that the relation proposed by SpEQ is actually a stuttering simulation. Next, we formalize how SpEQ proves a proposed relation is a stuttering simulation.

4.3 Transition Systems

With our goal of establishing that two sequential programs compute the same values, we use tools from simulation to formulate and automatically check a co-inductive argument for equivalence. In the following we treat the sequential programs as transition systems and formulate a proof rule that implies stuttering simulation between the two programs. The proof rule makes use of a third transition system that captures the *joint* behaviour of the two systems.

A transition system TS is a tuple $\langle V, \Theta, \mathbb{T} \rangle$, where V is a finite set of system variables, Θ is a satisfiable assertion characterizing all initial states, and $\mathbb{T}$ is a finite set of *transitions* [34]. The i'th state in an execution, σ_i, assigns system variables V to values. Every transition $t \in \mathbb{T}$ is a function that maps each state $\sigma_i \in \Sigma$ to a set of states $t(\sigma_i) \subseteq \Sigma$. Any state in $t(\sigma_i)$ is a *t-successor* of σ_i. The

transition t is called *enabled* for a state σ_i if $t(\sigma_i) \neq \emptyset$. Otherwise, it is *disabled*. An execution trace σ is a finite or infinite sequence of states, $\sigma_0, \sigma_1, \ldots$, such that $\sigma_0 \models \Theta$ and for $i > 0$, $\sigma_i \in t(\sigma_{i-1})$ for some $t \in \mathbb{T}$ for every state σ_i. If σ is finite, none of the transitions in $\mathbb{T}$ are enabled in the last state σ_n.

Every transition $t \in \mathbb{T}$ is represented by a first-order formula $\rho_t(V, V')$, known as the *transition relation*, where the set V' is a copy of V of primed variables. The transition relation expresses the relation between a state σ_i and any t-successors $\sigma_{i+1} \in t(\sigma_i)$ if and only if $\sigma_i, \sigma_{i+1} \models \rho_t(V, V')$ where σ_{i+1} is an evaluation of the variables in V'. The set of transitions $\mathbb{T}$ can be summarized as a single transition relation as $\rho \equiv \bigvee_{t \in \mathbb{T}} \rho_t$. Thus, we can also represent a transition system as $\langle V, \Theta, \rho \rangle$. For example, the transition relation $\mathtt{sum}' = \mathtt{sum} + 1$ expresses that $\mathtt{sum}'$, the value of $\mathtt{sum}$ in σ_{i+1}, is 1 greater than $\mathtt{sum}$, the value of $\mathtt{sum}$ in σ_i. Whether a transition t is enabled or disabled is also expressed using the transition relation ρ_t: $En(t) = \exists V'.\ \rho_t(V, V')$. Note that $En(t)$ is true only if $t(\sigma_i)$ is not empty for at least some σ_i. In addition to transitions extracted from code, we introduce the *idling* (or *stuttering*) transition ϵ, where $\rho_\epsilon : V = V'$. The transition is used to model a sparse program that does not advance, while the dense version processes a zero.

4.3.1 Transition Systems from Folds. Consider any **fold** in FIR:

$$\langle r_1, \ldots, r_s \rangle = \textbf{fold}\ \langle a_1, \ldots, a_n \rangle\ (\lambda\ \langle p_1, \ldots, p_n \rangle\ iv.\ body)\ lb\ ub$$

We create a tuple $acc = \langle acc_1, \ldots, acc_n \rangle$ with the same sort as $\langle a_1, \ldots, a_n \rangle$. Then, the transition system for a fold is straightforwardly extracted:

$$\begin{aligned} V &= \langle p_1, \ldots, p_n \rangle \\ \Theta &= acc = \langle a_1, \ldots, a_n \rangle \wedge iv = lb \\ \rho_{body}(V, V') &= iv < ub \\ &\quad \wedge iv' = iv + stride \\ &\quad \wedge acc' = (\lambda\ \langle p_1, \ldots, p_n \rangle\ iv.\ body)\ \langle acc_1, \ldots, acc_n, iv \rangle \end{aligned}$$

FIR enables a generic and compact method for encoding loops as transition systems.

4.4 Proving Stuttering Simulation

We work with two transition systems TS^N and TS^A and compose them into a single transition system TS^{NA}. The main property of the joint transition system is that it captures that the two programs compute the same outputs, not only when they terminate but also at every step. To accomplish this, TS^{NA} is constructed from a cross-product of TS^N and TS^A where the TS^N is allowed to stutter when TS^A does not update output variables. The resulting system preserves equality of output variables at every step. We construct the joint system as follows:

$$\begin{aligned} V^{NA} &\equiv V^N, V^A \\ \Theta^{NA} &\equiv \Theta^N \wedge \Theta^A \\ \rho^{NA} &\equiv \rho^A_{body} \wedge ((\rho^N_{body} \wedge M = 0) \vee (\rho^N_\epsilon \wedge M > 0)) \end{aligned}$$

where M is a well-order used to ensure that TS^N is eventually forced a non-stuttering step. For a suitable M, the joint system has the property that every reachable state σ^{NA}_i of TS^{NA} is a composition of reachable states in the two original systems, and every transition $\sigma^{NA}_i, \sigma^{NA}_{i+1}$ is a step of TS^A and either a step in TS^N or an idling step. Conversely every pair of sequences σ^N, σ^A corresponds to a sequence σ^{NA} that can be mapped back to the original sequences modulo stuttering steps for σ^N. When the joint system has the property that it entails an invariant $W^A = W^N$ for output variables

For assertion $W^A = W^N$ over disjoint V^A, V^N,

$$\begin{array}{lll} \text{SR1:} & \Theta^{AN} & \implies I \\ \text{SR2:} & I \wedge \rho^{NA} & \implies I' \\ \text{SR3:} & I & \implies W^A = W^N \\ \text{SR4:} & I \wedge \rho^{NA} \wedge 0 < M & \implies 0 \leq M' < M \\ \text{SR5:} & I \wedge 0 = M \wedge En\left(body^N\right) & \implies En\left(body^A\right) \end{array}$$

TS^N and TS^A are stutter similar with respect to $W^A = W^N$.

