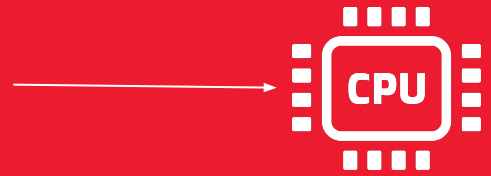
Latent Idiom Recognition for a Minimalist Functional Array Language using Equality Saturation

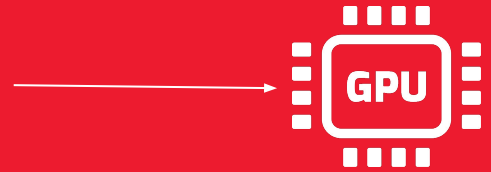
No library

```
for (i = 0; i < N; i++) {
  sum += xs[i];
}
```



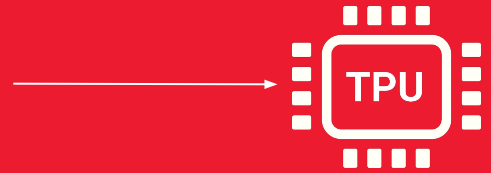
BLAS

```
double ones[N] = { 1, 1, … };
ddot(N, xs, 1, ones, 1);
```



PyTorch

```
dot(xs, full(N, 1))
-OR-
sum(xs)
```



McGill

Jonathan Van der Cruysse and Christophe Dubach

McGill University – CGO '24 – Tuesday March 5, 2024

Levels of abstraction

Domain-specific language

```
dot(xs, ys)
```

Functional array programming

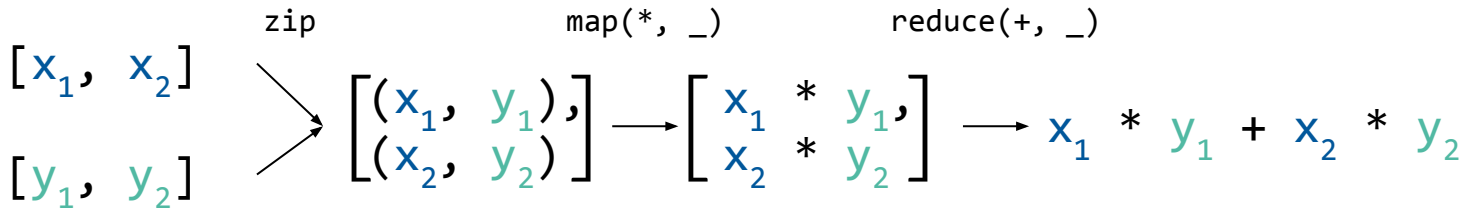
```
reduce(+, map(*, zip(xs, ys)))
```

Systems programming

```
sum = 0.0;
for (i = 0; i < N; i++) {
  sum += xs[i] * ys[i];
}
```

Functional array programming

```
dot(xs, ys) = reduce(+, map(*, zip(xs, ys)))
```



Idiom recognition

```
dot(xs, ys) = reduce(+, map(*, zip(xs, ys)))
```

Dot product

```
reduce(+, map(*, zip(xs, ys)))
```

→ `dot(xs, ys)`

Vector norm

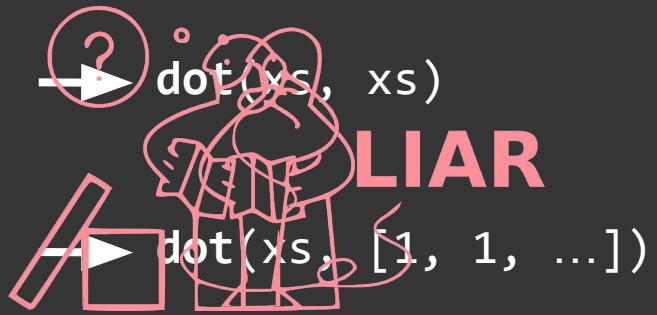
```
reduce(+, map(λx. x * x, xs))
```

→ `dot(xs, xs)`

Array sum

```
reduce(+, xs)
```

→ `dot(xs, [1, 1, ...])`



Three ingredients

Minimalist IR

map
reduce
zip
join
split
concat
…

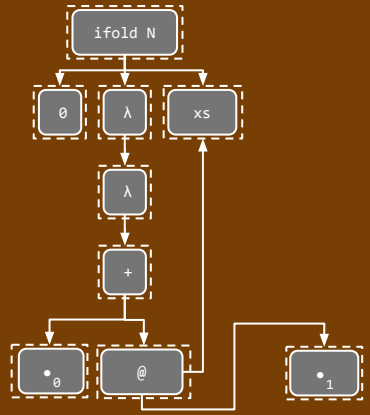
build
ifold

Language rules + library idioms

`build(N, f)@i = f(i)`
(8 rules in total)

`fill(c) = build(N, λi. c)`
(10 for BLAS, 11 for PyTorch)

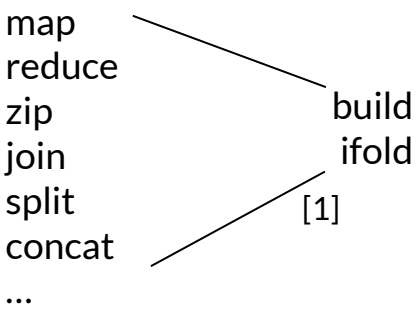
Equality saturation



Ingredient #1: Minimalist IR

```
build(3, f) = [f(0), f(1), f(2)]

ifold(3, f, init) = f(f(f(init, 0), 1), 2)
```



