Latent Idiom Recognition for a Minimalist Functional Array Language using Equality Saturation

Jonathan Van der Cruysse
McGill University
Montreal, Quebec, Canada
jonathan.vandercruysse@mail.mcgill.ca

Christophe Dubach
McGill University & Mila
Montreal, Quebec, Canada
christophe.dubach@mcgill.ca

***Abstract*—Accelerating programs is typically done by recognizing code idioms matching high-performance libraries or hardware interfaces. However, recognizing such idioms automatically is challenging. The idiom recognition machinery is difficult to write and requires expert knowledge. In addition, slight variations in the input program might hide the idiom and defeat the recognizer.**

This paper advocates for the use of a minimalist functional array language supporting a small, but expressive, set of operators. The minimalist design leads to a tiny sets of rewrite rules, which encode the language semantics. Crucially, the same minimalist language is also used to encode idioms. This removes the need for hand-crafted analysis passes, or for having to learn a complex domain-specific language to define the idioms.

Coupled with equality saturation, this approach is able to match the core functions from the BLAS and PyTorch libraries on a set of computational kernels. Compared to reference C kernel implementations, the approach produces a geometric mean speedup of 1.46× for C programs using BLAS, when generating such programs from the high-level minimalist language.

***Index Terms*—equality saturation, functional programming, array programming, pattern matching, libraries**

I. Introduction

Generating high-performance code for today's heterogeneous specialized hardware is challenging. A promising approach is to automatically rewrite specific idioms found in programs as highly optimized library calls [3, 5]. This decouples the compiler's pattern recognition from the hardware-specific knowledge embedded in the library implementation.

However, the library is only useful when idioms are found. Most prior work [3, 5, 8] encodes idioms in the compiler at a low level of abstraction. Crafting low-level idioms is tedious, requiring expert compiler knowledge and dedicated analysis passes. Furthermore, recognition might fail in response to minor changes to the input program. In short, idiom recognition faces two challenges: to specify idioms in a high-level language and to be robust to minor changes to the input program.

To solve the first challenge, this paper proposes to express both programs and idioms using a high-level functional array language. Functional array programming for high-performance computing has become increasingly popular in recent years [6, 16, 19] and the concise, high-level nature of functional languages simplifies code pattern detection and rewriting [7].

In a functional language, the second challenge of robust pattern detection amounts to finding hidden idioms. Consider the following vector sum program: `sum(v)` = `fold (+) 0 v`. If we have at our disposal a library function that can quickly perform such a sum, then we rewrite the program as a call to that function. However, if the library supports more general primitives, the rewriting problem becomes more complicated.

Suppose the library has a function **`dot`** that performs a dot product and a function **`fill`** that creates an array of identical elements. A human could combine both to implement the vector sum: `sum(v)` = **`dot`**`(v,` **`fill`**`(1))`. A pattern matcher does not have human intuition and would be hard-pressed to find this solution as neither the idiom corresponding to **`dot`** nor that for **`fill`** appear in the original program.

This paper proposes to find such intuitive solutions using Latent Idiom Array Rewriting (LIAR), a trustworthy technique that finds and exploits *latent* idioms using equality saturation. Equality saturation [20] discovers all possible program variants encoded as a finite data structure by applying rewrite rules until a fixed point is reached. LIAR relies on a minimalist Intermediate Representation (IR) based on a few simple functional programming primitives, resulting in a compact set of rewrite rules. This makes it easier to capture the essential structure of a program without getting bogged down in language-specific details. By using equality saturation to apply rewrite rules that capture both the library-independent semantics of the IR and library-specific idioms, LIAR efficiently transforms programs to expose hidden idioms and improve program efficiency.

This paper shows how this technique can be applied on a subset of the Basic Linear Algebra Subprograms (BLAS) [1] and PyTorch [13] libraries. To evaluate LIAR's effectiveness, this work applies it to custom kernels, and to linear algebra and numerical simulation kernels from the PolyBench suite. The results show that LIAR leads to significant improvements in program efficiency. Overall, this work demonstrates that equality saturation is a powerful tool for idiom recognition, and that LIAR can be easily adapted to different libraries by providing appropriate idiom descriptions for those libraries.

To summarize, this paper makes the following contributions:

- Presents a minimalist IR and its tiny subset of rewrites suitable for capturing the language semantics;
- Shows how this minimalist IR can be used to express idioms found in BLAS and PyTorch;
- Demonstrates the effectiveness of using equality saturation with a minimalist IR on a set of computational kernels.

979-8-3503-9509-9/24 © 2024 IEEE

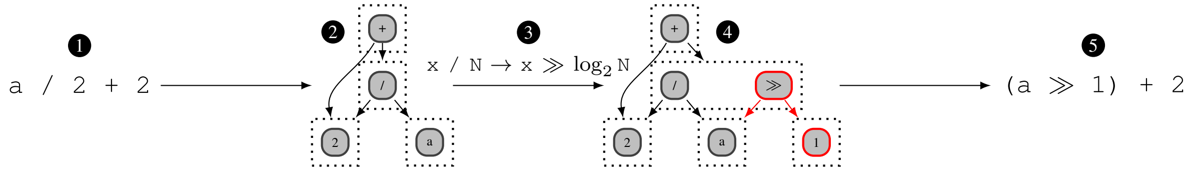


Fig. 1. Expression ❶ is converted to e-graph ❷, which is subsequently saturated. In this example, only rule ❸ is applied: x / N → x ≫ $\log_2$ N. From saturated e-graph ❹, expression ❺ is selected by an extractor that prefers bitwise shift.

The rest of this paper is organized as follows: Section II introduces equality saturation while section III provides an overview of the technique proposed. Section IV introduces the minimalist IR and the core rewrite rules. Section V shows two use-cases based on BLAS and PyTorch. Section VI evaluates the approach. Finally section VII discusses related work and section VIII concludes.

II. Background: Equality Saturation

Equality saturation is a rule-based rewriting algorithm that explores all variants [20] of an input program. These variants arise from applying rewrite rules as a fixed-point iteration on a dedicated data structure: the e-graph. Once the e-graph is built, a performance model extracts a single expression.

e-Graphs: Equality saturation engines store program variants in a specialized data structure called an e-graph. Each e-graph consists of a set of e-nodes and an equivalence relation that partitions the nodes into e-classes. e-Nodes have e-classes as children, allowing each e-node and e-class to represent a possibly unbounded set of expressions.

Encoding a program as an e-graph is straightforward. Each expression node becomes an e-node and each unique e-node is placed in its own e-class. This is illustrated in fig. 1 for expression ❶, and its corresponding e-graph ❷. Multiple occurrences of 2 are represented by multiple incoming edges.

Saturation: Once an expression has been converted to an e-graph, the graph is saturated. Saturation repeatedly applies rewrite rules to all eligible nodes in the graph. Rule application can be performed in any order, but is more efficient when performed in batch [24]. A batch consists of all currently possible rewrites. Applying such a batch is referred to as a *saturation step* or simply *step* in this paper.

In the simplest of cases, a step consists of a single rewrite rule application. This is the case in fig. 1, where e-graph ❷ is expanded by applying rule ❸: x/N → x ≫ $\log_2$ N, where ≫ represents the right-shift operator.

Assuming the set of rules only contains this one rule, there are no further possible applications of this rule in e-graph ❹; hence the e-graph is said to have reached a fixpoint. Such a fixpoint may not always exist, in such cases standard practice is to terminate equality saturation using a timeout [11].

Extraction: After reaching either a fixpoint or timeout, e-graph ❹ is ready for extraction. Extraction reduces an e-class or e-node to a single expression. This single output expression corresponds to a walk through the e-graph, starting from an e-node or e-class. Each time the traversal encounters an e-class, a single e-node is selected from that e-class.

To guide the walk through the e-graph, one typically looks for the best expression in the e-class or e-node, based on some measure of quality using a cost function. To illustrate the use of a cost function, we could define such a function to assign a lower cost to a bitwise shift than to integer division. This difference in cost gives rise to an extractor that selects expression ❺, (a ≫ 1) + 2, from the e-graph.

III. Overview

LIAR uses equality saturation to find idioms in functional array programs. Figure 2 shows how LIAR augments the basic equality saturation workflow with a target-independent minimalist array IR and its associated language semantics rules. The IR is the common representation at every step of the system, from input expression ❶ through e-graphs ❷ and ❹ to final extracted expression ❺. To allow LIAR to target libraries, it carves out two target-specific components: an extractor and idiom rewrite rules. Those rewrite rules are combined with the language semantics rules in ❸. Together, they allow the equality saturation engine to expose and exploit latent idioms, as illustrated by the appearance of library function calls in extracted expression ❺.

A. Minimalist Array IR and Rewrite Rules

IR design is an important consideration within equality saturation because the choice of IR determines the size of e-graphs and the number of rewrite rules. Those rules in turn affect the system's feasibility, maintainability, and efficiency.

Recent work [9] has applied equality saturation to a Lift-like [19] functional programming language rooted in the map-reduce paradigm of array processing. This paradigm offers expressiveness to programmers and abundant parallelizability to compilers. Despite these attractive features, a marriage of map-reduce and equality saturation faces two main hurdles: the encoding of λ-calculus and the creation of a comprehensive set of rewrite rules for map-reduce–style operators. The aforementioned recent work addressed the first hurdle by relying on De Bruijn indices instead of named parameters.

