KernelFaRer: Replacing Native-Code Idioms with High-Performance Library Calls

JOÃO P. L. DE CARVALHO, University of Campinas (UNICAMP), Brazil
BRAEDY KUZMA, IVAN KOROSTELEV, and JOSÉ NELSON AMARAL, University of Alberta, Canada
CHRISTOPHER BARTON, IBM Corporation, Canada
JOSÉ MOREIRA, IBM Corporation, USA
GUIDO ARAUJO, University of Campinas (UNICAMP), Brazil

Well-crafted libraries deliver much higher performance than code generated by sophisticated application programmers using advanced optimizing compilers. When a code pattern for which a well-tuned library implementation exists is found in the source code of an application, the highest performing solution is to replace the pattern with a call to the library. Idiom-recognition solutions in the past either required pattern matching machinery that was outside of the compilation framework or provided a very brittle solution that would fail even for minor variants in the pattern source code. This article introduces Kernel Find & Replacer (KernelFaRer), an idiom recognizer implemented entirely in the existing LLVM compiler framework. The versatility of KernelFaRer is demonstrated by matching and replacing two linear algebra idioms, general matrix-matrix multiplication (GEMM), and symmetric rank-2k update (SYR2K). Both GEMM and SYR2K are used extensively in scientific computation, and GEMM is also a central building block for deep learning and computer graphics algorithms. The idiom recognition in KernelFaRer is much more robust than alternative solutions, has a much lower compilation overhead, and is fully integrated in the broadly used LLVM compilation tools. KernelFaRer replaces existing GEMM and SYR2K idioms with computations performed by BLAS, Eigen, MKL (Intel's x86), ESSL (IBM's PowerPC), and BLIS (AMD). Gains in performance that reach **2000×** over hand-crafted source code compiled at the highest optimization level demonstrate that replacing application code with library call is a performant solution.

CCS Concepts: • **Software and its engineering** → **Compilers**; *Runtime environments*; • **Theory of computation** → *Massively parallel algorithms;*

Additional Key Words and Phrases: Idiom recognition, compiler analysis and transformations, GEMM, LLVM

ACM Reference format:
João P. L. de Carvalho, Braedy Kuzma, Ivan Korostelev, José Nelson Amaral, Christopher Barton, José Moreira, and Guido Araujo. 2021. KernelFaRer: Replacing Native-Code Idioms with High-Performance Library Calls. *ACM Trans. Arch. Code Optim.* 18, 3, Article 38 (June 2021), 22 pages.
https://doi.org/10.1145/3459010

J. P. L. de Carvalho also with University of Alberta.
Authors' addresses: J. P. L. de Carvalho and G. Araujo, University of Campinas (UNICAMP), Campinas, SP, Brazil; emails: {joao.carvalho, guido}@ic.unicamp.br; B. Kuzma, I. Korostelev, and J. N. Amaral, University of Alberta, Edmonton, AB, Canada; emails: {braedy, korostel, jamaral}@ualberta.ca; C. Barton, IBM Corporation, Markham, ON, Canada; email: kbarton@ca.ibm.com; J. Moreira, IBM Corporation, Yorktown, NY; email: jmoreira@us.ibm.com.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Association for Computing Machinery.
1544-3566/2021/06-ART38
https://doi.org/10.1145/3459010

1 INTRODUCTION

General matrix-matrix multiplication (GEMM) and **symmetric rank-2k update (SYR2K)** are simple to implement naively but complex when designed to optimize the memory hierarchy [1]. GEMM in particular is idiomatic, because it can be succinctly expressed and it exhibits a direct relation between implementation complexity and performance [2]. Most generic compiler loop transformations fail to exploit specific features of programming idioms such as GEMM [3, 4]. This article reaffirms that both sophisticated programmers and compilers still do not generate code with the performance of well-tuned libraries [5]. Thus, instead of optimizing an idiom, a compiler may simply replace it with a call to a library such as the **Basic Linear Algebra Subprograms (BLAS)** library [6]. These expertly crafted implementations efficiently exploit the memory hierarchy and can deliver high throughput on their target platform [1, 7, 8]. This article argues that idiom replacement must be both *robust* and *safe* to be an effective solution.

Despite progress in recent research [5, 9], existing approaches for idiom identification are brittle and fail to recognize minor variants of an idiom. Existing solutions require intimate knowledge of polyhedral analysis [9] or of a new domain-specific language that describes idioms [5]. They also have high compilation time costs even when the target idiom is absent from the code (see Section 5.5).