[1]: A. Shaikhha et al. Destination-passing style for efficient memory management. **FHCP 2017**

Idiom recognition on minimalist IR

```
dot(xs, ys) = reduce(+, 0, λi. a + xs@i * ys@i)
```

Dot product

```
reduce(+, 0, λi. a + xs@i * ys@i)  →  dot(xs, ys)
reduce(+, map(*, zip(xs, ys)))
```

Vector norm

```
reduce(+, 0, λi. a + xs@i * xs@i)  →  dot(xs, xs)
reduce(+, map(λx. x * x, xs))
```

Array sum

```
reduce(+, 0, λi. a + xs@i)
reduce(+, 0, xs)
```

Ingredient #2: Rules and idioms

Language rules

```
x = x * 1
```

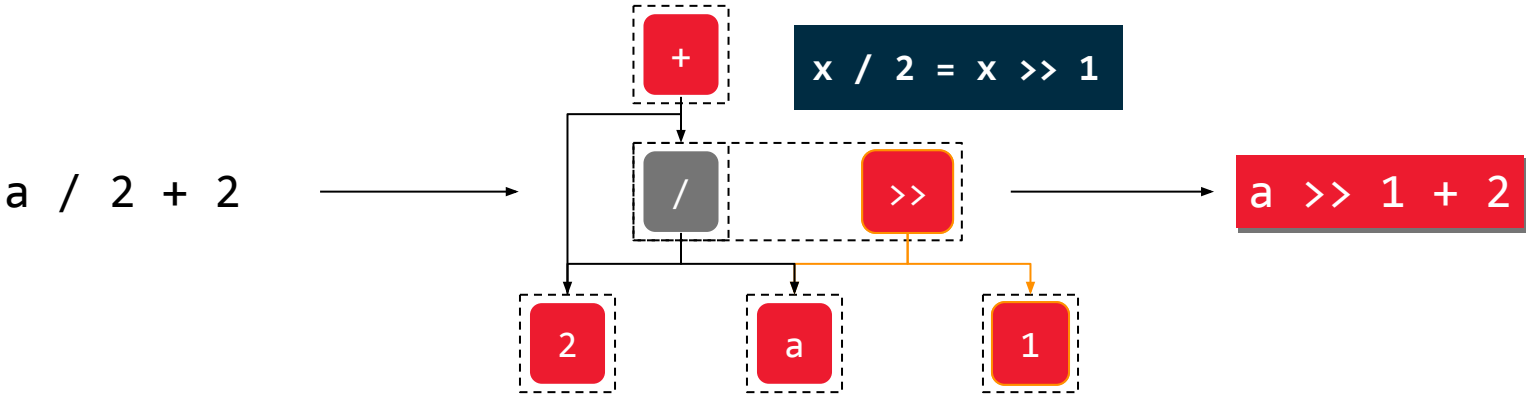
```
build(N, f)@i = f(i)
```

Library idioms (e.g., PyTorch, BLAS)

```
dot(A, B) =
  ifold(N, λa. λi. A@i * B@i + a)
```

```
fill(c) = build(N, λi. c)
```

Ingredient #3: Equality saturation



[2]: R. Tate et al. Equality saturation: a new approach to optimization. **POPL '09**