The second challenge was addressed by capturing operator identities using a dazzling 156 rewrite rules, requiring over 1000 lines of Scala code![1] Such a large number of rules is undesirable as it required an enormous effort from the system programmer, increases chances of bugs, and makes it hard to understand whether the rule set is complete. That large number

[1] See https://github.com/rise-lang/shine/blob/sges/src/main/scala/rise/eqsat/rules.scala

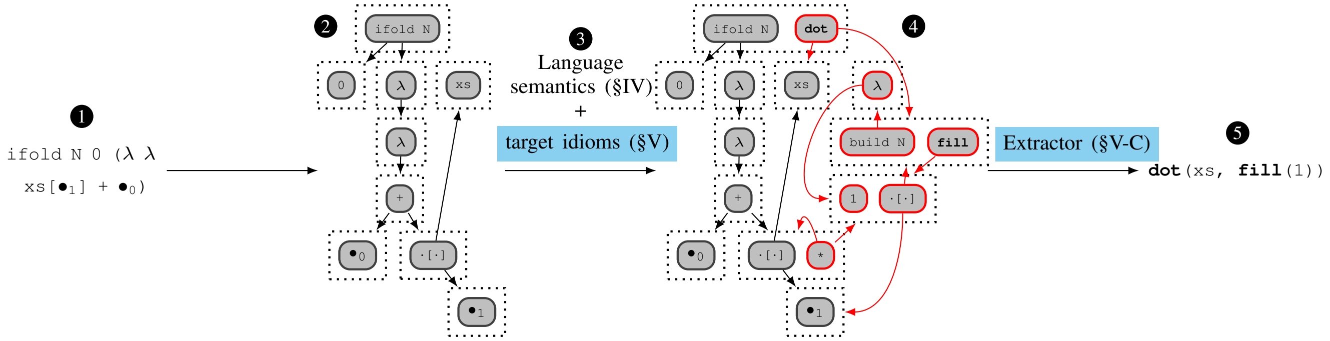


Fig. 2. Overview of LIAR, the proposed technique. Vector sum expression ❶ is converted to e-graph ❷. Equality saturation applies a set of rules ❸ to e-graph ❷. These rules consist of target-independent language semantics and target-specific idioms. Rule application yields updated e-graph ❹. From e-graph ❹, a target-specific extractor chooses expression ❺. The only target-specific components are the target idioms and extractor, both of which are highlighted in blue.

of rules is a direct consequence of the high number of operators and all their interactions that need to be considered.

This work retains the innovation of De Bruijn indices while replacing the map-reduce paradigm's operators — map, reduce, concat, zip, . . . — with just three fundamental operators: build, ifold and array indexing. This combination of three operators was originally introduced by work on Destination-Passing Style (DPS) [18] and is sufficiently powerful to model the map-reduce paradigm's plethora of primitives [10]. As we will see, by paring the number of array processing primitives down to three operators, the number of rewrite rules that represents a robust subset of the IR's semantics drops down to just eight!

B. Target-Specific Rules and Extractor

LIAR supplements the IR and its core rewrite rules with two target-specific components. The first is a set of rewrite rules that recognize library idioms. The second component is an extractor that selects expressions from e-graphs.

A more in-depth discussion of these components is provided in section V. That section describes an implementation of the components for two different libraries: BLAS and PyTorch. Since the target-specific components rely on the target-independent IR, we first describe the minimalist IR design and its rewrite rules in the next section.

IV. Minimalist Array IR and Rewrite Rules

This section describes LIAR's minimalist functional array IR by first introducing the IR's grammar and operators. The section then proceeds by deriving rewrite rules from language semantics and concludes with some examples.

A. Syntax and Primitives

Figure 3 shows the grammar for the proposed minimalist IR. The IR consists of four classes of language primitives: λ-calculus with De Bruijn indices, array operations, tuple operations, and named function calls.

1) λ-Calculus with De Bruijn Indices: De Bruijn indices remove the need for named parameters by identifying a lambda's parameter by the number of lambda definitions between the parameter use and the lambda defining the parameter [2]. For instance, let $\bullet_i$ denote a De Bruijn index. λ $\bullet_0$ is equivalent to λx. x and λ λ $\bullet_1$ means λx. λy. x.

e ::= λ e	*lambda abstraction*
| e e	*lambda application*
| $\bullet_i$	*parameter use*
| build N f	*array construction*
| e[e]	*array indexing*
| ifold N e e	*iteration with accumulator*
| tuple e e	*tuple creation*
| fst e | snd e	*tuple unpacking*
| **f**($\overline{e}$)	*named function application*

Fig. 3. The grammar describing the minimalist IR. $\overline{e}$ indicates zero or more instances of e. N is a compile-time integer constant and **f** is an anonymous function. The set of available named functions depends on the problem domain. For example, **gemm** is a named function when targeting BLAS but not when targeting PyTorch.

A consequence of this standardized naming scheme is that semantically equivalent lambdas become syntactically identical. This syntactic equivalence is, as discussed in related work [9], beneficial in the context of equality saturation because identical expressions correspond to the same e-node in an e-graph. In short, De Bruijn indices keep e-graphs small.

2) Array operations: The IR supplements λ-calculus with three array operations: build, array indexing, and ifold.

The build operator takes an array length N and a lambda f. For each index i ∈ $\{0, 1, \ldots, N-1\}$, build computes f i and packages the resulting values in an array:

$$\texttt{build N f} = \begin{array}{|c|c|c|c|}\hline \texttt{f 0} & \texttt{f 1} & \ldots & \texttt{f (N - 1)} \\ \hline \end{array}$$

Array indexing is conventional. Given an array a and an index i, a[i] produces the ith element of a.

$$\left(\begin{array}{|c|c|c|c|}\hline a_0 & a_1 & \ldots & a_{N-1} \\ \hline \end{array}\right)[i] = a_i$$

The ifold operator's main use lies in array aggregation. It takes three arguments: a compile-time length N, an initial accumulator value init, and a folding function f that takes both an index and an accumulator value. This function

```
(λ e) y = subst(e,y)                              (E-BETAREDUCE)
(build f N)[i] = f i                              (E-INDEXBUILD)
fst (tuple a b) = a                                 (E-FSTTUPLE)
snd (tuple a b) = b                                 (E-SNDTUPLE)
ifold 0 init f = init                                (E-FOLDINIT)
ifold (N + 1) init f = f N (ifold N init f)
                                                     (E-FOLDSTEP)
```

Listing 1. Reduction semantics for the minimalist IR.

is applied iteratively according to the following recursive definition:

```
ifold 0 init f = init
ifold (N + 1) init f = f N (ifold N init f)
```

3) Tuple Operations: Arrays capture sequences of homogeneous data structures whereas tuples in the IR encode sequences of heterogeneous data. The IR defines only binary tuples, since *n*-ary tuples can be built by nesting binary tuples. The two main operations related to tuples are tuple construction with `tuple`, and tuple extraction with `fst` and `snd`. The semantics of these operations are:

```
tuple a b = (a, b)
fst (a, b) = a
snd (a, b) = b
```

4) Named Function Calls: The IR operators discussed so far support fundamental data types like functions, arrays, and tuples. To support operations not covered by the aforementioned operators, the IR uses named function calls.

Nullary named functions such as **0**`()`, **1**`()`, **2**`()`, ..., can be used to model integer and floating-point constants in the IR. For the sake of simplicity, the function call parentheses are omitted when using constants: *e.g.,* `0`, `1`, `2`.

A set of binary functions implement standard scalar arithmetic functions such as **+**`(a, b)`. To make the notation more natural and readable, infix notation is used for these operators, *e.g.,* `a + b`. The IR also supports other scalar functions such as comparison, *e.g.,* `a > b`.

Named functions can also be used to capture external library calls in the IR. For instance, the PyTorch `torch.sum(xs)` function represents the sum of a vector's elements and is equivalent to `ifold n 0` $(\lambda\ \lambda\ \texttt{xs}[\bullet_1]\ +\ \bullet_0)$, where `n` is `xs`'s length. Section V will dive deeper into the process of finding such equivalences in programs and how that process is used to target various libraries.

B. Language Semantics as Rewrite Rules

We now derive the eight rewrite rules that capture the relationships between the core IR primitives. We then discuss how these rules are applied. We also provide examples to illustrate how the rules can simplify expressions and derive new ones.

```
(λ e) y → subst(e,y)                          (R-BETAREDUCE)
e → (λ e↑) y                                  (R-INTROLAMBDA)
(build f N)[i] → f i                      (R-ELIMINDEXBUILD)
f i → (build f N)[i]                     (R-INTROINDEXBUILD)
fst (tuple a b) → a                         (R-ELIMFSTTUPLE)
a → fst (tuple a b)                        (R-INTROFSTTUPLE)
snd (tuple a b) → b                         (R-ELIMSNDTUPLE)
b → snd (tuple a b)                        (R-INTROSNDTUPLE)
```

Listing 2. Eight rewrite rules that capture the relationships between `build`, array access, tuple construction, and tuple deconstruction, λ-abstraction and β-reduction.

1) Language Semantics: The rewrite rules are obtained from the IR's reduction semantics. Those semantics are themselves obtained primarily by observing that the IR semantics equations from the previous subsection can be conveniently substituted into each other. Augmenting that substitution with an identity for β-reduction results in the set of identities in listing 1.