Fig. 6. Stuttering Simulation Rule (SR)

$W \subseteq V$, we say that the two transition systems are *stutter similar* on the relation $W^A = W^N$. SpEQ uses the proof rule SR in Figure 6 to prove TS^N and TS^A are stutter similar.

Proposition 4.1 (Correctness of SR). *When there is an inductive invariant I and well order M satisfying the premises in Figure 6, then TS^A is stutter-similar to TS^N. Furthermore, when TS^N terminates, TS^A also terminates.*

Proof Outline. The argument is by induction on length of traces $\sigma^A, \sigma^N, \sigma^{NA}$. The premises SR1, SR2 establish that I is an inductive invariant over ρ^{NA}, which is either based on joint steps of the two systems or a step of TS^A and a stutter with respect to TS^N. Premise SR3 establishes that $W^A = W^N$ holds at every state and therefore after termination. Premise SR4 ensures that the joint system cannot be stuck in idling steps for TS^N, and SR5 establishes that when the well-order cannot be decreased, the traces of TS^N have a matching state in TS^A. □

We are left with the task of finding the inductive invariant I and measure M. This is where we will use the auxiliary predicates *Sync* and *NotSync*. For this purpose we use the following insights:

- The well order M can be extracted from the programs.
- The synchronization points, where the joint transition system performs non-idling steps, can be captured by a condition *Sync* that can be extracted from the programs.
- Similarly, non-synchronization is captured by a condition *NotSync*.
- The two conditions are used to define a transition abstraction $\rho^{AN\alpha} \equiv (Sync \wedge \rho^A_{body} \wedge \rho^N_{body}) \vee (NotSync \wedge \rho^A_{body} \wedge \rho^N_{\epsilon})$, which represents an over-approximation of ρ^{NA}, since it replaces the measure conditions on M by consequences.
- We can, optionally, also establish that $\rho^{AN\alpha}$ is an over-approximation of ρ^{NA} by checking $I \wedge \rho^{NA} \wedge 0 < M \implies NotSync$, and $I \wedge \rho^{NA} \wedge 0 = M \implies Sync$.

To synthesize an inductive invariant I we found that a single iteration of *back-propagation* [8, 25] applied on $\rho^{AN\alpha}$ is sufficient. Back-propagation is based on the *weakest liberal precondition* to derive a candidate I automatically. The weakest precondition for a transition relation ρ is shown in Equation 1. We can then define the candidate for I as:

$$I \equiv \mathcal{B}^1_{\rho^{AN\alpha}}(W^A = W^N) \equiv (W^A = W^N) \wedge wp(\rho^{AN\alpha}, W^A = W^N)$$

The *Sync* and *NotSync* predicates are extracted from the program relation formulated by SpEQ. Recall that the relation consists of two maps, `crd` and `val`. Figure 5c shows an example relation.

$$wp(\rho, \varphi)(V) \equiv \forall V' \cdot \rho(V, V') \implies \varphi(V') \tag{1}$$

$$\mathcal{B}^0_\rho(\varphi) \equiv \varphi, \quad \mathcal{B}^{n+1}_\rho(\varphi) \equiv \varphi \wedge wp(\rho, \mathcal{B}^n_\rho(\varphi)) \tag{2}$$

Fig. 7. Weakest liberal pre-conditions and back-propagation

$$\begin{array}{rcll}
\Theta^{AN} & \equiv & \Theta^N \wedge \Theta^A & \equiv & (\texttt{sum} = 0 \wedge \texttt{k} = \texttt{row[j]}) \wedge (\texttt{sum2} = 0 \wedge \texttt{k2} = 0) \\
 & & W^A = W^N & \equiv & \texttt{sum} = \texttt{sum2} \\
I & \equiv & \mathcal{B}^1_{\rho^{AN\alpha}}(W^A = W^N) & \equiv & (\texttt{sum} = \texttt{sum2}) \wedge (\forall V^{A'} V^{N'} \cdot \rho^{AN\alpha} \implies W^{A'} = W^{N'}) \\
Sync & \equiv & \bigwedge_{e \in \texttt{crd} \cup \texttt{val}} e & \equiv & \texttt{col[k]} = \texttt{k2} \wedge \texttt{a[k]} = \texttt{A[j*m + k2]} \\
NotSync & \equiv & \bigwedge_{v \in B} v = 0 & \equiv & \texttt{A[j*m + k2]} = 0 \\
M & \equiv & a - \texttt{crd}(a) & \equiv & \texttt{col[k]} - \texttt{k2}
\end{array}$$

Fig. 8. Instantiating rule SR for the system in the running example.

The *Sync* and *NotSync* predicates are generated from `val` and `crd`, where $\texttt{val} : A \rightarrow B$ and $\texttt{crd} : A \rightarrow B, |A| = |B| = 1$:

$$Sync \equiv \bigwedge_{e \in \texttt{crd} \cup \texttt{val}} e, \quad NotSync \equiv \bigwedge_{v \in B} v = 0$$

For example, the *Sync* predicate for the two codes in Figure 5c is `col[k] = k2∧a[k] = A[j*m + k2]`. The *NotSync* predicate for the two codes in Figure 5 is `A[j*m + k2] = 0`. The *measure* function, M, is computed from the `crd` map:

$$M \equiv a - \texttt{crd}(a), \qquad M' \equiv a' - \texttt{crd}(a)', \qquad a \in A$$

This is similar to the *ranking function* used in [38].