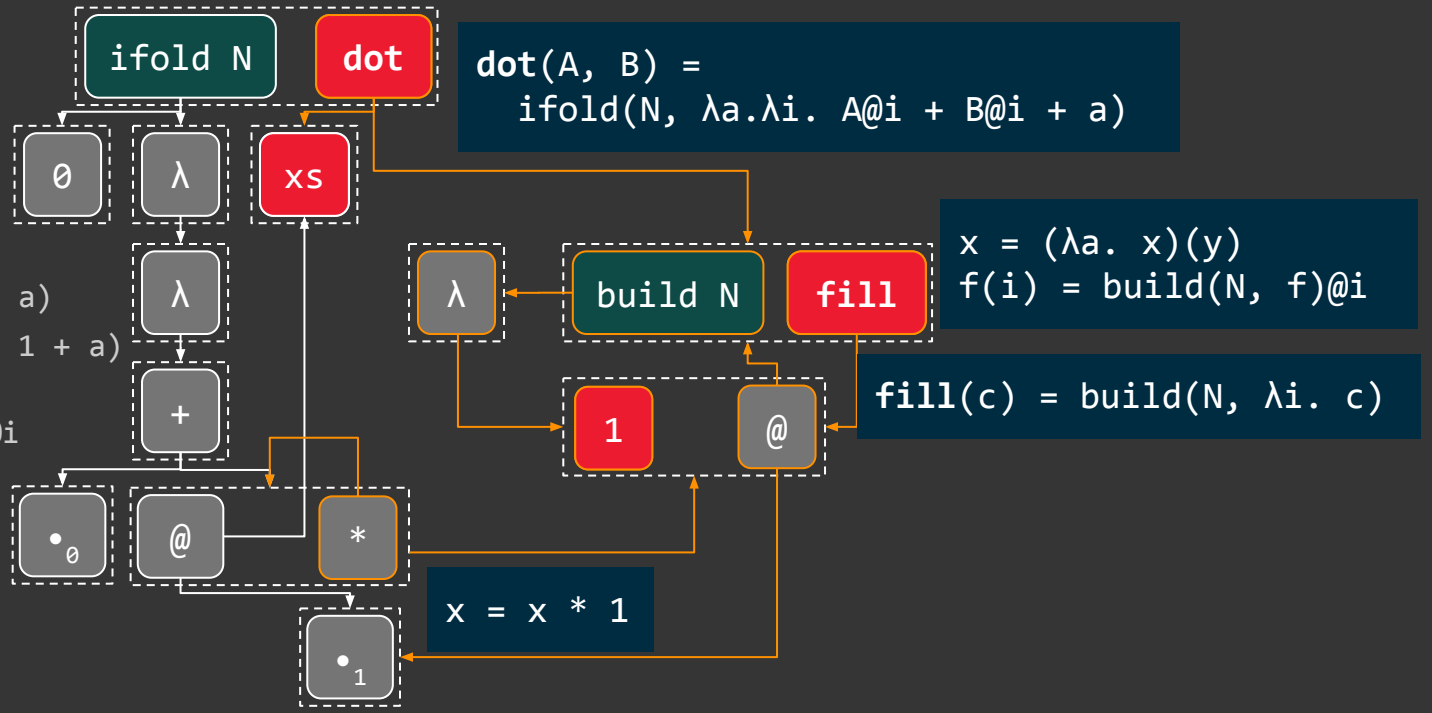
Combining the three ingredients

Expressions in e-graph:

```
ifold(N, 0, λa.λi. xs@i + a)
ifold(N, 0, λa.λi. xs@i * 1 + a)
ifold(N, 0, λa.λi.
  xs@i * build(N, λj. 1)@i
  + a)

ifold(N, 0, λa.λi.
  xs@i * fill(1)@i + a)
```

dot(xs, fill(1))



Evaluation methodology

PolyBench C

```
sum = 0.0;
for (i = 0; i < N; i++) {
  sum += xs[i] * ys[i];
}
```

Minimalist IR

```
ifold N 0 (λa. λi. a + xs@i)
```

LIAR

IR with library calls

BLAS
`ddot(xs, [1, 1, …])`

PyTorch
`sum(xs)`

[3]

C code

BLAS

```
double ones[N] = { 1, 1, … };
ddot(N, xs, 1, ones, 1);
```

[3]: Z. Lin, C. Dubach. From functional to imperative: combining destination-passing style and views. **ARRAY 2022**

PolyBench: `doitgen` example

```
for (int r = 0; r < R; r++) {
  for (int q = 0; q < Q; q++) {
    for (int p = 0; p < P; p++) {
      sum[p] = 0.0;
      for (int s = 0; s < P; s++) {
        sum[p] += A[r][q][s] * B[s][p];
      }
    }
    for (int p = 0; p < P; p++) {
      A[r][q][p] = sum[p];
    }
  }
}
```

PolyBench: `doitgen` example

```
build(N, λr. build(N, λq. build(N, λp.
  ifold(N, 0, λa. λs. A@r@q@s * B@s@p + a))))
```

Targeting PyTorch

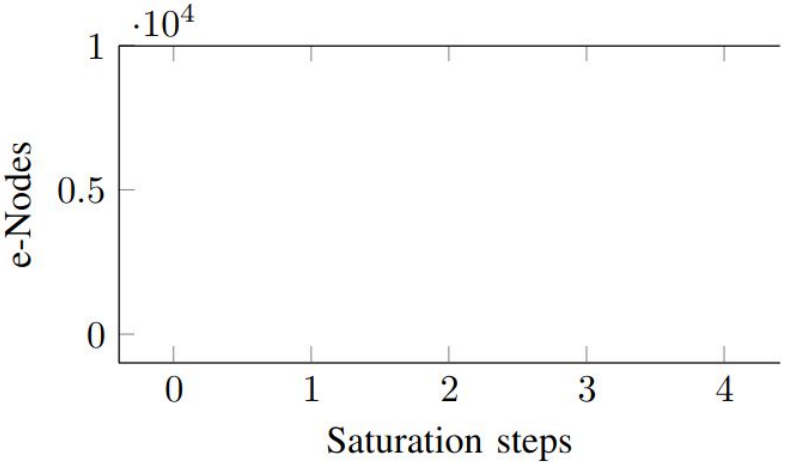
```
build(N, λr. mm(A@r, transpose(B)))
```

Targeting BLAS

```
build(N, λr. dgemm^F,T(1, A@r, B, 0, build(N, λq. memset(0))))
```