The first identity corresponds to β-reduction where $\mathbf{subst}\,(\texttt{e}, \texttt{y})$ represents the substitution operator for De Bruijn indices. This operator transforms an expression by replacing all references to free variable $\bullet_0$ in `e` with `y` and by then lowering the indices of all other free variables in the resulting expression [2]. For instance, $\mathbf{subst}\,(\bullet_0, \texttt{y}) = \texttt{y}$ and $\mathbf{subst}\,(\bullet_1, \texttt{y}) = \bullet_0$.

2) Rewrite Rules: The identities shown above translate readily to the set of rewrite rules in listing 2. Except for `ifold`, there is one pair of rewrite for each identity. Although the semantics of `ifold` could also be expressed as rewrites, the evaluation section will show that for the examples considered, it is not necessary to rewrite any `ifold`.

All rewrites should be self-explanatory, with the exception of the second one. The ↑ operator is called the *shift operator.* This operator increments the indices of all free variables in `e` to make room for the additional parameter introduced by the lambda. For instance, if $\texttt{e} = \bullet_0$ then $(\lambda\ \texttt{e}\uparrow)\ \texttt{y} = (\lambda\ \bullet_1)\ \texttt{y}$.

That close relationship with the language semantics guarantees that the rewrite rules capture a robust subset of the IR's semantics. This subset will prove of interest in section V and the experiments in section VI will further show that these eight core rules allow an equality saturation engine to effectively reason about and restructure array programs to expose latent idiom occurrences. These occurrences will then be rewritten as calls to highly optimized library functions.

3) Substitution and Shift Operators: An interesting property of the rewrite rules is that R-BETAREDUCE and R-INTROLAMBDA include operators that are part of the rule application process itself: the substitution operator **subst** and the shift operator ↑. These operators manipulate expressions rather than values, making them challenging to express in an equality saturation setting.

This challenge stems from the fact that pattern matching on e-graphs maps the unbound expressions from the rules' left-hand sides to e-classes. Each such e-class captures a potentially unbounded set of expressions whereas substitution and shifting are defined for single expressions.

The literature covers two approaches to address this mismatch. The first approach lifts the substitution and shift operators into the e-graph and manipulates them with special rewrite rules, but these rules necessitate additional saturation steps and generates wasteful intermediate nodes [24]. The second approach applies the operators to individual expressions extracted from each e-class, which requires only one saturation step and generates no wasteful intermediate nodes [9]. This work makes use of the second technique.

4) Free Variables in Patterns: Another challenge arises from rules that inflate expressions. Consider R-IntroFstTuple:

$$\texttt{a} \rightarrow \texttt{fst (tuple a b)}.$$

Standard rule application dictates that whenever the rule matches an e-class `a`, an expression `fst (tuple a b)` is constructed, added to the e-graph, and unified with `a`. Most of these steps are unproblematic, but constructing expression `fst (tuple a b)` is non-obvious because `b` is an unbound variable. Intuitively, this means the rule holds for all `b`.

This work implements that intuition by searching the e-graph for all e-classes that could serve as `b`. In this case, that is every e-class in the graph. Expression `fst (tuple a b)` is then constructed for each suitable `b`, added to the e-graph, and unified with `a`. This approach is applied for every rule where an unbound variable appear on the right-hand side: R-IntroFstTuple, R-IntroSndTuple, R-IntroIndexBuild and R-IntroLambda.

C. Examples

To illustrate how the minimalist IR and its core rewrite rules work in practice, we consider two examples: map fusion and constant array construction.

1) Map Fusion: A standard identity under the map-reduce paradigm is that a pair of map calls can be fused:

$$\texttt{map f (map g xs) = map (g o f) xs.}$$

This assertion is an axiom if `map` is an irreducible operator. However, if `map` is expressed in terms of `build`, map fusion and fission follows readily from E-IndexBuild and E-BetaReduce as stated in listing 1:

$$\begin{aligned}&\texttt{build n } (\lambda\ \texttt{f (build n } (\lambda\ \texttt{g xs}[\bullet_0]))[\bullet_0])\\ &= \texttt{build n } (\lambda\ \texttt{f } ((\lambda\ \texttt{g xs}[\bullet_0])\ \bullet_0))\\ &= \texttt{build n } (\lambda\ \texttt{f (g xs}[\bullet_0])).\end{aligned}$$

Left to right, the identity also follows from rewrite rules R-ElimIndexBuild and R-BetaReduce, meaning an equality saturation engine equipped with those rules will find that maps can be fused. Map fission, the right to left reading of the identity, would require an additional rule due to the specific λ-abstraction rule in use. We choose not to include such a rule because it will not be of interest to the evaluation in section VI.

2) Constant Array Construction: Suppose that we would like to optimize the following expression, which adds 42 to each element of the array `xs`:

$$\texttt{build n } (\lambda\ \texttt{xs}[\bullet_0]\ \texttt{+ 42}).$$

Further assume that we have a library available with very fast implementations of functions **addvec** and **constvec**. **addvec** computes the elementwise addition of two vectors and **constvec** creates a vector of constants. We can express the relationship between these functions and their corresponding idioms using rewrite rules in the LIAR IR:

$$\begin{aligned}&\texttt{build n } (\lambda\ \texttt{a}[\bullet_0]\ \texttt{+ b}[\bullet_0])) \rightarrow \textbf{addvec(a, b)},\\ &\texttt{build n } (\lambda\ \texttt{c}) \rightarrow \textbf{constvec(c)}.\end{aligned}$$

Unfortunately, neither of these patterns appear in the expression to optimize. However, the rewrite rules seen earlier allow for a scalar to be transformed to an indexed array of such scalars:

$$\texttt{0} = (\lambda\ \texttt{0})\ \texttt{i} = \texttt{build n } (\lambda\ \texttt{0})[\texttt{i}].$$

This identity allows us to infer that

$$\begin{aligned}&\texttt{build n } (\lambda\ \texttt{xs}[\bullet_0]\ \texttt{+ 42})\\ &= \texttt{build n } (\lambda\ \texttt{xs}[\bullet_0]\ \texttt{+ (build n } (\lambda\ \texttt{42}))[\bullet_0]))\\ &= \textbf{addvec}\texttt{(xs, build n } (\lambda\ \texttt{42}))\\ &= \textbf{addvec}\texttt{(xs, }\textbf{constvec}\texttt{(42))}.\end{aligned}$$

To summarize, one can creatively rewrite programs to use library functions by relying on a combination of core rewrite rules and library-specific idiom rewrite rules. This rewriting process can be fully automated by leaving rule application to an equality saturation engine, which will explore potential rule applications and arrive at the same conclusion as a human expert who carefully uses the same rules.

Notably, the conclusion to which both humans and equality saturation arrive is in this case that there are *latent* idioms in the original expression — idioms that are not directly observable. Once these idioms have been identified, the original application can be readily accelerated using high-performance library calls.

V. Use Cases: BLAS and PyTorch

The constant array construction example from the previous section illustrates that LIAR's IR is suitable for robust pattern matching and rewriting, even when input programs do not exactly match library idioms. This section generalizes the approach from that example and implements it more rigorously for two libraries: BLAS and PyTorch. Those implementations follow the blueprint from fig. 2 in section III; hence, they consist of two components: target idioms and an extractor.

Since BLAS and PyTorch both operate on floating-point numbers, this section separates the idiom rules into shared scalar arithmetic rules and library-specific idiom rules. The section then presents the extractor and its cost model, which also facilitates sharing between the BLAS and PyTorch targets.

$\texttt{x + 0} = \texttt{x}$ (E-AddZero)

$\texttt{1 * x} = \texttt{x}$ (E-MulOneL)

$\texttt{x * 1} = \texttt{x}$ (E-MulOneR)

$\texttt{x * y} = \texttt{y * x}$ (E-CommuteMul)

Listing 3. Scalar arithmetic identities. Each identity corresponds to two rewrite rules: a left-to-right rule and a right-to-left rule. x and y are numbers.

A. Scalar Arithmetic Rules

Listing 3 provides a small list of scalar rewrite rules. When combined with the core rules, the scalar rules allow an equality saturation engine to reason about tensors. That reasoning is designed to enable idiom detection even if the idioms seem hidden at first.

For example, consider the latent dot product in vector sum

$$\texttt{ifold n 0 } (\lambda\ \lambda\ \texttt{xs}[\bullet_1] + \bullet_0).$$

We can expose that dot product idiom by first applying E-MulOneR to $\texttt{xs}[\bullet_1]$, yielding $\texttt{xs}[\bullet_1] * \texttt{1}$. We then use R-IntroLambda and R-IntroIndexBuild on that constant 1 to obtain a final expression of

$$\texttt{ifold n 0 } (\lambda\ \lambda\ \texttt{xs}[\bullet_1] * (\texttt{build n } (\lambda\ \texttt{1}))[\bullet_1] + \bullet_0)$$

which is equivalent to a dot product with a vector of ones: **dot**(xs, build n (λ 1)).

B. Idiom Rules

The dot product idiom recognition example illustrates how the core and scalar rewrite rules work together to expose latent idioms. Once such idioms have been exposed, they still need to be recognized as such. To that end, we now introduce idiom-recognizing rewrite rules for BLAS and PyTorch.