Condition SR4 ensures that the measure decreases at each non-synchronization step. In other words, as the sparse program skips zeros, the affine program is guaranteed to "catch up" eventually. Although $\texttt{k}' = \texttt{k}$ in a stutter transition for the sparse example, $\texttt{k2}' = \texttt{k2} + 1$ in the dense version, allowing the measure to decrease. Additionally, the two programs must rejoin again, at which point the distance is 0, shown in condition SR5 that is used to ensure that both programs can take a synchronization step.

Above, we have shown how auxiliary predicates and the measure predicate are generated algorithmically from information in `crd` and `val` to instantiate the proof rule SR. Figure 8 shows an example instantiation of the rule for the running example. Using an SMT solver each formula in the proof rule is checked for validity, dually their negations are checked for unsatisfiability. If all premises are valid, the proof rule establishes that $W^A = W^N$ holds in all reachable states of TS^{NA}. The equivalence checker module concludes that the programs are equivalent under the storage format preconditions. However, SpEQ can only know at run-time whether the preconditions are satisfied. The next section explains how SpEQ builds an executable check to verify that the required properties are satisfied at run-time.

4.5 Run-time Check

If the equivalence check succeeds, SpEQ needs to ensure that the input data at run-time satisfies the required storage format preconditions. By this stage, storage formats used in the input code are known, otherwise, the equivalence check would fail. Therefore, SpEQ generates a run-time check for the requirements of each format, which are well-studied in prior works [11].

Most common formats require a combination of two properties: *monotonicity* and *periodic monotonicity*. For example, the CSR storage format requires that elements of the `row` array increase monotonically. The `col` array stores column indices of elements in each row; therefore, regions of `col` corresponding to a row must strictly increase, referred to as *periodic monotonicity*. To generate a run-time check for a storage format, SpEQ emits a loop for each array with a required property (monotonic/periodic monotonic), in the same loop order as the input code.

The running example in Figure 1d shows the run-time check that SpEQ generates for CSR. The outer loop ensures that `row` elements are increasing (monotonic); the inner loop ensures that `col` elements are increasing within a row (periodic monotonicity). SpEQ also uses the run-time check to calculate the number of rows or columns in a sparse matrix, which is required by some backends.

5 IDENTIFY KERNELS

The fundamental goal of SpEQ is to detect a code's semantics. By understanding what the code "does," it can be replaced with high-performance library calls or DSL code that does better optimization than a general-purpose compiler. Searching for specific code patterns (idiom matching) is one strategy to determine code semantics, but it is a fragile approach that covers a narrow notion of equivalent semantics. On the other hand, synthesizing a code's semantics from scratch (verified lifting) is an expensive process.

Our insight is that detecting code semantics is only useful when an optimized implementation exists. In other words, we do not care about *deriving* the semantics of arbitrary codes; rather, we only want to know one code (input code) is equivalent to another (high-performance implementation). SpEQ casts this equivalence problem as a congruence relation between programs. Therefore, instead of deriving code semantics from scratch, we can leverage recent advancements to e-graphs, which efficiently represent congruence relations over expressions [55, 56]. We first explain equality saturation and e-graphs, and then describe how SpEQ uses equality saturation to match input code with known reference implementations.

5.1 Equality Saturation

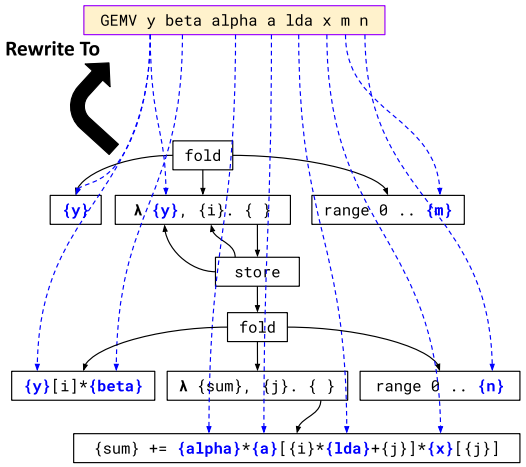
Basic term rewriting applies a single rewrite rule at a time, which destroys the original term and makes the process sensitive to rewrite order (*phase-ordering* problem). Equality Saturation (EqSat) [51] addresses the phase-ordering problem by performing all possible rewrites at each step and storing each original and new term in an *e-graph* data structure. Recently, libraries such as `egg` and `egglog` provide flexible and efficient methods of performing equality saturation [55, 56]. An e-graph [39] efficiently represents a congruence relation over many expressions, which enables the large set of terms created by EqSat to be stored compactly. E-graphs are a set of *e-classes*, which contain *e-nodes* of the same equivalence class. An e-node is a function symbol with a list of children e-classes as the function operands.

Consider the expression $a \times 0$. To apply a rewrite rule, for example $x \times 0 \rightarrow 0$, matches for the left-hand side pattern term p are found through *e-matching* [16, 18, 56]. The result is a set of substitutions, for example, $\{x \mapsto a\}$, such that p would be represented in the same e-class as $a \times 0$. The substitutions are applied to the right-hand side pattern, $x \times 0 \mapsto a \times 0$, and the resulting term is merged into the relevant e-class.

SpEQ relies on e-matching to identify kernels in user input code (hereafter, a kernel refers to a linear algebra subprogram, e.g. matrix-vector multiplication [9]). For example, given a rewrite rule $g \rightarrow$ **GEMV**$(\dots)$, where g is a pattern representing a GEMV implementation, segments of user code that match g will be merged into the same e-class as GEMV. However, kernels are provided to SpEQ as concrete implementations, not patterns. The next section describes how concrete implementations are converted into patterns that can be used for e-matching.

```
void gemv(double *y,
          double beta,
          double alpha,
          double *a,
          int lda,
          double *x,
          int m,
          int n) {
  for (int i = 0; i < m; ++i) {
    double sum = y[i]*beta;
    for (int j = 0; j < n; ++j)
      sum += alpha * a[i*lda + j] * x[j];
    y[i] = sum;
  }
}
```

(a) The GEMV reference implementation used by SpEQ.