Solutions over time

doitgen benchmark

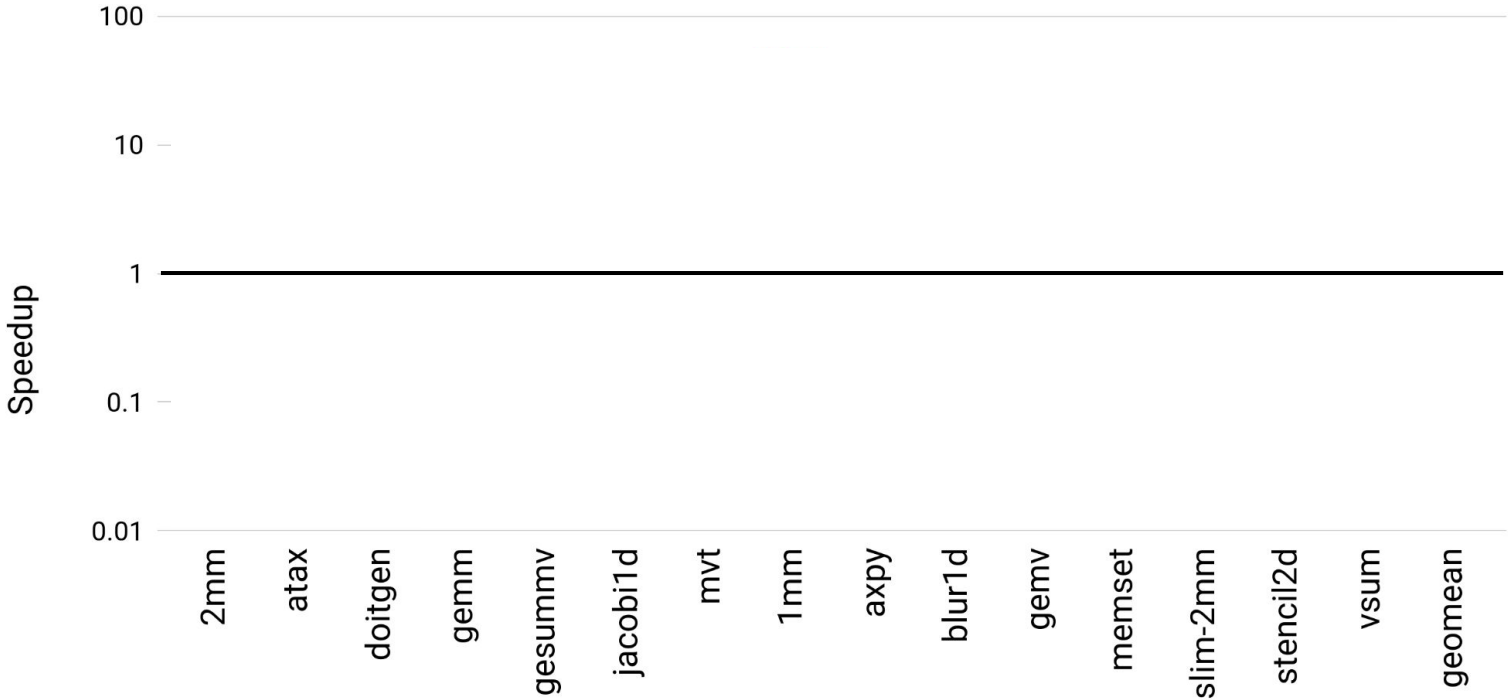


PyTorch

BLAS

Run time speedup

Compared to PolyBench C kernels, BLAS parallel CPU library



Conclusion

Jonathan's home page

- **Minimalist IR** simplifies semantics
- **Language rules + library idioms** scheme adapts to different APIs
- **Equality saturation** transforms programs to find idioms
- **More details in paper**
  - De Bruijn indices, full rules + idioms, etc.
- **Future work:** Improve scalability of equality saturation

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.

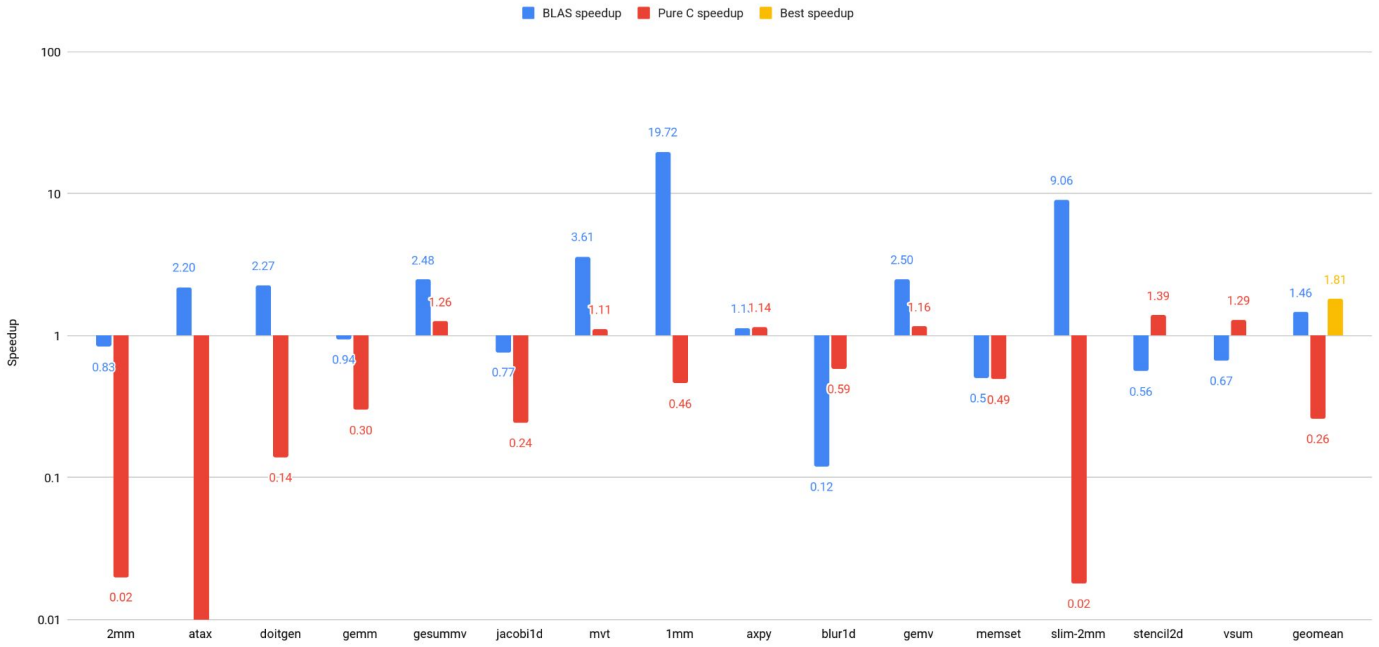
This paper shows how this technique can be applied on a subset of the Basic Linear Algebra Subprograms (BLAS) [1] and PyTorch [13] libraries. To evaluate LIAR's effectiveness, this work applies it to custom kernels, and to linear algebra and numerical simulation kernels from the PolyBench suite. The 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.