1) BLAS: Listing 4 contains equivalences that define idioms corresponding to five BLAS functions. These functions are:

1) **dot**: computes the dot product of two vectors;
2) **axpy**: computes $\alpha A + B$, where α is a scalar and A, B are vectors;
3) $\textbf{gemv}^{\text{X}}$: computes $\alpha AB + \beta C$, where α, β are scalars, B, C are vectors, and A is a matrix;
4) $\textbf{gemm}^{\text{X,Y}}$: computes $\alpha AB + \beta C$, where α, β are scalars and A, B, C are matrices; and
5) **transpose**: transposes a matrix.

The idioms in listing 4 encompass both the transposed and non-transposed variants of these BLAS functions, resulting in a larger number of equivalences. For instance, in the case of $\textbf{gemv}^{\text{X}}$, I-Gemv defines $\textbf{gemv}^{\text{F}}$, and I-TransposeInGemv connects $\textbf{gemv}^{\text{F}}$ to its transposed counterpart, denoted as $\textbf{gemv}^{\text{T}}$. The inclusion of these transposed variations allows for a more comprehensive definition of BLAS functions. Additionally, listing 4 also presents an idiom for the C standard library **memset** function, which can be used to quickly create an all-zeros vector. The shift operator ($\uparrow$) applications in listing 4 increment De Bruijn indices to avoid overlap with new parameters introduced by λ-abstraction.

$\textbf{axpy}(\alpha, \texttt{A}, \texttt{B})$

$= \texttt{build N } (\lambda\ \alpha\uparrow * \texttt{A}\uparrow[\bullet_0] + \texttt{B}\uparrow[\bullet_0])$ (I-Axpy)

$\textbf{dot}(\texttt{A}, \texttt{B})$

$= \texttt{ifold N 0 } (\lambda\ \lambda\ \texttt{A}\uparrow\uparrow[\bullet_1] * \texttt{B}\uparrow\uparrow[\bullet_1] + \bullet_0)$ (I-Dot)

$\textbf{gemv}^{\text{F}}(\alpha, \texttt{A}, \texttt{B}, \beta, \texttt{C})$

$= \texttt{build N } (\lambda\ \alpha\uparrow * \textbf{dot}(\texttt{A}\uparrow[\bullet_0], \texttt{B}\uparrow) + \beta\uparrow * \texttt{C}\uparrow[\bullet_0])$ (I-Gemv)

$\textbf{gemm}^{\text{F,T}}(\alpha, \texttt{A}, \texttt{B}, \beta, \texttt{C})$

$= \texttt{build N } (\lambda\ \textbf{gemv}^{\text{N}}(\alpha\uparrow, \texttt{B}\uparrow, \texttt{A}\uparrow[\bullet_0], \beta\uparrow, \texttt{C}\uparrow[\bullet_0]))$ (I-Gemm)

$\textbf{transpose}(\texttt{A})$

$= \texttt{build N } (\lambda\ \texttt{build M } (\lambda\ \texttt{A}\uparrow\uparrow[\bullet_0][\bullet_1]))$ (I-Transpose)

$\textbf{gemv}^{\text{X}}(\alpha, \textbf{transpose}(\texttt{A}), \texttt{B}, \beta, \texttt{c})$

$= \textbf{gemv}^{\neg\text{X}}(\alpha, \texttt{A}, \texttt{B}, \beta, \texttt{c})$ (I-TransposeInGemv)

$\textbf{gemm}^{\text{X,Y}}(\alpha, \textbf{transpose}(\texttt{A}), \texttt{B}, \beta, \texttt{C})$

$= \textbf{gemm}^{\neg\text{X,Y}}(\alpha, \texttt{A}, \texttt{B}, \beta, \texttt{C})$ (I-TransposeAInGemm)

$\textbf{gemm}^{\text{X,Y}}(\alpha, \texttt{A}, \textbf{transpose}(\texttt{B}), \beta, \texttt{c})$

$= \textbf{gemm}^{\text{X,}\neg\text{Y}}(\alpha, \texttt{A}, \texttt{B}, \beta, \texttt{c})$ (I-TransposeBInGemm)

$\textbf{dot}(\texttt{build N } (\lambda\ \alpha * \texttt{A}[\bullet_0]), \texttt{B})$

$= \texttt{alpha * } \textbf{dot}(\texttt{A}, \texttt{B})$ (I-HoistMulFromDot)

$\textbf{memset}(\texttt{0})$

$= \texttt{build N } (\lambda\ \texttt{0})$ (I-MemsetZero)

Listing 4. BLAS idioms considered in this work. F in $\textbf{gemv}^{\text{F}}$ is short for *false* and indicates that matrix A is not transposed. F,T in $\textbf{gemm}^{\text{F,T}}$ indicate that A is not transposed and B is transposed.

$\textbf{dot}(\texttt{A}, \texttt{B})$

$= \texttt{ifold N 0 } (\lambda\ \lambda\ \texttt{A}\uparrow\uparrow[\bullet_1] * \texttt{B}\uparrow\uparrow[\bullet_1] + \bullet_0)$ (I-Dot)

$\textbf{sum}(\texttt{A})$

$= \texttt{ifold N 0 } (\lambda\ \lambda\ \texttt{A}\uparrow\uparrow[\bullet_1] + \bullet_0)$ (I-VecSum)

$\textbf{mv}(\texttt{A}, \texttt{B})$

$= \texttt{build N } (\lambda\ \textbf{dot}(\texttt{A}\uparrow[\bullet_1], \texttt{B}\uparrow))$ (I-MatVec)

$\textbf{mm}(\texttt{A}, \texttt{B})$

$= \texttt{build N } (\lambda\ \textbf{mv}(\texttt{B}\uparrow, \texttt{A}\uparrow[\bullet_1]))$ (I-MatMat)

$\textbf{transpose}(\texttt{A})$

$= \texttt{build N } (\lambda\ \texttt{build M } (\lambda\ \texttt{A}\uparrow\uparrow[\bullet_0][\bullet_1]))$ (I-Transpose)

$\textbf{transpose}(\textbf{transpose}(\texttt{A}))$

$= \texttt{A}$ (I-TransposeTwice)

$\textbf{add}(\texttt{A}, \texttt{B})$

$= \texttt{build N } (\lambda\ \texttt{A}\uparrow[\bullet_0] + \texttt{B}\uparrow[\bullet_0])$ (I-AddVec)

$\textbf{add}(\texttt{A}, \texttt{B})$

$= \texttt{build N } (\lambda\ \textbf{add}(\texttt{A}\uparrow[\bullet_0], \texttt{B}\uparrow[\bullet_0]))$ (I-LiftAdd)

$\textbf{mul}(\alpha, \texttt{A})$

$= \texttt{build N } (\lambda\ \alpha * \texttt{A}\uparrow[\bullet_0])$ (I-MulScalarAndVec)

$\textbf{mul}(\alpha, \texttt{A})$

$= \texttt{build N } (\lambda\ \textbf{mul}(\alpha, \texttt{A}\uparrow[\bullet_0]))$ (I-LiftMul)

$\textbf{full}(\texttt{c})$

$= \texttt{build N } (\lambda\ \texttt{c}\uparrow)$ (I-FullVec)

Listing 5. PyTorch idioms considered in this work. I-TransposeTwice captures a property of the **transpose** function; all other rules recognize idioms.

$$
\begin{aligned}
&\mathit{cost}\,(\texttt{build N f}) &&= \texttt{N}\cdot(\mathit{cost}\,(\texttt{f})+1)+1\\
&\mathit{cost}\,(\texttt{A[i]}) &&= \mathit{cost}\,(\texttt{A})+\mathit{cost}\,(\texttt{i})+1\\
&\mathit{cost}\,(\texttt{ifold N init f}) &&= \mathit{cost}\,(\texttt{init})+\texttt{N}\cdot\mathit{cost}\,(\texttt{f})+1\\
&\mathit{cost}\,(\texttt{tuple a b}) &&= \mathit{cost}\,(\texttt{a})+\mathit{cost}\,(\texttt{b})+1\\
&\mathit{cost}\,(\texttt{fst t}) &&= \mathit{cost}\,(\texttt{t})+1\\
&\mathit{cost}\,(\texttt{snd t}) &&= \mathit{cost}\,(\texttt{t})+1\\
&\mathit{cost}\,(\lambda\ \texttt{e}) &&= \mathit{cost}\,(\texttt{e})+1\\
&\mathit{cost}\,(\texttt{f e}) &&= \mathit{cost}\,(\texttt{f})+\mathit{cost}\,(\texttt{e})+1\\
&\mathit{cost}\,(\bullet_{\texttt{k}}) &&= 1\ (\forall \texttt{k}\in\mathbb{N})\\
&\mathit{cost}\,(\texttt{a + b}) &&= \mathit{cost}\,(\texttt{a})+\mathit{cost}\,(\texttt{b})+1\\
&\mathit{cost}\,(\texttt{a * b}) &&= \mathit{cost}\,(\texttt{a})+\mathit{cost}\,(\texttt{b})+1\\
&\mathit{cost}\,(c) &&= 1\ (\forall \texttt{c}\in\mathbb{R})
\end{aligned}
$$

Listing 6. Definition of the base cost function.

2) PyTorch: We implement a similar set of idioms for PyTorch, described in listing 5. The PyTorch functions corresponding to those idioms take fewer parameters than their BLAS counterparts, but are often polymorphic. For instance, `A` in **`mul`**`(alpha, A)` could be a scalar, vector, matrix or higher-order tensor. An array of multiplications `[mul(a,` $\texttt{A}_1$`), mul(a,` $\texttt{A}_2$`), ...]` can hence be rewritten as a single call `mul(a, [`$\texttt{A}_1$`,` $\texttt{A}_2$`, ...])`. I-LiftMul captures this polymorphic property by defining a higher-dimensional **`mul`** as a vector of lower-dimensional **`mul`** calls. I-LiftAdd accomplishes the same for the **`add`** function.