(b) The rewrite rule generated from the GEMV reference implementation 9a.

```
for (int i = 0; i < m; ++i) {
  y[i] = y[i]*beta;
  for (int j = 0; j < n; ++j)
    y[i] += alpha * a[i*lda + j] * x[j];
}
```

(c) The reference implementation 9a loop nest after applying the *store-sink* rewrite rule.

Fig. 9. The GEMV reference implementation 9a is abstracted into the rewrite rule 9b. From the reference implementation, SpEQ also generates and abstracts additional forms, such as 9c, to support "one-way" rules like hoisting stores out of loops.

5.2 Abstracting Implementations to Rewrite Rules

SpEQ automatically converts concrete kernel implementations into rewrite rules that are used during EqSat. An input kernel implementation is a function in LLVM IR and then converted to the functional IR. The function name, input parameters, and return type are extracted from the LLVM IR. The function parameters define which program variables should be symbolic and converted to placeholder variables in the pattern.

Figure 9 shows an example reference implementation for GEMV (Figure 9a) and its corresponding rewrite rule after abstraction (Figure 9b). Given a reference implementation, the rewrite rule $l \longrightarrow r$ is constructed as follows. First, to obtain l (bottom of Figure 9b), SpEQ converts the reference implementation to FIR. Each occurrence of a function parameter in the FIR becomes a variable placeholder (blue curly-braces) for e-matching. Then, r is a fresh function symbol with the same name as the reference implementation and with each variable placeholder as an input argument (top of Figure 9b). The resulting rewrite rule corresponds to collecting the matching arguments from the user input code and outlining it as a call to the kernel. In practice, user input code will not exactly match the structure of the rewrite rule generated by a kernel implementation. To be robust in such cases, SpEQ also includes rewrite rules for common arithmetic and loop transformations. Table 1 shows a list of generic rewrite rules used by SpEQ for deoptimization. While formally verifying the soundness of each rule is possible, it is not the main focus of this paper, and we leave it for future work. In practice, no incorrect translations have occurred while evaluating SpEQ.

Transformation	Rule
Sink	$op \in$ commutative, associative, then: $op(a, \textbf{if } c \textbf{ then } x \textbf{ else } y) \rightarrow \textbf{if } c \textbf{ then } op(a,x) \textbf{ else } op(a,y)$ $op(a, \textbf{fold } ret\ c\ start\ lb\ ub) \rightarrow \textbf{fold } ret\ c\ op(a, start)\ lb\ ub$
Arithmetic	$a+0 \rightarrow a,\ a \cdot 0 \rightarrow 0,\ a+0 \rightarrow a,\ a < b \rightarrow b > a, \ldots$ $op(a,b) \rightarrow op(b,a), \qquad op \in$ commutative
Loop Interchange	c has no dependencies, then: $\textbf{fold } ret_1\ (\lambda\ params_1, i_1.\ \textbf{fold } ret_2\ c\ start_2\ lb_2\ ub_2)\ start_1\ lb_1\ ub_1 \rightarrow$ $\textbf{fold } ret_1\ (\lambda\ params_1, i_2.\ \textbf{fold } ret_2\ c\ start_2\ lb_1\ ub_1)\ start_1\ lb_2\ ub_2$
Loop Fission	$\textbf{fold } \langle r_1, \ldots, r_n \rangle\ c\ s\ lb\ ub \rightarrow \textbf{let } r_1 = \textbf{fold } r_1\ c\ s\ lb\ ub \textbf{ in let } r_2 = \ldots$
Merge	$s_1 = s_2 \wedge i =$ outer fold induction variable, then: $\textbf{fold } r\ (\textbf{fold } r\ c\ s_2\ i\ \textbf{min}(ub, i+B))\ s_1\ lb\ ub \rightarrow \textbf{fold } r\ c\ s_1\ lb\ ub$

Table 1. A subset of the generic rewrite rules used by SpEQ.

It is a well-known problem that certain "one-way" loop transformations are difficult to describe through rewrite rules [27, 55]. For example, consider the two program versions in Figure 9a and Figure 9c. Rather than performing a store at each loop iteration, it is more efficient to accumulate a scalar result and only perform one store if possible. Therefore, for the purposes of optimization, it is better to rewrite Figure 9c to Figure 9a. Performing transformation in this direction requires introducing a new variable `sum` in the right-hand side of a rewrite rule, which is not possible. However, it is possible to express in the opposite direction (9a to 9c). Therefore, as an initial step, SpEQ applies such "one-way" rules to the reference implementation and obtains a class of equivalent reference implementations. Each one is abstracted and added as a rewrite rule.

User codes often omit parameters that general-purpose reference implementations do not. Consider the `alpha` and `beta` parameters of the GEMV reference implementation (Figure 9a). Commonly, user codes implicitly set `alpha=1` and `beta=0` by omitting `alpha` and overwriting the output vector. In order to recover the values of these parameters from the input code, SpEQ also includes a set of "expansion" rewrite rules. One example is $a \rightarrow a \times 1$. Because these rules can cause exponential growth in the e-graph, SpEQ takes advantage of `egglog`'s scheduling API to limit how often the rules are applied. Empirically, we have found that a single application of the expansion rules is sufficient to discover implicit parameters in the chosen benchmarks (Section 6).

5.3 Code Generation

After each kernel implementation is translated into a rewrite rule, SpEQ runs EqSat until the e-graph is fully saturated or a configurable timeout is reached. It is the job of each backend to decide which kernels to translate and how to translate them. In the next section, we present three backend examples: OMP, cuSPARSE, and Intel MKL [6, 12, 36].