Our solution is a new optimization pass for the Low-Level Virtual Machine framework [10]. This pass combines tree matching and idiom recognition [2, 11–13] with a data dependence analysis to deliver a robust idiom recognition and a safe replacement strategy. Previously, Carvalho et al. presented a work-in-progress report of this compiler pass [14]. This solution extends tooling already present in the Low-Level Virtual Machine framework and is fully integrated in this widely used compiler framework. Therefore it will be easier to adopt, maintain and update. This article also introduces the first, to the best of our knowledge, formulation of an analysis that determines a GEMM matrix's access order by matching the induction variables used in the memory access (see Section 4.1.2). Access-order detection is crucial to enable transparent and correct usage of high-performance libraries.

This article makes the following contributions:

(1) A robust idiom-recognition compiler pass that identifies many variants of the GEMM and SYR2K idioms and replaces them with optimized library calls (see Section 4 and Section 4.3). **Kernel Find & Replacer (KernelFaRer)**[1] identifies both näive implementations and hand-optimized variants of these idioms(see Section 5.3).
(2) The first formulation of an analysis that determines a GEMM matrix's access orders by combining pattern matching and loop information analysis in LLVM **Intermidiate Representation (IR)** (see Section 4.1.2). The same strategy was employed to match SYR2K's access order.
(3) An experimental evaluation that provides evidence that (a) the pass is robust and identifies many more idiom variants than other approaches, (b) the addition to compilation time is significantly smaller, and (c) the increase in performance is consistent across architectures and libraries (see Section 5).

The remainder of the article is organized as follows: Section 2 presents background material; Section 3 describes preceding related works while adding perspective regarding the proposed approach; Section 4 details how LLVM IR pattern matching and optimized library call insertion are combined in KernelFaRer's implementation; Section 5 describes the experimental setup and analyzes the performance improvements achieved by the proposed approach when compared with current solutions and manual library-based programming; finally, Section 6 concludes the work.

[1]Source code available at https://github.com/jaopaulolc/KernelFaRer.

```
for (i = 0; s[i]; i++);
```
(a)

```
(i < j) ? i : j;
```
(b)

Fig. 1. (a) Idiom of finding the length of a string and (b) returning the minimum of two numbers.

2 PATTERN MATCHING IDIOMS

This section presents programming idioms in the context of pattern matchers that target idioms in LLVM IR code and reviews how the GEMM operation is typically programmed and optimized.

2.1 Programming Idioms

Programming idioms are recurrent constructs that express a computation, can be easily recognized (by humans), and are simple to compose [2]. The introduction of the array-oriented APL language in the 1960s [15] with concise statements that exhibit high memory and high runtime complexity motivated research on idiom recognition and selection [16]. Later, recognition of idioms was used in many programming languages [5, 9, 11–13, 17–20] (see Section 3). Existing idiom-based approaches differ mostly on the following:

- the program representation—usually graph-based structures (e.g., data-dependence graphs, expression trees);
- the matching algorithm (e.g., depth-first traversals or solver-based);
- the normalization constraints for the matching mechanism to work (e.g., memory accesses must be affine) or to be more effective (e.g., common subexpressions elimination);
- how idioms are expressed, either through a **domain-specific language (DSL)** or programatically via a Visitor Pattern [21], and replaced (e.g., code generation or employment of high-performance library).

Idioms have concise syntax and convey common understanding of recurrent computations [15]. For example, Figure 1(a) shows an idiom, written in C, for finding the length of a string `s`. This example relies on the `NULL` byte at the end of well-formed strings, a false value in C that terminates the loop. Figure 1(b) shows how a ternary operator concisely expresses the `min` operation without the need for `if/else` control flow. This article identifies larger idioms that express an identifiable and well-known operation: GEMM. These idioms often account for a significant portion of the execution time of an application and their performance can be greatly improved by replacing them with calls to fine-tuned libraries.

2.2 Pattern Matching in LLVM IR

The LLVM compiler framework has a `PatternMatch` namespace that provides a mechanism to build simple and efficient matchers for LLVM IR [10]. `PatternMatch` is used for static analysis (e.g., Demanded-Bits and Instruction-Simplify analyses), code generation (e.g., Instruction Selection) and transformations (e.g., Instruction Combining). The code in Figure 2 illustrates how `PatternMatch` can simplify instructions in the LLVM `InstCombine` pass.