C. Cost Model

The extraction step of equality saturation, initially designed for a pseudo-Boolean solver, can be implemented in various ways [20]. The most popular implementation employs a local cost model, which quickly and simply calculates the cost of each e-node within an e-class based on its arguments' cost [24]. An e-class's cost is determined by its cheapest e-node, and the extraction process involves selecting the cheapest e-node recursively from an e-class.

This paper opts for the cost model-based approach for simplicity's sake — the extractor for the BLAS and PyTorch use cases is not the focal point of this work. The cost models for the two use cases consist of a common base and a library function cost model. The common base cost in listing 6 describes the cost of the core IR operators and library-independent named functions. Listing 7 and listing 8 capture the BLAS and PyTorch library function cost model respectively.

VI. Evaluation

The previous section has examined qualitatively how LIAR can target different libraries such as BLAS and PyTorch. This section uses quantitative experiments to measure how well LIAR identifies idioms and speeds up programs.

To perform these experiments, we implement the IR as a Scala Domain-Specific Language (DSL). We encode in that DSL a subset of the PolyBench/C 4.2.1-beta benchmark suite [15] and add custom kernels to evaluate specific tasks. The custom and PolyBench kernels are described in table I.

$$
\begin{aligned}
&\mathit{cost}\,(\textbf{memset}\texttt{(c)}) &&= \mathit{cost}\,(\texttt{c})+.8N+1\\
&\mathit{cost}\,(\textbf{dot}\texttt{(A,B)}) &&= \mathit{cost}\,(\texttt{A})+\mathit{cost}\,(\texttt{B})+.8N\\
&\mathit{cost}\,(\textbf{axpy}\texttt{(a,A,B)}) &&= \mathit{cost}\,(\texttt{a})+\cdots+\mathit{cost}\,(\texttt{B})+.8N\\
&\mathit{cost}\,(\textbf{gemv}\texttt{(a,A,B,b,C)}) &&= \mathit{cost}\,(\texttt{a})+\cdots+\mathit{cost}\,(\texttt{C})+.7NM\\
&\mathit{cost}\,(\textbf{gemm}\texttt{(a,A,B,b,C)}) &&= \mathit{cost}\,(\texttt{a})+\cdots+\mathit{cost}\,(\texttt{C})+.6NMK\\
&\mathit{cost}\,(\textbf{transpose}\texttt{(A)}) &&= \mathit{cost}\,(\texttt{A})+.9NM
\end{aligned}
$$

Listing 7. BLAS-specific additions to *cost*. Calls to external functions are discounted to make them more attractive. Discounting factors are chosen semi-arbitrarily. N, M, and K are array dimensions.

$$
\begin{aligned}
&\mathit{cost}\,(\textbf{full}\texttt{(c)}) &&= \mathit{cost}\,(\texttt{c})+.8N+1\\
&\mathit{cost}\,(\textbf{add}\texttt{(A, B)}) &&= \mathit{cost}\,(\texttt{A})+\mathit{cost}\,(\texttt{B})+.4N+.4M\\
&\mathit{cost}\,(\textbf{mul}\texttt{(A, B)}) &&= \mathit{cost}\,(\texttt{A})+\mathit{cost}\,(\texttt{B})+.4N+.4M\\
&\mathit{cost}\,(\textbf{sum}\texttt{(A, B)}) &&= \mathit{cost}\,(\texttt{A})+\mathit{cost}\,(\texttt{B})+.8N\\
&\mathit{cost}\,(\textbf{dot}\texttt{(A, B)}) &&= \mathit{cost}\,(\texttt{A})+\mathit{cost}\,(\texttt{B})+.8N\\
&\mathit{cost}\,(\textbf{mv}\texttt{(A, B)}) &&= \mathit{cost}\,(\texttt{A})+\mathit{cost}\,(\texttt{B})+.7NM\\
&\mathit{cost}\,(\textbf{mm}\texttt{(A, B)}) &&= \mathit{cost}\,(\texttt{A})+\mathit{cost}\,(\texttt{B})+.6NMK\\
&\mathit{cost}\,(\textbf{transpose}\texttt{(A)}) &&= \mathit{cost}\,(\texttt{A})+.9NM
\end{aligned}
$$

Listing 8. PyTorch-specific additions to *cost*. N and M are array dimensions. For polymorphic arrays, N and M represent the product of the arrays' dimensions.

Kernels are expressed by composing `build`-`ifold` implementations of the respective mathematical operators as in prior work [18]. For instance, *gemv* becomes:

$$
\begin{aligned}
&\texttt{gemv(}\alpha\texttt{, A, B, }\beta\texttt{, C)}\\
&\ = \texttt{vadd(vscale(}\alpha\texttt{, matvec(A, B)), vscale(}\beta\texttt{, C)).}
\end{aligned}
$$

`vadd`, `vscale` and `matvec` expand as below.

$$
\begin{aligned}
&\texttt{vadd(A, B) = build N (}\lambda\ \texttt{A}\!\uparrow\!\texttt{[}\bullet_0\texttt{] + B}\!\uparrow\!\texttt{[}\bullet_0\texttt{])}\\
&\texttt{vscale(}\alpha\texttt{, A) = build N (}\lambda\ \alpha\!\uparrow\ \texttt{* A}\!\uparrow\!\texttt{[}\bullet_0\texttt{])}\\
&\texttt{matvec(A, B) = build N (}\lambda\ \texttt{dot(A}\!\uparrow\!\texttt{[}\bullet_0\texttt{], B}\!\uparrow\!\texttt{))}\\
&\texttt{dot(A, B) = ifold N 0 (}\lambda\ \lambda\ \texttt{A}\!\uparrow\uparrow\!\texttt{[}\bullet_1\texttt{] * B}\!\uparrow\uparrow\!\texttt{[}\bullet_1\texttt{] + }\bullet_0\texttt{)}
\end{aligned}
$$

The exceptions to this scheme are *doitgen* and *gemver*. We translate their C loops directly to `build` and `ifold`.

Equality Saturation engine: All kernels are transformed by a Scala equality saturation engine inspired by the efficient `egg` implementation [24]. The engine performs e-graph construction, saturation and expression extraction. Saturation proceeds based on one of three sets of rewrite rules, which we term *targets:*

1) **Pure C:** Core and scalar rules only;
2) **BLAS idioms:** Core, scalar and BLAS rules;
3) **PyTorch idioms:** Core, scalar and PyTorch rules.

The engine runs equality saturation for five minutes per kernel per benchmark. After each saturation step, the cost model from section V-C selects the optimal expression for that step.

Code Generation: For BLAS and pure C, the selected expressions are compiled to C using an approach similar to prior work [10] on C compilation from a functional IR. There are no run-time Python results presented since the compiler

TABLE I. OVERVIEW OF KERNELS EXAMINED IN THIS WORK. POLYBENCH KERNEL DESCRIPTIONS ADAPTED FROM BENCHMARK SUITE [15].

Kernel	Suite	Description
2mm	PolyBench	Two generalized matrix multiplications
atax	PolyBench	Matrix transpose and vector multiplication
doitgen	PolyBench	Multiresolution analysis kernel (MADNESS)
gemm	PolyBench	Generalized matrix product
gemver	PolyBench	Vector multiplication and matrix addition
gesummv	PolyBench	Scalar, vector and matrix multiplication
jacobi1d	PolyBench	1D Jacobi stencil computation
mvt	PolyBench	Matrix-vector product and transpose
1mm	Custom	One matrix multiplication
axpy	Custom	Vector scaling and addition
blur1d	Custom	1D stencil
gemv	Custom	Generalized matrix-vector product
memset	Custom	Zero vector creation
slim-2mm	Custom	Two matrix multiplications
stencil2d	Custom	2D stencil
vsum	Custom	Vector reduction with sum

used does not currently have a Python back-end. As such, results for PyTorch are purely qualitative.

We assess LIAR through experiments on library calls, their evolution over time, code profiling, and run time comparison.

A. Experimental Setup

We run all experiments on a server with two 18-core Intel Xeon Gold 6254 CPUs and 1 TiB of RAM. The server uses the following software: CentOS Linux 7, Scala 2.12.7, GCC 11.2.1, and OpenBLAS 0.3.3. Through OpenBLAS, compiled kernels take advantage of the server's many cores. Kernels not compiled for OpenBLAS are single-threaded. The LIAR implementation that generates kernels is also single-threaded.

B. Idioms Recognized

This section qualitatively assesses how well LIAR finds BLAS and PyTorch idioms. To perform this assessment, we report the library calls found in extracted expressions. These data are reported for all kernels by table II and table III.