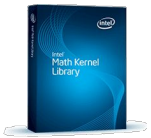
Run time speedup

Compared to reference kernels



BLAS and PyTorch

BLAS 



- Library specification
- Linear algebra
- Complex API

```
dgemm(
  transa, transb,
  m, n, k,
  alpha, a, lda,
  b, ldb,
  beta, c, ldc)
```

PyTorch 

- Library
- Machine learning
- Streamlined API

```
mm(a, b)
add(a, b)
mul(a, b)
transpose(a)
```

LIAR in action

```
dot(xs, ys) = ifold N 0 (λa. λi. a + xs_i * ys_i)
```

Dot product

```
ifold N 0 (λa. λi. a + xs_i * ys_i)
reduce (+) (map (*) (zip xs ys))
```

→ `dot(xs, ys)`

Vector norm

```
ifold N 0 (λa. λi. a + xs_i * xs_i)
reduce (+) (map (λx. x * x) xs)
```

→ `dot(xs, xs)`

Array sum

```
ifold N 0 (λa. λi. a + xs_i)
reduce (+) 0 xs
```

→ `dot(xs, fill(1))`

Equality saturation pipeline

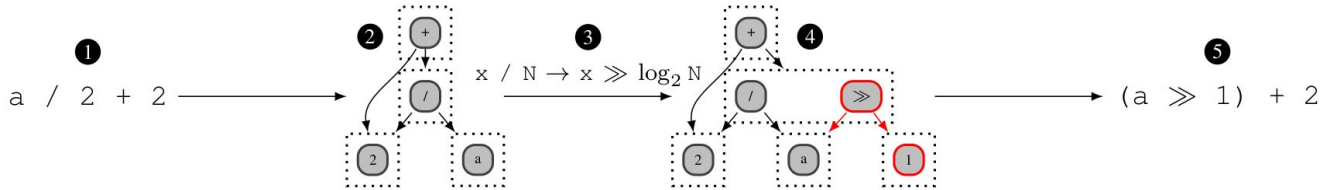


Fig. 1: Expression ❶ is converted to e-graph ❷, which is subsequently saturated. In this example, only rule ❸ is applied: `x / N` → `x ≫` $\log_2$ `N`. From saturated e-graph ❹, expression ❺ is selected by an extractor that prefers bitwise shift.

LIAR pipeline overview

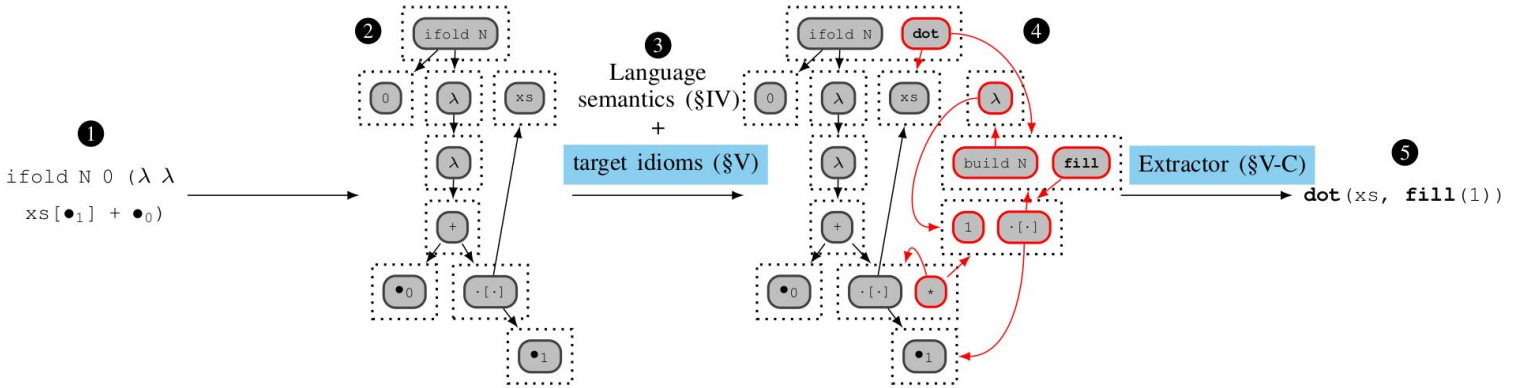


Fig. 2: Overview of LIAR, the proposed technique. Vector sum expression ❶ is converted to e-graph ❷. Equality saturation applies a set of rules ❸ to e-graph ❷. These rules consist of target-independent language semantics and target-specific idioms. Rule application yields updated e-graph ❹. From e-graph 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.