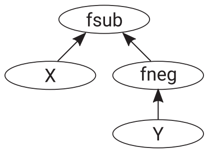
Figure 2(b) shows the tree representation of the expression in the return statement in line 3 of Figure 2(a). Figure 2(c) has a code excerpt from the LLVM `InstCombine` pass that employs `PatternMatch` to simplify the expression $X - (-Y)$ into $X + Y$, shown as a tree in Figure 2(d).[2] In the code of Figure 2(c), `I` references an `fsub` (floating-point subtract) instruction. The method `match(V, P)` returns true if and only if the value V matches the pattern P.

`PatternMatch` provides template methods that both describe IR patterns and bind values in the pattern to pass variables. For example, the `m_FSub` and `m_FNeg` methods shown describe the

[2]The code in LLVM uses a visitor for `fsub` instructions that only matches `fneg`.

```
float foo (float X, float Y)
{
  return X - (-Y);
}
```

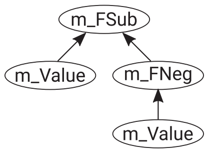
(a)



(b)

```
// X - (-Y) => X + Y
if (match(I,
          m_FSub(
            m_Value(X),
            m_FNeg(
              m_Value(Y)))))
{
  return CreateFAdd(X, Y, &I);
}
```

(c)



(d)

Fig. 2. (a) Double negation example, (b) tree representation of (a), (c) matcher and replacement code, and (d) pattern matched.

instructions `fsub` and `fneg`. The `m_Value` method matches and binds the left-hand side operand of `fsub` to X and the single argument of `fneg` to Y. `PatternMatch` also provides flexible commutative versions of matcher methods (e.g., `m_c_FAdd`[3]) that avoids cumbersome code describing and testing two symmetric patterns. Within a basic block, LLVM can find simple patterns used for instruction selection.

The LLVM `LoopIdiomRecognize` attempts to recognize two idioms—memory-set and memory-copy—to replace the loop code with the `llvm.memset` and `llvm.memcpy` intrinsics. Backends generate inline code or runtime library calls for these intrinsics. The existing LLVM `PatternMatch` cannot be used in the `LoopIdiomRecognize` pass, because it is limited to the scope of a single basic block. Instead, that pass employs only **Data-Flow Analyses (DFAs)** [22] to identify and replace both memory idioms. KernelFaRer uses `PatternMatch` to find loop idioms and DFAs to assess if the the transformations are legal.

2.3 General Matrix-Matrix Multiplication

GEMM, already pervasive in linear algebra computations [23], regained attention, because convolution kernels used in neural networks can be efficiently implemented using GEMM as a primitive [24]. Formally, GEMM can be defined as follows.

Definition 2.1. Let A and B be matrices of dimensions $M \times D$ and $D \times N$, respectively. Let α and β be any value in $\mathbb{R}$. The general matrix-matrix product of A and B is a matrix C of dimensions $M \times D$ such that:

$$C(i,j) = \beta \cdot C(i,j) + \alpha \cdot \sum_{k=1}^{D} A(i,k) \cdot B(k,j). \quad (1)$$

Although a naïve implementation of GEMM is simple, seminal work by Goto et al. shows that a fast GEMM must utilize the memory hierarchy well [1]. Their approach splits the computation into blocks to increase data reuse and creates a matrix memory layout that increases temporal and spatial access locality, a process called *packing the matrices*. The central idea is to focus on the data movement from main memory, through caches, and into the processor's registers.

[3] `_c` indicates that the instruction to be matched (`fadd`) is commutative.

```
1  void cblas_dgemm (
2    const CBLAS_LAYOUT Layout,
3    const CBLAS_TRANSPOSE transa,
4    const CBLAS_TRANSPOSE transb,
5    const int m, const int n, const int d,
6    const double alpha,
7    const double    *a, // A's base address
8    const int      lda, // and leading dimension
9    const double    *b, // B's base address
10   const int      ldb, // and leading dimension
11   const double  beta,
12   double          *c, // C's base address
13   const int     ldc); // and leading dimension
```

Fig. 3. CBLAS interface for double-precision GEMM.

The GotoBLAS library [1] inspired other BLAS libraries. Processor-manufacturer solutions include IBM's **Engineering and Scientific Subroutine Library (ESSL)** [25] and Intel's **Math Kernel Library (MKL)** [26]. Manufacturers also contribute to OpenBLAS [27]. AMD's version is the **BLAS-like Library Instantiation Software (BLIS)** framework [28]. CBLAS is a unified interface to these libraries for C languages [6].