The tables show that LIAR finds idioms in each kernel. The idioms found depend on the kernel and target. For instance, LIAR finds that the *gemv* kernel is best described as a **gemv** call when targeting BLAS. Indeed, the solution found is simply

$$\textbf{gemv}^{\mathrm{F}}(\alpha,\ \texttt{A},\ \texttt{B},\ \beta,\ \texttt{C}).$$

When targeting PyTorch, LIAR implements the same kernel as a combination of more granular `add`, `mul` and `mv` calls.

$$\textbf{add}(\textbf{mv}(\textbf{mul}(\alpha,\ \texttt{A}),\ \texttt{B}),\ \textbf{mul}(\beta,\ \texttt{C})).$$

The idioms in the *gemv* kernel are readily apparent, but this is not the case for all benchmarks. Consider *doitgen*:

$$\begin{array}{l}\texttt{build N }(\lambda\texttt{ build N }(\lambda\texttt{ build N }\lambda\texttt{ (}\\ \quad\texttt{ifold N 0 }(\lambda\ \lambda\texttt{ A[}\bullet_4\texttt{][}\bullet_3\texttt{][}\bullet_1\texttt{] * B[}\bullet_2\texttt{][}\bullet_1\texttt{] + }\bullet_0\texttt{)))).}\end{array}$$

LIAR finds a surprisingly insightful PyTorch solution:

$$\texttt{build N }(\lambda\ \textbf{mm}\texttt{(A[}\bullet_0\texttt{], }\textbf{transpose}\texttt{(B))).}$$

TABLE II. SOLUTIONS FOUND FOR KERNELS WHEN TARGETING BLAS. *Steps* DESCRIBES THE NUMBER OF SATURATION STEPS. *Solution* DESCRIBES THE LIBRARY CALLS FOUND AT THE LAST STEP. *e-Nodes* COUNTS THE UNIQUE E-NODES IN THE E-GRAPH, ALSO AT THE LAST STEP.

Kernel	Solution	Steps	e-Nodes
2mm	3 × axpy + 1 × dot + 1 × gemv + 3 × memset + 1 × transpose	6	3.46×10^4
atax	2 × gemv + 2 × memset	7	3.95×10^4
doitgen	1 × gemm + 1 × memset	8	4.70×10^4
gemm	1 × axpy + 1 × gemv + 1 × memset	7	4.95×10^4
gemver	2 × axpy + 2 × dot	5	1.69×10^4
gesummv	2 × gemv + 1 × memset	7	4.27×10^4
jacobi1d	1 × gemv + 1 × memset	5	2.53×10^4
mvt	2 × gemv + 2 × memset	7	2.69×10^4
1mm	1 × gemm + 1 × memset	8	4.47×10^4
axpy	1 × axpy	11	1.36×10^4
blur1d	1 × gemv + 1 × memset	6	5.39×10^4
gemv	1 × gemv	7	3.43×10^4
memset	1 × memset	11	5.31×10^3
slim-2mm	1 × gemm + 1 × gemv + 2 × memset	7	5.18×10^4
stencil2d	1 × gemv + 1 × memset	5	5.88×10^4
vsum	1 × dot	10	1.59×10^4

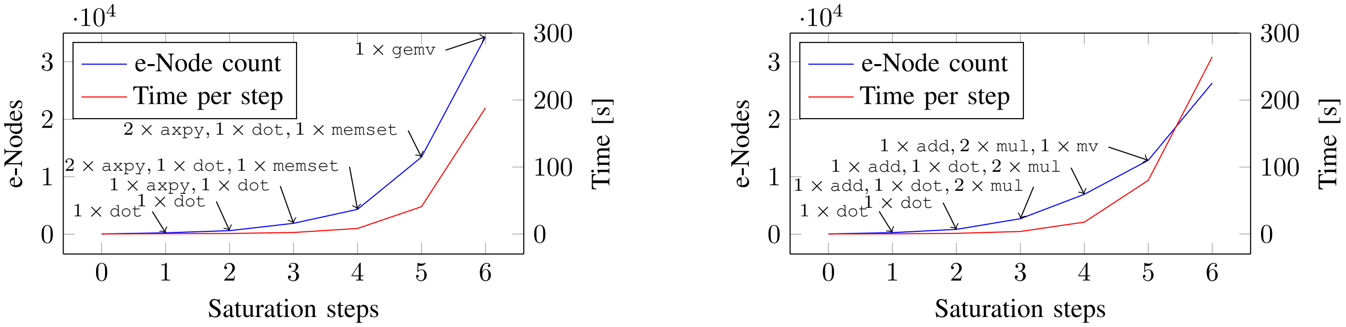
TABLE III. SOLUTIONS FOUND FOR KERNELS WHEN TARGETING PYTORCH. COLUMNS HAVE THE SAME MEANING AS IN TABLE II.

Kernel	Solution	Steps	e-Nodes
2mm	1 × add + 2 × mul + 2 × mv + 2 × transpose	5	2.28×10^4
atax	2 × mv + 1 × transpose	7	1.98×10^4
doitgen	1 × mm + 1 × transpose	7	2.75×10^4
gemm	1 × add + 1 × mm + 2 × mul	6	2.68×10^4
gemver	2 × add + 1 × dot + 1 × mul + 1 × mv	5	2.38×10^4
gesummv	1 × add + 2 × mul + 2 × mv	7	3.16×10^4
jacobi1d	1 × full + 1 × mv	5	3.13×10^4
mvt	2 × mv + 1 × transpose	7	1.69×10^4
1mm	1 × mm	7	1.99×10^4
axpy	1 × add + 1 × mul	10	2.27×10^4
blur1d	1 × full + 1 × mv	5	2.13×10^4
gemv	1 × add + 2 × mul + 1 × mv	7	2.63×10^4
memset	1 × full	11	2.03×10^3
slim-2mm	1 × mm + 1 × mv + 1 × transpose	6	2.03×10^4
stencil2d	1 × full + 1 × mv	5	9.06×10^4
vsum	1 × sum	10	1.79×10^4

The BLAS **gemm** function is even trickier to find in *doitgen*. LIAR nonetheless uncovers the idiom by inserting constants and by building a zero matrix using **memset**.

$$\begin{array}{l}\texttt{build N }(\lambda\ \textbf{gemm}^{\mathrm{F,T}}\texttt{(1, A[}\bullet_0\texttt{], B,}\\ \qquad\qquad\texttt{1, build N }(\lambda\ \textbf{memset}\texttt{(0)))}\end{array}$$

LIAR does not find an optimal result for all kernels. The *2mm* kernel, for example, could be implemented as **gemm** calls. LIAR does not find that solution because it would require more saturation steps than the time budget allows for. Nonetheless, as we will see later, there is still a large speedup obtained by using the detected idioms and the corresponding library calls.



(a) Solutions over time for *gemv*, targeting BLAS.

(b) Solutions over time for *gemv*, targeting PyTorch.

Fig. 4. Solutions over time. The x-axes show equality saturation steps; the y-axes correspond to the number of e-nodes in the e-graph and the amount of time spent performing a saturation step. Labeled arrows indicate a new best solution has been found.

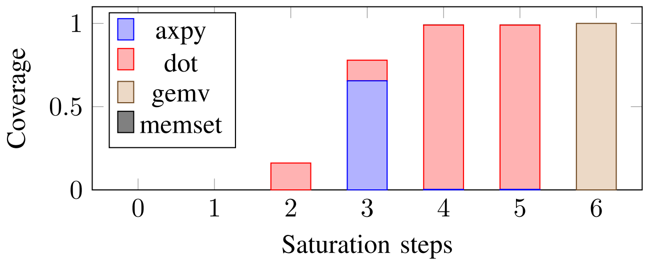


Fig. 5. Coverage over time for the *gemv* kernel, targeting BLAS. The x-axis shows saturation steps; the stacked bars depict the ratio of time spent in library functions. Higher is better.

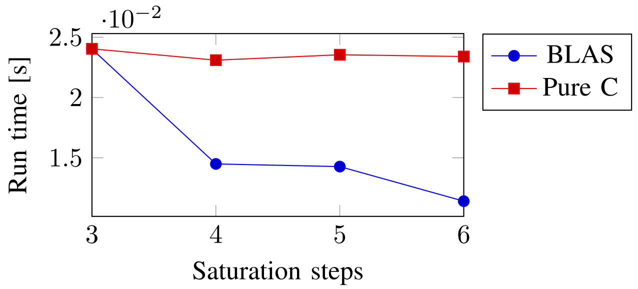


Fig. 6. *gemv* run times. The x-axis shows saturation steps; the y-axis shows the run time of solutions. Lower is better.

C. Idioms over Time

The evaluation continues by investigating how LIAR's solutions evolve. This evolution is made apparent by plotting kernel solutions over time, as fig. 4 does for the *gemv* kernel.

Figure 4a demonstrates that LIAR progressively discovers more suitable solutions with each saturation step. Initially, these solutions consist of dot products. From steps three to five, LIAR incorporates vector addition and scaling using **axpy**. In step six, these function calls converge into a **gemv** call. A similar evolution is observed for PyTorch, as illustrated in Figure 4b.

D. Coverage

We measure the ratio of time kernels spend in the library function to validate LIAR's effective work offloading. Figure 5 presents the coverage for the *gemv* kernel, which targets BLAS. The initial **dot**-based solutions show poor coverage, with negligible coverage at step one and only 16 % at step two. However, step three demonstrates a significant improvement, with **axpy** achieving 66 % coverage and an additional 12 % coverage from **dot**. Steps four and five favor **dot**, reaching 99 % coverage. Finally, at step six, **gemv** achieves complete 100 % coverage. Although **memset** appears in two solutions, its coverage contribution is insignificant.

E. Run Time Performance

We now examine how LIAR's idiom recognition affects run-time performance. For every kernel and every saturation step, we compile LIAR's pure C and BLAS solutions to C code. We run each solution as many times as we can over the course of one minute and calculate the mean run time.