IR grammar

```
e ::= λ e                 lambda abstraction
    | e e                 lambda application
    | •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(ē)                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.

IR semantics

```
(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.

```
(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.

Arithmetic identities

`x + 0 = x`	(E-AddZero)
`1 * x = x`	(E-MulOneL)
`x * 1 = x`	(E-MulOneR)
`x * y = 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.

BLAS and PyTorch idioms

```
axpy(alpha, A, B)
 = build N (λ alpha↑ * A↑[•0] + B↑[•0])                    (I-AXPY)
dot(A, B)
 = ifold N 0 (λ λ A↑↑[•1] * B↑↑[•1] + •0)                   (I-DOT)
gemv^F(alpha, A, B, beta, C)
 = build N (λ alpha↑ * dot(A↑[•0], B↑) + beta↑ * C↑[•0])
                                                           (I-GEMV)
gemm^F,T(alpha, A, B, beta, C)
 = build N (λ gemv^N(alpha↑, B↑, A↑[•0], beta↑, C↑[•0]))
                                                           (I-GEMM)
transpose(A)
 = build N (λ build M (λ A↑↑[•0][•1]))                     (I-TRANSPOSE)
gemv^X(alpha, transpose(A), B, beta, c)
 = gemv^¬X(alpha, A, B, beta, c)                           (I-TRANSPOSEINGEMV)
gemm^X,Y(alpha, transpose(A), B, beta, C)
 = gemm^¬X,Y(alpha, A, B, beta, C)                         (I-TRANSPOSEAINGEMM)
gemm^X,Y(alpha, A, transpose(B), beta, c)
 = gemm^X,¬Y(alpha, A, B, beta, c)                         (I-TRANSPOSEBINGEMM)
dot(build N (λ alpha * A[•0]), B)
 = alpha * dot(A, B)                                       (I-HOISTMULFROMDOT)
memset(0)
 = build N (λ 0)                                           (I-MEMSETZERO)
```

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