Figure 3 shows the interface for double-precision GEMM. The `Layout` specifies if the resulting matrix C is stored in row-major or column-major order. The arguments `transa` and `transb` indicate if the matrices A and B are transposed, allowing matrices in different storage orders to be multiplied without the creation of copies. An idiom matcher (see Section 4) must deduce the layout of the matrices being matched to obtain these first three parameters. The matrix dimensions (`m`, `n`, and `d`) appear in line 5, followed by the scalar factors α and β (lines 6 and 11), and the base address pointers (lines 7, 9, and 12). The leading dimensions (lines 8, 10, and 13) are the number of elements in the *first* dimension of each matrix.

BLAS libraries rely heavily on direct use of assembly and thus are not portable across platforms, thus the many versions of BLAS. In contrast, Eigen [29] implements BLAS routines using C++ template meta programming to hierarchically organize the decisions and the tuning for each platform. The templates encode platform-specific information that allow the compiler to choose a strategy that is tuned based on cache sizes, vectorization support and other instruction-set-architecture features (e.g., fused multiply-add support). Eigen also relies on code transformations available in modern compilers such as **GNU's C/C++ Compiler (GCC)** and Clang [30]. There is no routine to call to compute GEMM in Eigen, because it is a header-file only library that defines builtin vector and matrix types. Computations are overloaded operations on these types. Thus, the programmer needs to include the header files providing Eigen's types and operators and write C++ expressions of the form: `C = beta * C + alpha * A * B` as per Definition 2.1. Using Eigen to replace a GEMM idiom required the creation of a wrapper library encapsulating Eigen code (see Section 4.3).

3 A BRIEF HISTORY OF IDIOM MATCHING

David Callahan's seminal work identifies bounded-recurrence idioms in data-dependency graphs and replaces them with hand-written parallelized implementations in FORTRAN [11]. Another seminal work by Pinter et al. formalizes idiomatic recognition in a Computational Graph that is used to construct a tree where pattern matching is implemented [12]. They were among the first to normalize loops to improve idiom matching. Their normalization includes loop unrolling to guarantee that all loop-carried dependencies have a maximum distance of one. The goal is to improve idiom recognition by canonicalizing the code, thus eliminating syntactic variations that lead to subtle changes in the intermediate representation. To the best of our knowledge, there is no experimental evaluation of Pinter et al.'s work. KernelFaRer differs from Callahan's approach, because it is integrated in the compilation, and differs from Pinter et al.'s approach, because it does not build auxiliary structures for pattern matching, instead it relies only on the LLVM IR [10].

Menon et al. propose **Abstract Matrix Form (AMF)**, an algebraic language, as an IR and optimization language [13]. AMF expressions are transformed through a set of axioms used as rewrite rules provided by the user. AMF targets loop-based languages, such as FORTRAN and MATLAB, via source-to-source translation. The evaluation of AMF is restricted to simple matrix-matrix and matrix-vector kernels on a single platform. Birkbeck et al. also define a canonical representation for MATLAB programs and find idioms to match patterns in an extensible database [31]. In contrast, KernelFaRer applies to any language once it is compiled to LLVM IR and is thoroughly evaluated across three platforms (see Section 5).

Sato Hiroyuki use grammar rewriting rules based on the array semantics of FORTRAN to propose a rule-based term rewritter for idiom recognition and replacement [17]. Later, Hiroyuki supports graph rewriting optimizations [18]. This approach is limited to FORTRAN, vulnerable to syntactic variations in the source code, and there is also no public evaluation of the approach.

He et al.'s tree-based idiom recognizer uses a **Reduced Affinity Relation Graph (RAG)**, a directed tree structure rooted at the left-hand side value of an assignment expression [19]. A RAG encodes data-dependency relations between values in an expression. Temporaries do not appear in a RAG. Likewise, KernelFaRer eliminates unnecessary temporaries through standard compiler transformation passes. He et al.'s approach incurs a memory and time overhead for creating and maintaining the RAG. In contrast KernelFaRer uses existing data representations already built during compilation.

Kawahito et al. match idioms to perform JIT compilation of JAVA string operations and replace them with IBM's System Z machine instructions [20]. This pattern matcher uses the **Abstract Syntax Tree (AST)** and the **Control-Flow Graph (CFG)**. Topological Embedding [32] enables them to implement non-exact idiom matching in the AST or CFG. Non-exact matches are made exact via a custom transformation pipeline. However, the transformation passes needed for non-exact matching limit the general applicability of this solution.