Figure 6 reports run times for the *gemv* kernel when targeting BLAS and pure C. Numbers are reported for steps three through six only because the high-level kernel needs to be optimized for a few steps by equality saturation before it can be executed in reasonable time. Once this is achieved at step three, the pure C and BLAS solutions are equally fast. They diverge as the BLAS solution achieves increasingly high coverage.

Figure 7 shows that recognizing idioms results in a run-time speedup of 2.5 on *gemv*. Speedups vary by kernel. Notable outliers include *1mm*, *vsum*, *blur1d*, and *stencil2d*. The *1mm* kernel benefits from an ideal BLAS solution, netting it a speedup of 19.72. The cost model replaces the *vsum* kernel with a **dot** call, but the associated input array construction outweighs the benefit. Similarly, the cost model chooses to reduce the convolutions in *blur1d* and *stencil2d* to matrix-vector products, performing an im2col transformation. In practice, this transformation is slower than a direct solution. The *gemver* kernel is excluded from the chart as none of its solutions completed within the one-minute time limit. The geometric mean speedup across all kernels except *gemver* is 1.46 with idiom recognition. Pure C incurs a slowdown of 0.26 on average. If we choose the fastest solution for each kernel, LIAR's generated code is 81 % faster than reference.

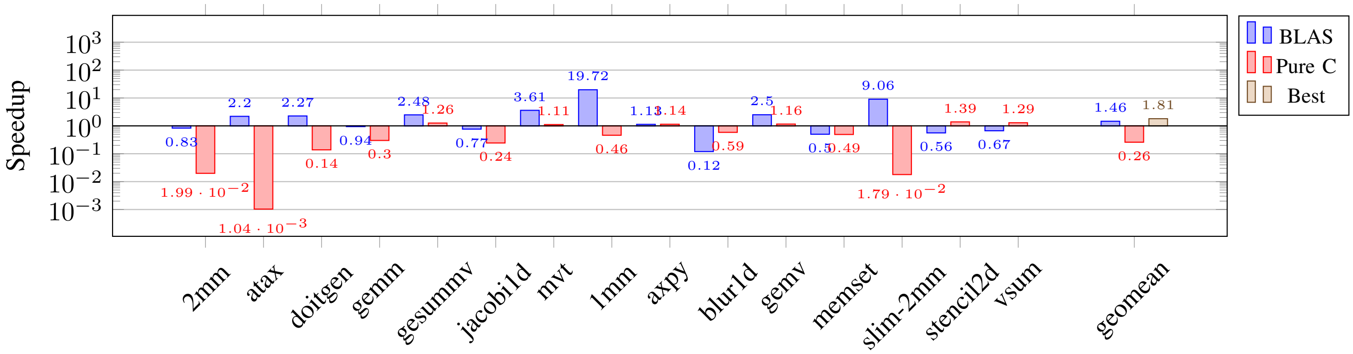


Fig. 7. Run time speedup of LIAR's solutions compared to reference implementations in C. For PolyBench kernels, the reference implementations are the original benchmarks; for custom benchmarks, they are hand-written C programs coded in the style of PolyBench kernels. Each bar represents the quotient of the reference run time and the LIAR solution run time. Higher is better.

VII. Related Work

There exist a number of works that relate to LIAR. LIAR's IR lies at the intersection of work on the `build-ifold` paradigm and advances in equality saturation.

a) Build-Ifold: The `build` and `ifold` operators were originally introduced in work on bringing DPS to functional languages [18]. In that work, `build` and `ifold` were chosen for their similarity to C `for` loops. Despite the simplicity of these two operators, further work showed that they can model map-reduce's plethora of data processing primitives [10]. To the best of our knowledge, LIAR is the first to combine `build-ifold` with equality saturation.

b) Equality Saturation: Another core dependency on which this paper relies is equality saturation. Originally conceived as a means to optimize Java programs [20], a recent boom in equality saturation research has applied the technique to various specialized problems [12, 22, 23, 25]. These problems include the linear algebra domain that forms LIAR's use case in this paper. Specifically, one work focuses on sum-product optimization [23] while another optimizes tensor graphs [25]. LIAR differs from these previous studies since it is not domain-specific. It is a general technique for detecting array processing idioms in a functional IR.

c) λ-Calculus and Equality Saturation: The mentioned use cases avoided name bindings found in λ-calculus. An initial attempt to encode λ-calculus in e-graphs was demonstrated in a broader work on `egg`, an optimized equality saturation implementation [24]. Another study, which did not involve idiom recognition, improved on `egg`'s encoding by using De Bruijn indices [9]. It was also the first to examine equality saturation for functional array programming. LIAR adopts both innovations and drastically reduces the number of rewrite rules compared to that prior work by relying on `build-ifold`, paving the way for a novel approach to idiom recognition.

d) Idiom Recognition: Another body of related work is idiom recognition itself. Idiom rewriting has long served to optimize both imperative [14] and functional languages [7]. State-of-the-art idiom rewriting research relies on flexible patterns that capture idioms and their variations.

One approach to make patterns more flexible is to encode them as graphs and to find candidate matches using topological embedding [8]. Dedicated transformations can then massage some of these candidates to match the original pattern.

Patterns can also be explicitly made flexible. A recent study describes linear algebra patterns in LLVM IR using a language called Idiom Description Language (IDL) [5]. The language relies on composable patterns that abstract away variation points such as multiplications by one. This allows IDL to recognize but not automatically rewrite patterns including BLAS calls. KernelFaRer, another recent work, takes a similar approach [3]. It captures variations using flexible idioms that are expressed as LLVM pattern matching components.

By contrast, Source Matching and Rewriting (SMR) [4] and the MLIR PDL dialect [17] describe patterns as source language snippets, making the patterns easier to express at the cost of making them vulnerable to program variations.

LIAR stands out from previous approaches by recognizing program variations, as IDL and KernelFaRer do, while relying on simple rules in the same language as input programs, in the style of SMR and PDL. LIAR accomplishes this feat through its fusion of a minimalist `build-ifold` IR and equality saturation. The cost of that accomplishment is speed: the approach outlined in this paper is orders of magnitude slower than direct pattern matching, limiting its use for larger kernels. Future work includes addressing that limitation.

VIII. Conclusion

This paper has addressed idiom recognition for functional array programs by applying equality saturation to a minimalist IR with a small set of core rewrite rules. As seen, this minimalist approach is robust and is able to find idioms that are not explicitly present in the original input program.

Using the BLAS and PyTorch libraries as libraries, this paper has shown how the idioms founds in these libraries can be expressed easily using the minimalist IR. The evaluation also demonstrated the robustness of this approach and that it is possible to find idioms corresponding to these libraries on a set of computational kernels, leading to improved performance.

Data-Availability Statement

This paper's artifact is available on Zenodo [21].

Acknowledgements

We thank Christof Schlaak for implementing the core SHIR project that we build on. We also thank Zhitao Lin for the SHIR C code back-end that we use to generate C code from high-level kernels. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants Program [grant RGPIN-2020-05889], and the Canada CIFAR AI Chairs Program. This work was also supported by a Fonds de Recherche du Québec – Nature et Technologies 3rd cycle scholarship, award #304858.

Appendix

A. Abstract

This artifact contains a source tree and a Dockerfile. Docker can assemble these components into a container that includes the LIAR implementation and its dependencies. The container is designed to reproduce this paper's experimental results.

B. Artifact Check-List (Meta-Information)

- **Algorithm:** An equality saturation–based idiom recognition and rewriting algorithm for a minimalist functional array IR.
- **Compilation:** Docker builds and loads the artifact. We used Docker version 20.10.21.
- **Data set:** PolyBench/C 4.2.1-beta kernels and custom kernels, included in the artifact.
- **Hardware:** A system with one or more x86-64 CPUs. We used a server with two 18-core Intel Xeon Gold 6254 CPUs.
- **Metrics:** Library calls found, saturation steps, e-node counts, kernel execution times.
- **Output:** Kernel optimization and execution logs, graphs and tables generated from log data.
- **Experiments:** Kernel optimization with and without idiom recognition (section VI-B, section VI-C); and optimized kernel execution (section VI-D, section VI-E).
- **How much time is needed to complete experiments (approximately)?:** Approximately 12–18 hours, varies depending on CPU.
- **Publicly available?:** Yes
- **Code licenses?:** MIT license
- **Archived (provide DOI)?:** 10.5281/zenodo.8316752

C. Description

1) How Delivered: The artifact is available both on Zenodo (DOI: 10.5281/zenodo.8316752) and in the `cgo24-artifact` branch of the `cdubach/shir` BitBucket repository. Clone it as follows:

```
$ git clone --recursive -b cgo24-artifact \
    https://bitbucket.org/cdubach/shir.git
```

2) Hardware Dependencies: An x86-64 CPU.

3) Software Dependencies: A Docker installation.

D. Installation

Build and run a Docker container from the artifact. In the artifact directory, run the following commands:

```
$ docker build -t liar-image .
$ docker run --name liar-ubuntu -i -t liar-image bash
```

E. Experiment Workflow

To replicate our findings, we recommend a workflow consisting of three steps.

1) Time-Limited Optimization: The first step is to optimize the evaluation's kernels using the same methodology as described in section VI. That methodology is to optimize each kernel for five minutes for each target (pure C, BLAS idioms or PyTorch idioms). This experiment will take four hours and its results vary based on single-thread CPU performance. Faster CPUs will be able to perform more saturation steps and find more advanced solutions.