```
dot(A, B)
 = ifold N 0 (λ λ A↑↑[•1] * B↑↑[•1] + •0)                   (I-DOT)
sum(A)
 = ifold N 0 (λ λ A↑↑[•1] + •0)                             (I-VECSUM)
mv(A, B)
 = build N (λ dot(A↑[•1], B↑))                             (I-MATVEC)
mm(A, B)
 = build N (λ mv(B↑, A↑[•1]))                              (I-MATMAT)
transpose(A)
 = build N (λ build M (λ A↑↑[•0][•1]))                     (I-TRANSPOSE)
transpose(transpose(A))
 = A                                                       (I-TRANSPOSETWICE)
add(A, B)
 = build N (λ A↑[•0] + B↑[•0])                             (I-ADDVEC)
add(A, B)
 = build N (λ add(A↑[•0], B↑[•0]))                         (I-LIFTADD)
mul(alpha, A)
 = build N (λ alpha * A↑[•0])                              (I-MULSCALARANDVEC)
mul(alpha, A)
 = build N (λ mul(alpha, A↑[•0]))                          (I-LIFTMUL)
full(c)
 = build N (λ c↑)                                          (I-FULLVEC)
```

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

Cost model

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

Listing 6: Definition of the base cost function.

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

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.

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

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.

Overview of kernels

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

TABLE I: Overview of kernels examined in this work. PolyBench kernel descriptions adapted from benchmark suite [13].

Solutions found

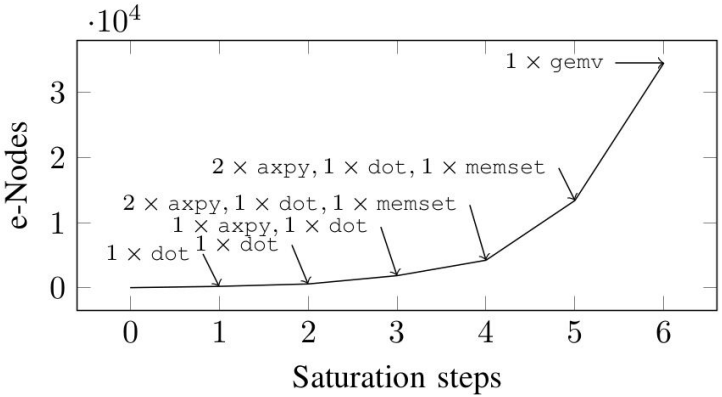
Kernel	Solution	Steps	e-Nodes
2mm	$3 \times$ axpy + $1 \times$ dot + $1 \times$ gemv + $3 \times$ memset + $1 \times$ transpose	6	3.44×10^4
atax	$2 \times$ gemv + $2 \times$ memset	6	1.49×10^4
doitgen	$1 \times$ gemm + $1 \times$ memset	7	2.14×10^4
gemm	$3 \times$ axpy + $1 \times$ gemv + $3 \times$ memset	6	1.93×10^4
gemver	$3 \times$ axpy + $2 \times$ dot + $1 \times$ memset	5	1.74×10^4
gesummv	$1 \times$ dot + $1 \times$ gemv	6	1.70×10^4
jacobi1d	$1 \times$ gemv + $1 \times$ memset	5	2.53×10^4
mvt	$2 \times$ gemv + $2 \times$ memset	7	2.69×10^4
1mm	$1 \times$ gemm + $1 \times$ memset	7	2.05×10^4
axpy	$1 \times$ axpy	13	2.57×10^4
blur1d	$1 \times$ gemv + $1 \times$ memset	5	1.71×10^4
gemv	$1 \times$ gemv	7	3.46×10^4
memset	$1 \times$ memset	19	2.86×10^4
slim-2mm	$1 \times$ gemm + $1 \times$ gemv + $2 \times$ memset + $2 \times$ transpose	6	2.04×10^4
stencil2d	$1 \times$ gemv + $1 \times$ memset	5	5.88×10^4
vsum	$1 \times$ dot	10	1.59×10^4

TABLE II: Solutions found for kernels when targeting BLAS. *Steps* describes the number of saturation steps applied to the kernel. *Solution* describes the external library calls found at the last step. *e-Nodes* counts the number of unique e-nodes in the e-graph, also at the last step.

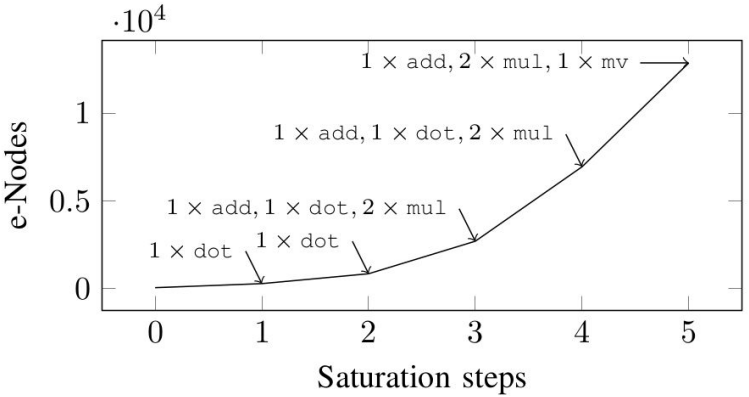
Kernel	Solution	Steps	e-Nodes