6 EXPERIMENTAL EVALUATION

This experimental evaluation assesses (1) the robustness of SpEQ's matching technique compared to KernelFaRer and LiLAC (Section 6.2); (2) the performance impact from replacing user input code with high-performance library calls and the related run-time check overhead (Section 6.3); and (3) the effect of equivalence checks and EqSat on compilation time (Section 6.4).

SpEQ is implemented using LLVM 17 [28], the Z3 SMT solver [17], and the `egglog` equality saturation library [56]. An initial LLVM pass performs format detection and emits the input code as FIR program f. Next, a Python script parses the FIR, verifies any proposed precondition-free loops, and replaces the original loop if the equivalence check succeeds, resulting in f'. Finally, SpEQ adds f' to an e-graph and uses the `egglog` Python interface to perform equality saturation. Bottom-up extraction is used to retrieve translated kernels from the e-graph.

6.1 Experimental Setup

This section presents the experimental setup, methodology, benchmarks, and configuration of all tools used in the evaluation.

6.1.1 Architecture. All experiments are run on an Intel Xeon W-2145 CPU and NVIDIA GeForce RTX 3080 Ti GPU. Hyperthreading is disabled, leaving the CPU with a total of eight cores.

6.1.2 Prior Work. We compare SpEQ against KernelFaRer and LiLAC, which are two state-of-the-art work based on idiom matching. All three tools are based on LLVM. We build the LiLAC Clang executable using repositories shared with us by the authors (release build).[1,2] We build the KernelFaRer Clang executable using the repository cited in their paper [15].[3]

6.1.3 Compilation Pipeline. Each tool has a differing compilation pipeline for transforming the LLVM IR into an expected form. For example, LiLAC uses the LLVM version 7.0.0 O3 pipeline with loop unrolling disabled. KernelFaRer uses the LLVM version 15.0.6 O3 pipeline, with loop unrolling and vectorization disabled. SpEQ uses a different pipeline based on LLVM version 17.0.0. Therefore, for a fair comparison of compilation overhead, we only time the compiler passes that are specific to each technique. LiLAC has three relevant passes: `preprocessor`, `flangfix`, and `replacer`. KernelFaRer has one relevant pass: `GEMMReplacerPass`. SpEQ has one relevant LLVM pass, `SpEQPass`, for emitting FIR, along with the `EqCheck` and `EqSat` passes. The compilation overhead of each tool is the sum of its relevant passes.

6.1.4 Backends. This evaluation uses the following backends for SpEQ: Intel MKL (CPU) [36], OpenMP (CPU) [6], and NVIDIA cuSPARSE (GPU) [12]. The MKL and OpenMP libraries are called in 8 threads to use all the cores.

6.1.5 Kernels. Table 2 shows the four kernels used in the evaluation: GEMM, SpMV, histogram, and reduction. All four computations are often compute bottlenecks in applications such as scientific computing and machine learning. GEMM is a dense matrix-matrix multiplication that does not require a run-time check. SpMV is a sparse matrix-vector multiplication that does require a run-time check. The TACO-Gen and CSparse benchmarks contain an SpMV using the CSC storage format, while the Scimark4, NPB CG, and Netlib C benchmarks use the CSR format. Histogram contains indirect memory accesses but also does not require a run-time check. The Reduction kernel reduces repeated associative operations into a scalar result.

6.1.6 Benchmarks. We select ten benchmarks to evaluate the SpEQ implementation: Scimark 4 [42], Conjugate Gradient (CG) and Integer Sort (IS) kernel from NAS Parallel Benchmarks (NPB) [32], Netlib sparse benchmark suites [19], Polybench [41], Parboil [49], TPAL [45], CSparse [14], a sparse kernel generated from the TACO compiler [26], and TSVC2 [33]. We report the median of 10 executions for each benchmark. Each execution time is normalized to the sequential benchmark execution time.

[1] https://github.com/ginsbach/llvm/tree/linearalgebra
[2] https://github.com/ginsbach/clang/tree/research
[3] https://github.com/jaopaulolc/KernelFaRer

Benchmark	Computation	Kernel
Polybench	GEMM	GEMM
TACO-Gen	SpMV+CSC	GEMV
CSparse	SpMV+CSC	GEMV
Scimark4	SpMV+CSR	GEMV
NPB CG	SpMV+CSR	GEMV
Netlib C	SpMV+CSR	GEMV
TPAL	SpMV+CSR	GEMV
NPB IS	Histogram	Histogram
Parboil	Histogram	Histogram
TSVC2	Reduction	Reduction

Table 2. The different kernels present in each selected benchmark.

6.1.7 Timing Methodology. To guarantee a fair comparison against KernelFaRer and LiLAC, the same input codes are compiled. Additionally, we collect the time results of the loop passes added by prior tools and also collect the compile-time of O3 optimization pipeline as a reference.

Compile-time overhead for all tools is measured using the `-ftime-report` compilation flag, which emits the time spent in each pass during compilation to standard output. Compile-time overhead is the sum of time spent in each tool's relevant passes (Section 6.1.3). Because SpEQ has two passes that are not implemented in LLVM, they are timed using the Python `time` module.

6.2 Robustness

In this section, we evaluate the ability of SpEQ, KernelFaRer, and LiLAc to detect different implementations of the same computation. Table 3 shows the four different computations used in this evaluation and the ability of each tool to detect it.

While SpEQ and KernelFaRer both can detect the GEMM in the Polybench suite, LiLAC cannot, despite searching for a GEMM idiom. This is because the GEMM idiom that LiLAC uses for detection assumes that the inner-most loop (dot product) accumulates into a scalar variable. However, the GEMM in Polybench accumulates directly into the destination matrix.