To perform this experiment, run the following commands in the running Docker container:

```
$ mkdir unlimited-steps
$ cd unlimited-steps
$ ../artifact/src/main/drivers/evaluate_all.py \
    -t300 --optimize-only
$ cd ..
```

2) Step-Limited Optimization: Since time-limited optimization delivers CPU-dependent results, we recommend a step-limited optimization phase to more precisely replicate the findings from table II and table III. This second step performs the same optimization process as before, but in a way that produces CPU-invariant results at the cost of requiring a CPU-dependent amount of time. This trade-off results from a saturation step limit instead of a time limit. The baked-in step limits are chosen to correspond to the steps reported in the tables.

To perform this second step of the evaluation, run the following:

```
$ mkdir limited-steps
$ cd limited-steps
$ ../artifact/src/main/drivers/evaluate_all.py \
    -t3600 --limit-steps --optimize-only
$ cd ..
```

3) Kernel Execution: The third step of the workflow involves running the optimized kernels. We recommend starting from the step-limited results, since these match the solutions reported in section VI. The following commands will run each solution for one minute, requiring a total of 4½ hours:

```
$ cd limited-steps
$ ../artifact/src/main/drivers/evaluate_all.py \
    -t3600 --limit-steps --build-paper
$ cd ..
```

This invocation of the evaluation script will report that it is reusing the optimized kernels from the previous step and then run each solution for each kernel. Once each solution has been run, the script regenerates this paper from the updated results, allowing for easy inspection of the affected tables and figures.

F. Evaluation and Expected Result

Running the evaluation workflow described in the previous section will fill the `unlimited-steps` and `limited-steps` directories with subdirectories containing the following:

1) Logs for each target, stored in the `blas-logs`, `pytorch-logs` and `none-logs` directories. These directories respectively correspond to the BLAS, PyTorch

and pure C targets. Each log contains detailed information for every equality saturation step. Log files whose names have a `cov-` prefix are derived from their `log-` equivalents by augmenting them with the results of kernel execution.

2) Aggregate data and plots, stored in the `plots` directory.

The tables and figures in this table are derived from the files in the latter directory.

- Table II is obtained by arranging into a table the following columns from `blas-overview.csv`: `name`, `externs`, `steps`, and `nodes`.
- Table III is similarly obtained from the data in `pytorch-overview.csv`.
- Figure 4 is derived by including in a TeX PGFPlots line chart the commands from `{nodes, comp-time}-over-time-gemv-{blas, pytorch}.tex`.
- Figure 5 is a PGFPlots bar chart constructed from the commands in `coverage-over-time-gemv-blas.tex`.
- Figure 6 is constructed by creating a PGFPlots line chart from the coordinates in `run-time-gemv-blas.tex`, cropped to solutions three through six.
- Figure 7 is a PGFPlots bar chart visualization of the commands in `speedup-bars.tex`.

These tables and figures are generated afresh from locally-computed results by invoking the evaluation script with `--build-paper`. The experiment workflow makes use of this flag and stores the resulting paper as `paper/conference_101719.pdf` in the `limited-steps` directory.

G. Experiment Customization

Experiments can be restricted to specific kernels by passing the names of the kernels to the evaluation script, like so:

```
$ ../artifact/src/main/drivers/evaluate_all.py \
    mvt -t3600 --limit-steps
```

H. Notes

A known discrepancy between the evaluation as performed in section VI and the artifact's Docker container is that the container relies on Ubuntu whereas section VI runs directly on a CentOS system. We do not believe this difference significantly affects results.

References

[1] L Susan Blackford et al. "An Updated Set of Basic Linear Algebra Subprograms (BLAS)". In: *ACM Trans. Math. Softw.* 28.2 (June 2002), pp. 135–151. ISSN: 0098-3500. DOI: 10.1145/567806.567807. URL: https://doi.org/10.1145/567806.567807.

[2] Nicolaas Govert De Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)*. Vol. 75. 5. Elsevier. 1972, pp. 381–392.

[3] João P. L. De Carvalho et al. "KernelFaRer: Replacing Native-Code Idioms with High-Performance Library Calls". In: *ACM Trans. Archit. Code Optim.* 18.3 (June 2021). ISSN: 1544-3566. DOI: 10.1145/3459010. URL: https://doi.org/10.1145/3459010.

[4] Vinicius Espindola et al. "Source Matching and Rewriting for MLIR Using String-Based Automata". In: *ACM Trans. Archit. Code Optim.* 20.2 (Mar. 2023). ISSN: 1544-3566. DOI: 10.1145/3571283. URL: https://doi.org/10.1145/3571283.

[5] Philip Ginsbach et al. "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*. ASPLOS '18. Williamsburg, VA, USA: Association for Computing Machinery, 2018, pp. 139–153. ISBN: 9781450349116. DOI: 10.1145/3173162.3173182. URL: https://doi.org/10.1145/3173162.3173182.

[6] Troels Henriksen et al. "Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 556–571. ISBN: 9781450349888. DOI: 10.1145/3062341.3062354. URL: https://doi.org/10.1145/3062341.3062354.

[7] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. "Playing by the rules: rewriting as a practical optimisation technique in GHC". In: *Haskell workshop*. Vol. 1. 2001, pp. 203–233.

[8] Motohiro Kawahito et al. "Idiom Recognition Framework Using Topological Embedding". In: *ACM Trans. Archit. Code Optim.* 10.3 (Sept. 2013). ISSN: 1544-3566. DOI: 10.1145/2512431. URL: https://doi.org/10.1145/2512431.

[9] Thomas Koehler, Phil Trinder, and Michel Steuwer. "Sketch-Guided Equality Saturation: Scaling Equality Saturation to Complex Optimizations in Languages with Bindings". In: *arXiv preprint arXiv:2111.13040* (2021).

[10] Zhitao Lin and Christophe Dubach. "From Functional to Imperative: Combining Destination-Passing Style and Views". In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. ARRAY 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 25–36. ISBN: 9781450392693. DOI: 10.1145/3520306.3534502. URL: https://doi.org/10.1145/3520306.3534502.

[11] Chandrakana Nandi et al. "Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 31–44. ISBN: 9781450376136. DOI: 10.1145/3385412.3386012. URL: https://doi.org/10.1145/3385412.3386012.

[12] Chandrakana Nandi et al. "Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 31–44. ISBN: 9781450376136. DOI: 10.1145/3385412.3386012. URL: https://doi.org/10.1145/3385412.3386012.

[13] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[14] Shlomit S. Pinter and Ron Y. Pinter. "Program Optimization and Parallelization Using Idioms". In: *ACM Trans. Program. Lang. Syst.* 16.3 (May 1994), pp. 305–327. ISSN: 0164-0925. DOI: 10.1145/177492.177494. URL: https://doi.org/10.1145/177492.177494.

[15] Louis-Noël Pouchet and Tomofumi Yuki. *PolyBench/C: the Polyhedral Benchmark suite*. Feb. 8, 2016. URL: http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[16] Jonathan Ragan-Kelley et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 519–530. ISBN: 9781450320146. DOI: 10.1145/2491956.2462176. URL: https://doi.org/10.1145/2491956.2462176.

[17] River Riddle et al. *'pdl' Dialect*. May 10, 2023. URL: https://mlir.llvm.org/docs/Dialects/PDLOps/.

[18] Amir Shaikhha et al. "Destination-Passing Style for Efficient Memory Management". In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*. FHPC 2017. Oxford, UK: Association for Computing Machinery, 2017, pp. 12–23. ISBN: 9781450351812. DOI: 10.1145/3122948.3122949. URL: https://doi.org/10.1145/3122948.3122949.

[19] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "LIFT: A functional data-parallel IR for high-performance GPU code generation".

In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2017, pp. 74–85. DOI: 10.1109/CGO.2017.7863730.

[20] Ross Tate et al. “Equality Saturation: A New Approach to Optimization”. In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’09. Savannah, GA, USA: Association for Computing Machinery, 2009, pp. 264–276. ISBN: 9781605583792. DOI: 10.1145/1480881.1480915. URL: https://doi.org/10.1145/1480881.1480915.

[21] Jonathan Van der Cruysse and Christophe Dubach. *Latent Idiom Recognition for a Minimalist Functional Array Language using Equality Saturation*. Zenodo, Sept. 2023. DOI: 10.5281/zenodo.8316752.

[22] Alexa VanHattum et al. “Vectorization for Digital Signal Processors via Equality Saturation”. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’21. Virtual, USA: Association for Computing Machinery, 2021, pp. 874–886. ISBN: 9781450383172. DOI: 10.1145/3445814.3446707. URL: https://doi.org/10.1145/3445814.3446707.

[23] Yisu Remy Wang et al. “SPORES: sum-product optimization via relational equality saturation for large scale linear algebra”. In: *Proceedings of the VLDB Endowment* 13.12 (2020).

[24] Max Willsey et al. “egg: Fast and Extensible Equality Saturation”. In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434304. URL: https://doi.org/10.1145/3434304.

[25] Yichen Yang et al. “Equality Saturation for Tensor Graph Superoptimization”. In: *Proceedings of Machine Learning and Systems*. Ed. by A. Smola, A. Dimakis, and I. Stoica. Vol. 3. 2021, pp. 255–268. URL: https://proceedings.mlsys.org/paper/2021/file/65ded5353c5ee48d0b7d48c591b8f430-Paper.pdf.