KernelFaRer is not the first work to recognize that customized implementations of idiomatic operations can match the performance of specialized libraries. Spampinato et al. describe SLinGEN, a DSL for expressing small-scale linear algebra applications [33]. SLinGEN optimizes an algebraic description of a kernel in its high-level representation and generates a C program with architecture-specific vector intrinsics. In contrast with KernelFaRer, SLinGEN was designed to target only small-scale and statically sized kernels. KernelFaRer is not limited to small, static kernels and automatically identifies the idioms in the code written in general purpose languages, no DSL specification required. Rink et al. present CFDlang, yet another DSL that targets tensor operations in the fluid dynamics application domain [34]. Similarly to SLinGEN, CFDlang generates C programs that implement the idiom specified in its DSL and performs high-level optimizations as source-to-source transformations. KernelFaRer is not restricted to the fluid dynamics domain. In fact, KernelFaRer can readily identify GEMM and SYR2K in native programs without custom DSL specifications. The fact that KernelFaRer is integrated into LLVM, a major compiler framework, also sets it apart from both SLinGEN and CFDlang that are external tools.

The two most recent, closely related approaches to idiom matching are by Ginsbach et al. [5] and by Chelini et al. [9]. Ginsbach et al. introduce the **Idiom Description Language (IDL)**, a constraint-based DSL to describe idioms. IDL constraints are synthesized into LLVM IR passes that perform the matching process. While the first IDL prototype [5] had no automatic idiom-replacement mechanism, Ginsbach et al. later proposed a new DSL called LiLAC that includes automatic idiom replacement for both dense and sparse matrix-vector operations [35]. However, LiLAC's passes cannot identify the matrix layouts and the evaluation uses hard-coded calls with an assumed access order. Adoption, maintenance, and extension of IDL is challenging, because it is external to the compilation framework. This prevents IDL users from exploiting rich debugging tools in LLVM and makes the task of specifying idioms much more involved, since IDL is composed of several small components written in different languages. Similar limitations are found in Chelini et al.'s Loop Tactics [36], a DSL to describe idioms to be matched in a polyhedral schedule tree—an auxiliary structure constructed from LLVM IR, and to be replaced by either a library call or by code generated by Polly [3]. Adopting either of the DSL approaches is nontrivial. Loop Tactics requires understanding the polyhedral model and it is difficult to understand a complex composition of constraints in IDL. In contrast, KernelFaRer requires no auxiliary data structures and is fully based on standard LLVM IR passes, allowing developers experienced with LLVM's IR to extend its methodology to other idioms.

4 AN IDIOM RECOGNITION AND REPLACEMENT PASS

This section presents KernelFaRer, an LLVM IR pass that performs idiomatic code rewrite and is integrated into LLVM's standard optimization pipeline. KernelFaRer is an independent IR pass that extends the LLVM `PatternMatch` **Application Programming Interface (API)** (see Section 2.2) with custom matchers to identify more complex idioms. KernelFaRer works independently of other LLVM optimization frameworks like Polly, but it can be of assistance to it or other paths to code generation. The description of this pattern-matching extension is demonstrated in a GEMM case study. However, the methodology is flexible enough to capture all IR constructs and thus enables the description of many idioms. In fact, performance results with our prototype implementation of a SYR2K matcher and replacer are discussed in Section 5. A standard dataflow analysis is sufficient to determine if values computed in the idiom are used elsewhere in the program and thus can also be extended to other idioms.

The KernelFaRer's algorithm can be divided into three phases as follows:

(1) Identify candidates that match the target idiom through IR matchers (see Section 4.1).
(2) Check data dependencies and isolate the matched code (see Section 4.2).
(3) Replace the idiom with a call to a high-performance library (see Section 4.3).

Phase 1 uses LLVM's `PatternMatch` to identify IR code that matches the target idiom. The data-dependence analysis in Phase 2 determines if the replacement of the matched code with a library call is legal. This phase also determines if code transformations, such as loop distribution or loop invariant code motion, are needed to make the transformation legal.