SpEQ and LiLAC both focus on sparse linear algebra, while KernelFaRer focuses on dense linear algebra. SpEQ is able to detect and derive preconditions for SpMV using both CSC and CSR, in all benchmarks. However, LiLAC can only detect SpMV using CSR, and not CSC. This is because LiLAC can only support idioms that express loops performing a reduction. Supporting other structures, such as the indirect memory store in SpMV using CSC, is not possible in the current LiLAC idiom grammar. In contrast, SpEQ can recognize SpMV using CSR and CSC as two implementations of GEMV by applying the generic rewrite rules shown in Table 1 during equality saturation.

SpEQ and LiLAC are able to detect histograms in the Parboil and NPB IS benchmarks. However, LiLAC relies on a special description of histogram in its domain specific language, totaling 320 lines. In contrast, SpEQ's description of histogram is 4 lines, shown below:

```
void histogram(int N, int *buckets, int *key, int add) {
  for (int i = 0; i < N; ++i)
    buckets[key[i]] = buckets[key[i]] + add;
}
```

Tool	GEMM	GEMV		Histogram	Reduction
		SpMV+CSC	SpMV+CSR		
SpEQ	•	•	•	•	•
KernelFaRer	•				
LiLAC			○	•	•

Table 3. Comparison of code semantics detection tools for different computations. The solid dot (•) denotes correct detection. The circle (○) denotes unsafe detection (preconditions ignored). No circle or dot denotes no detection.

This description is flexible enough to capture the control-flow in Parboil's histogram, where `buckets[key[i]]` is conditionally incremented. Because FIR uses functional **`if`** constructs, which reduce to a value, the `add` argument is guarded by the relevant condition.

6.3 Performance Impact

Figure 10 shows the potential performance benefits when SpEQ translates the input codes to the chosen backends. The run-time check overhead is included as a percentage of execution time.

When the run-time check is executed inside a loop where its inputs do not change, such as inside a conjugate gradient loop, the run-time check is loop-invariant and hoisted outside the loop to amortize overhead. We have divided the attained results into two sub-figures as depicted in Figure 10 based on whether run-time check overhead is amortized or not. The left sub-figure illustrates that the run-time check overhead is amortized by executing the translated code for numerous iterations. The right sub-figure shows a slow-down for two reasons: (1) the translated code is only executed once; therefore, the run-time check overhead is not amortized, and (2) CSC prevents outer-loop parallelism. As a result, SpEQ-OpenMP only parallelizes the inner loop of SpMV CSC, which causes synchronization overhead during each outer loop iteration.

The histogram computations in NPB IS and Parboil, as well as the reduction computation in TSVC2, do not require a run-time check. However, the histogram in Parboil contains a conditional update, which introduces additional synchronization overhead. Therefore, the OpenMP backend disables parallelism, and the OpenMP execution time is the same as the sequential time. In contrast, the histogram in NPB IS and reduction in TSVC2 are parallelized efficiently using the OMP `reduction` pragma. Because MKL and cuSPARSE do not implement histograms or reductions, the corresponding bars of NPB IS, Parboil, and TSVC2 are not included.

In the benchmark code, SpEQ demonstrates a geometric mean speedup of 3.25×, 5.09× and 8.04× for OpenMP, MKL, and cuSPARSE, respectively. The run-time check overhead constitutes, on average, 3.6, 6.1 and 4.2 percent of the execution time of the entire benchmark.

6.4 Compilation Time

In this section, SpEQ's compile-time overhead is compared to KernelFaRer [15] and LiLAC [21], as shown in Table 4. For reference, LLVM's O3 pass compile-time latency is also shown.

SpEQ's translation workflow comprises three phases: FIR, Parse + EqCheck, and EqSat. The latter two phases exhibit much longer processing time compared to the first phase because they are implemented in Python for proof of concept. Despite this, SpEQ's compilation time is at most 190 milliseconds slower than LLVM's O3 compilation time, which is implemented in highly optimized and multithreaded C++ code.

On average, e-graphs contain 157 nodes after applying rewrite rules; the largest e-graph is Parboil, with 277 nodes. At most 46 rules are applied per round.

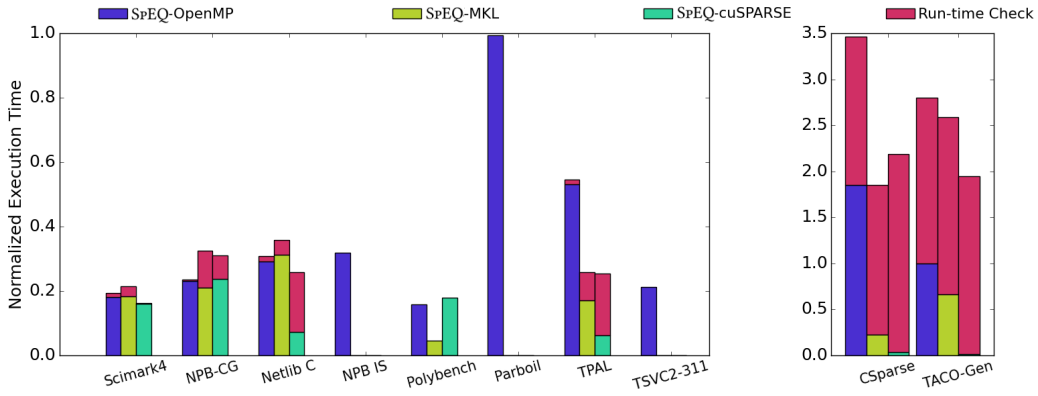


Fig. 10. The performance of SpEQ-generated code. Execution times are normalized to CPU sequential code running times.