2mm	$1 \times$ add + $2 \times$ mul + $2 \times$ mv + $2 \times$ transpose	5	2.28×10^4
atax	$2 \times$ mv + $1 \times$ transpose	7	1.98×10^4
doitgen	$1 \times$ mm + $1 \times$ transpose	6	1.47×10^4
gemm	$1 \times$ add + $2 \times$ mul + $1 \times$ mv + $1 \times$ transpose	5	1.55×10^4
gemver	$2 \times$ add + $2 \times$ dot + $1 \times$ mul	4	9.06×10^3
gesummv	$1 \times$ add + $2 \times$ mul + $2 \times$ mv	6	1.54×10^4
jacobi1d	$1 \times$ full + $1 \times$ mv	5	3.13×10^4
mvt	$2 \times$ mv + $1 \times$ transpose	7	1.69×10^4
1mm	$1 \times$ mm	6	1.01×10^4
axpy	$1 \times$ add + $1 \times$ mul	9	1.52×10^4
blur1d	$1 \times$ full + $1 \times$ mv	5	2.13×10^4
gemv	$1 \times$ add + $2 \times$ mul + $1 \times$ mv	6	1.29×10^4
memset	$1 \times$ full	15	8.30×10^3
slim-2mm	$2 \times$ mv + $2 \times$ transpose	5	1.09×10^4
stencil2d	$1 \times$ mv	4	1.92×10^4
vsum	$1 \times$ sum	9	9.44×10^3

TABLE III: Solutions found for kernels when targeting PyTorch. Columns have the same meaning as in table II.

Solutions over time



(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. Arrows indicate that a new best solution has been found, and are labeled with the external calls in that solution.

Coverage over time

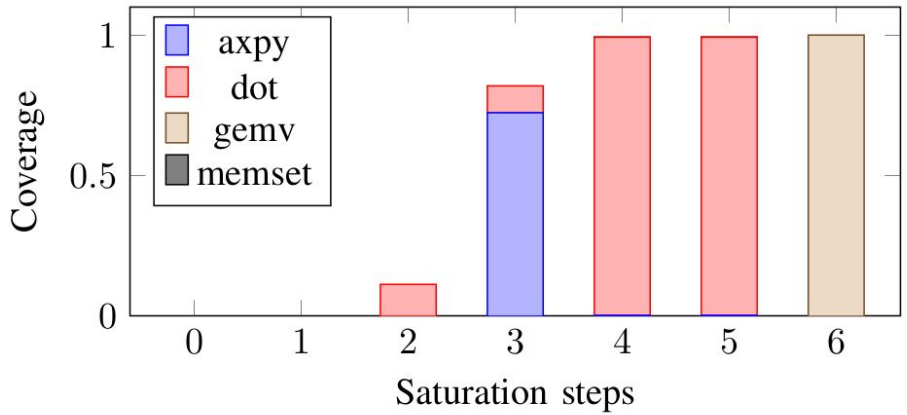


Fig. 5: Coverage over time for the *gemv* kernel, targeting BLAS. The x-axis shows saturation steps; the stacked bars depict the ratio of time spent in library functions. Higher bars indicate that more computation is offloaded to a library.

Run time over time

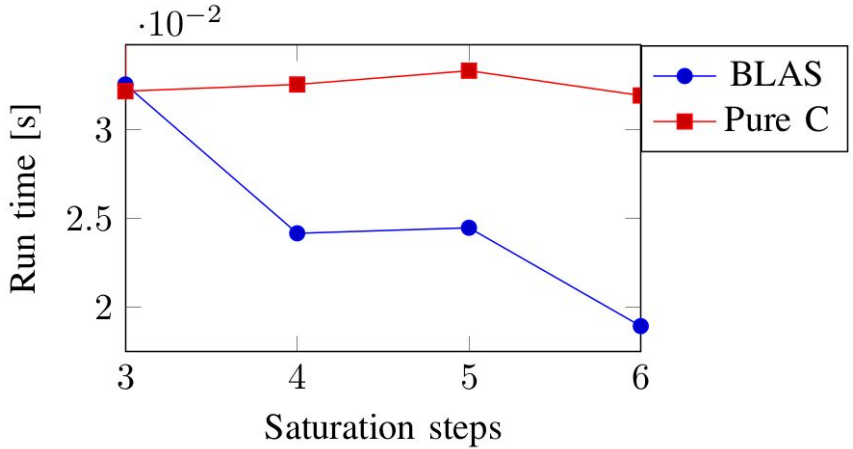


Fig. 6: Run time as a function of saturation steps. The x-axes show saturation steps; the y-axes depict the amount of time required to execute the solutions. Lower is better.

Run time speedup

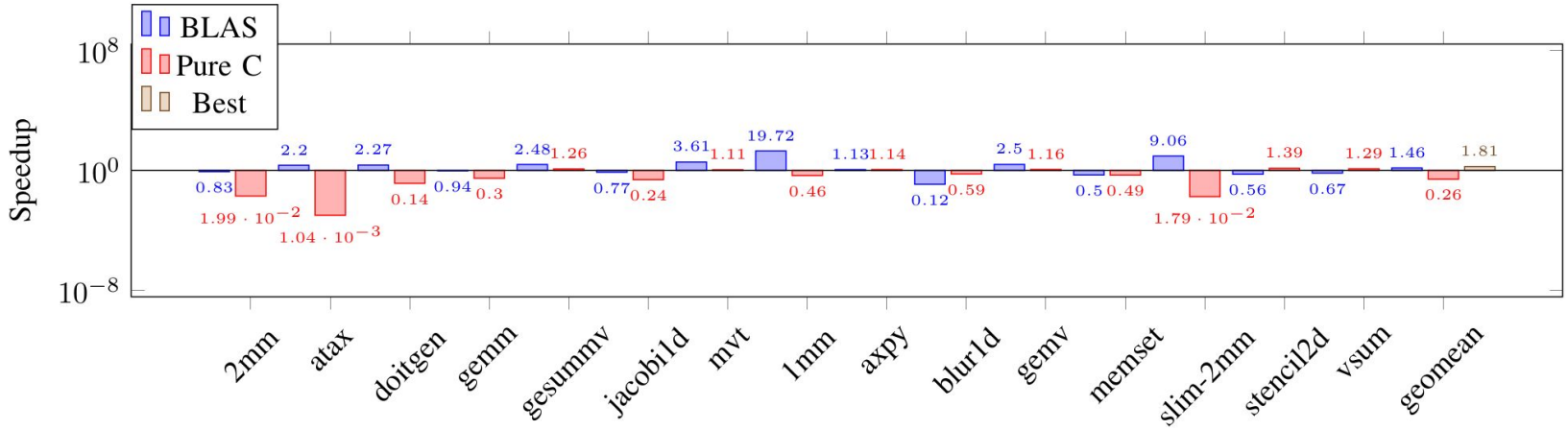


Fig. 7: Run time speedup of LIAR's solutions compared to reference implementations in C. For PolyBench kernels, the reference implementations are the original benchmarks. The reference for custom benchmarks is a hand-written C program coded in the style of a PolyBench kernel. Each bar represents the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

Functional array programming

map f $[x_1, x_2] = [f\ x_1, f\ x_2]$

reduce f init $[x_1, x_2] = f\ (f\ \text{init}\ x_1)\ x_2$

zip $[x_1, x_2]\ [y_1, y_2] = [(x_1, y_1), (x_2, y_2)]$

Also: `join`, `split`, `concat`, `slide`, …

Rewrite rules: naive approach

```
map g (map f as) → map (g ∘ f) as

zip (map f as) (map g bs) → map (λt. (f t₁, g t₂)) (zip as bs)

map (λa. f a a) as → map (λt. f t₁ t₂) (zip as as)

map (λa. f a c) as → map (λt. f t₁ t₂) (zip as [c, c, …])

as → map (λa. a) as
```

How many rules do we need?

De Bruijn indices

Named variables

```
λx. x
λx. λy. y
λx. λy. x
λx. λy. λz. z
```

De Bruijn indices

```
λ •0
λ λ •0
λ λ •1
λ λ λ •2
```

Latent idioms

Original computation

2D convolution

PyTorch

```
conv2d(
  A,
  weights
)
```

BLAS

```
gemm(
  im2col(...),
  weights’
)
```