		SpEQ					
Benchmarks	LLVM + O3	FIR	Parse + EqCheck	EqSat	Total	KernelFaRer	LiLAC
Polybench	60.9	2.6	80.1	14.0	97.5	220.0	217.4
TACO-Gen	16.9	4.6	193.3	9.4	207.4	5.6	4.0
CSparse	15.0	2.8	89.1	9.9	101.8	5.8	4.0
Scimark4	20.2	3.4	85.2	14.7	103.3	146.6	6.3
NPB CG	16.7	6.5	99.3	91.7	114.9	5.9	4.3
Netlib C	23.3	2.9	119.0	9.1	131.0	5.3	4.1
TPAL	13.9	2.7	94.7	9.2	106.5	5.2	7.4
NPB IS	13.2	1.1	13.4	3.0	17.5	5.5	4.0
Parboil	24.5	2.7	44.5	5.6	52.7	5.6	5.1
TSVC2	29.2	0.8	19.2	3.7	23.7	3.5	5.0

Table 4. Comparison of translation time (in milliseconds).

7 RELATED WORK

Idiom Matching. Prior works such as IDL and LiLAC implement idiom matching by solving a system of constraints [21, 22]. Each idiom is associated with certain constraints, and code regions satisfying those constraints are recognized as an idiom. The solution to the constraint system is used to perform translation. KernelFaRer is another idiom matching tool that uses pattern-matching instead of a constraint solver [15]. ATC uses a combination of synthesis, machine learning, and testing to find candidate code regions [35]. KernelFaRer and ATC focus on dense linear algebra, while IDL and LiLAC aim to support both dense and sparse codes.

Verified Lifting. Verified Lifting techniques typically translate codes through search: given the user program, a translation tool searches for a semantically equivalent program in the target DSL, using program synthesis and formal verification to find solutions [3, 24]. Unlike idiom matching, the goal is not to identify a particular computational pattern, but rather find any mapping from one language to another. Because verified lifting uses formal verification techniques, any translation is guaranteed to be correct, meaning the original and translated program are semantically equivalent.

Functional Intermediate Representations. Prior works have explored functional encodings for other languages. Appel showed a correspondence between functional programming and SSA, with loops defined through recursive function calls [5]. Radoi et al. propose an IR for Java based on the lambda calculus [44]. Several lambda calculus-based encodings are also proposed for Rise and Lift programs [23, 47, 48]. MLIR uses a recursive structure that is similar to the value-based design of FIR [29]. The main contribution of SpEQ is not a functional-based IR, but rather a strategy of program analysis. Therefore, we view IRs and languages such as MLIR or Rise as possible targets for support through additional backends.

Verifying and Optimizing Sparse Programs. Inspector/executor strategies are a common program transformation to optimize sparse or non-affine programs through dynamic parallelism [50]. Prior works leverage storage format properties, such as monotonicity and periodic monotonicity, to simplify inspector codes [37]. In addition, other works verify the inspector codes for sparse computations [40]. In addition to inspector/executor techniques, other works also extend the polyhedral model to optimize non-affine codes [52–54]. Finally, Dyer et al. propose a functional language for the bounded verification of sparse matrix computations [7, 20].

8 CONCLUSION

SpEQ uses a novel, fast, and flexible strategy for identifying preconditions in sparse linear algebra codes and detecting semantics in programs. Any preconditions due to compressed storage formats in sparse linear algebra codes are verified through a run-time check. Users can easily add new code semantics to SpEQ's analysis by supplying a reference implementation, rather than using a complicated and non-portable API. Code generation for different high-performance libraries or DSLs is supported through modular *backends*. SpEQ's novel precondition-free transformation and rewrite rule *abstraction* process enable quick and correct semantics detection.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers, as well as Larry Shi for his valuable help. This work was also supported in part by NSERC Discovery Grants (RGPIN-06516, DGECR00303), the Canada Research Chairs program, Ontario Early Researcher award, the Canada Research Chairs program, the Ontario Early Researcher Award, and the Digital Research Alliance of Canada (www.alliancecan.ca).

REFERENCES

[1] 2023. MemorySSA. https://llvm.org/docs/MemorySSA.html. Accessed: 2023-11-13.
[2] Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically leveraging mapreduce frameworks for data-intensive applications. In *Proceedings of the 2018 International Conference on Management of Data*. 1205–1220.
[3] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. 2019. Automatically translating image processing libraries to halide. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–13.
[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.
[5] Andrew W Appel. 1998. SSA is functional programming. *Acm Sigplan Notices* 33, 4 (1998), 17–20.
[6] OpenMP ARB. 2023. OpenMP. https://www.openmp.org/.
[7] Gilad Arnold, Johannes Hölzl, Ali Sinan Köksal, Rastislav Bodík, and Mooly Sagiv. 2010. Specifying and verifying sparse matrix codes. *ACM Sigplan Notices* 45, 9 (2010), 249–260.
[8] Nikolaj Bjørner, Anca Browne, and Zohar Manna. 1997. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science* 173, 1 (1997), 49–87.
[9] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.
[10] Alvin Cheung. 2023. MetaLift. https://github.com/metalift/metalift.

[11] Stephen Chou. 2022. *Format Abstractions for the Compilation of Sparse Tensor Algebra*. Ph. D. Dissertation. Massachusetts Institute of Technology.
[12] cuSPARSE [n. d.]. Basic Linear Algebra for Sparse Matrices on NVIDIA GPUs. https://developer.nvidia.com/cusparse.
[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 25–35. https://doi.org/10.1145/75277.75280
[14] Timothy A Davis. 2006. *Direct methods for sparse linear systems*. SIAM.
[15] Joao PL De Carvalho, Braedy Kuzma, Ivan Korostelev, José Nelson Amaral, Christopher Barton, José Moreira, and Guido Araujo. 2021. KernelFaRer: replacing native-code idioms with high-performance library calls. *ACM Transactions On Architecture And Code Optimization (TACO)* 18, 3 (2021), 1–22.
[16] Leonardo De Moura and Nikolaj Bjørner. 2007. Efficient E-matching for SMT solvers. In *Automated Deduction–CADE-21: 21st International Conference on Automated Deduction Bremen, Germany, July 17-20, 2007 Proceedings 21*. Springer, 183–198.
[17] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, 337–340.
[18] David Detlefs, Greg Nelson, and James B Saxe. 2005. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)* 52, 3 (2005), 365–473.
[19] Jack Dongarra, Victor Eijkhout, and Henk van der Vorst. 2001. An iterative solver benchmark. *Scientific Programming* 9, 4 (2001), 223–231.
[20] Tristan Dyer, Alper Altuntas, and John Baugh. 2019. Bounded verification of sparse matrix computations. In *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 36–43.
[21] Philip Ginsbach, Bruce Collie, and Michael FP O'Boyle. 2020. Automatically harnessing sparse acceleration. In *Proceedings of the 29th International Conference on Compiler Construction*. 179–190.
[22] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael FP O'Boyle. 2018. Automatic matching of legacy code to heterogeneous APIs: An idiomatic approach. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 139–153.
[23] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29.
[24] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. *ACM SIGPLAN Notices* 51, 6 (2016), 711–726.
[25] Shmuel Katz and Zohar Manna. 1976. Logical analysis of programs. *Commun. ACM* 19, 4 (1976), 188–206.
[26] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. Taco: A tool to generate tensor algebra kernels. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 943–948.
[27] Thomas Koehler, Phil Trinder, and Michel Steuwer. 2022. Sketch-Guided Equality Saturation.
[28] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*. San Jose, CA, USA, 75–88.
[29] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.
[30] LCSSA 2023. Loop Closed SSA (LCSSA). https://llvm.org/docs/LoopTerminology.html#loop-closed-ssa-lcssa.
[31] LoopSimplify 2023. Loop Simplify Form. https://llvm.org/docs/LoopTerminology.html#loop-simplify-form.
[32] Júnior Löff, Dalvan Griebler, Gabriele Mencagli, Gabriell Araujo, Massimo Torquati, Marco Danelutto, and Luiz Gustavo Fernandes. 2021. The NAS Parallel Benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems* 125 (2021), 743–757. https://doi.org/10.1016/j.future.2021.07.021
[33] Saeed Maleki, Yaoqing Gao, Maria J. Garzar´n, Tommy Wong, and David A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 372–382. https://doi.org/10.1109/PACT.2011.68
[34] Zohar Manna and Amir Pnueli. 2012. *Temporal verification of reactive systems: safety*. Springer Science & Business Media.
[35] Pablo Antonio Martínez, Jackson Woodruff, Jordi Armengol-Estapé, Gregorio Bernabé, José Manuel García, and Michael FP O'Boyle. 2023. Matching linear algebra and tensor code to specialized hardware accelerators. In *Proceedings*

of the 32nd ACM SIGPLAN International Conference on Compiler Construction. 85–97.

[36] MKL [n. d.]. Intel® oneAPI Math Kernel Library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

[37] Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C Davis, Mary Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, Anand Venkat, and Michelle Mills Strout. 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*. 594–609.

[38] Kedar S Namjoshi and Lenore D Zuck. 2013. Witnessing program transformations. In *Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings 20*. Springer, 304–323.

[39] Charles Gregory Nelson. 1980. *Techniques for program verification*. Stanford University.

[40] Michael Norrish and Michelle Mills Strout. 2015. An approach for proving the correctness of inspector/executor transformations. In *Languages and Compilers for Parallel Computing: 27th International Workshop, LCPC 2014, Hillsboro, OR, USA, September 15-17, 2014, Revised Selected Papers 27*. Springer, 131–145.

[41] Louis-Noel Pouchet and Tomofumi Yuki. 2019. Polyhedral Benchmark suite. http://polybench.sf.net/.

[42] Roldan Pozo. 2000. SciMark 2.0. *http://math.nist.gov/scimark2/* (2000).

[43] Reese T. Prosser. 1959. Applications of Boolean Matrices to the Analysis of Flow Diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference* (Boston, Massachusetts) *(IRE-AIEE-ACM '59 (Eastern))*. Association for Computing Machinery, New York, NY, USA, 133–138. https://doi.org/10.1145/1460299.1460314

[44] Cosmin Radoi, Stephen J Fink, Rodric Rabbah, and Manu Sridharan. 2014. Translating imperative code to MapReduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. 909–927.

[45] Mike Rainey, Kyle Hale, Ryan R. Newton, Nikos Hardavellas, Simone Campanoni, Peter Dinda, and Umut A. Acar. 2021. Task Parallel Assembly Language for Uncompromising Parallelism. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '21)*. ACM, New York, NY, USA. http://mike-rainey.site/papers/tpal-long.pdf

[46] Davide Sangiorgi. 2011. *Introduction to bisimulation and coinduction*. Cambridge University Press.

[47] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. *ACM SIGPLAN Notices* 50, 9 (2015), 205–217.

[48] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 74–85.

[49] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012), 27.

[50] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proc. IEEE* 106, 11 (2018), 1921–1934.

[51] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 264–276.

[52] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and data transformations for sparse matrix code. *ACM SIGPLAN Notices* 50, 6 (2015), 521–532.

[53] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. 2016. Automating wavefront parallelization for sparse matrix computations. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 480–491.

[54] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. 2014. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 185–194.

[55] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (jan 2021), 29 pages. https://doi.org/10.1145/3434304

[56] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proc. ACM Program. Lang.* 7, PLDI, Article 125 (jun 2023), 25 pages. https://doi.org/10.1145/3591239

Received 16 November 2023; revised 26 March 2024; accepted 1 April 2024