4.1 GEMM Pattern Matching (Phase 1)

GEMM memory access patterns can be expressed in a higher-level programming language as shown in Figure 4. The variables iv_i, iv_j, and iv_k on line 1 of Figure 4(a) are induction variables of a level-three loop nest (or deeper). The `for` loop syntax in Figure 4(a) indicates that the three loops can be nested in any order. For any nesting order, the reduction on line 2 expresses a GEMM. Parenthesis operators are used for indexing the arrays to indicate that the elements of the array may be accessed either in column-major or in row-major order. For instance, $A(i, k)$ means that the element on the ith row and kth column is accessed. The implementation of the operators for column-major order is shown in Figure 4(b) while the row-major access is shown in Figure 4(c), where `addr` is the base address of an array and `ld` is the leading dimension of the array. The dimensions of the matrices are $A_{M\times D}$, $B_{D\times N}$, and $C_{M\times N}$.

Identifying a GEMM idiom requires the identification of its two components: the loop nest on line 1 and the reduction operation on line 2 of Figure 4(a). The multiply-and-add operations of the

```
for (0 <= iv_i < M; 0 <= iv_j < N; 0 <= iv_k < D)
  C(iv_i,iv_j) += A(iv_i,iv_k) * B(iv_k,iv_j)
```

(a)

```
operator()(iv_1, iv_2) {
  return *(addr + iv_2 * ld + iv_1);
}
```

(b)

```
operator()(iv_1, iv_2) {
  return *(addr + iv_1 * ld + iv_2);
}
```

(c)

```
 1 %41 = phi [%53, %40], [0, %25]
 2 %42 = phi [%52, %40], [0, %25]
 3 %43 = mul %41, %17
 4 %44 = add %43, %23
 5 %45 = getelementptr %4, %44
 6 %46 = load %45
 7 %47 = mul %41, %16
 8 %48 = add %47, %26
 9 %49 = getelementptr %6, %48
10 %50 = load %49
11 %51 = fmul %46, %50
12 %52 = fadd %42, %51
13 %53 = add %41, 1
14 %55 = icmp eq %53, %21
15 br %55, %30, %40
```

(d)

Fig. 4. (a) Memory access of GEMM in source code; (b) column-major access order; (c) row-major access order; (d) simplified LLVM IR code of the innermost loop in (a) (Code in (b) and (c) is in C/C++).

reduction idiom appear on lines 11 and 12 of the LLVM IR of the innermost loop nest shown in Figure 4(d). In this **Static Single Assignment (SSA)** representation the result of each instruction is assigned a unique value. Thus, each of the instructions in Figure 4(d) can be referred to by its value V. For instance, the fmul in line 11 is uniquely identified by the value %51 and the fadd in line 12 by the value %52. In general, when presenting an algorithm that iterates over all the instructions in the body of a loop L, this article will say "for all values V in L."

A *GEMMReduction* is a multiply-add instruction sequence that satisfies the following conditions: (1) It appears in the innermost level of a loop nest of at least depth three; (2) the operands are memory accesses to arrays; (3) the address of the memory accesses are affine expressions of the form addr+iv_x×ld+iv_y, where iv_x and iv_y are index variables in the loop nest and addr is a loop-invariant expression—either the base address of a matrix or the base address or a block within the matrix; and (4) the combination of induction variables used in the address expressions is one of the combinations shown in Table 1.

The first component of the pattern, loop nests, are identified using LLVM's LoopInfo analysis pass. LoopInfo provides a consistent way to retrieve loop information from the IR of a program, such as the nesting level of a given loop. The second part of the pattern, *GEMMReduction*, cannot be identified with the existing LLVM PatternMatch API. Therefore, KernelFaRer extends PatternMatch by adding new matchers and new constructs for matching more complex patterns. Figure 5 shows the main patterns contributed by KernelFaRer, those printed in red are the proposed extensions and those in black are the existing constructs. The main pattern that matches a store of a *GEMMReduction* to the destination matrix C is shown in Figure 5(a), a similar pattern was created to match SYR2K reductions.

KernelFaRer introduces the OneOf combiner construct to allow a list of multiple matchers, which usually represent subtle variations in a target pattern, to be tested in turn. OneOf only returns a successful match if one of the provided sub-matchers match the underlying piece of LLVM IR. This construct is widely used in KernelFaRer extensions to facilitate the description of variations in the target pattern. PatternMatch provides basic disjunctive and conjunctive nodes. Disjunction nodes allows the capturing of idioms with polymorphic components. Conjunction nodes allow further specification and pattern component constraints. Combining OneOf with these nodes makes KernelFaRer pattern matcher more robust. For example, *ScaledVOrV*(s, V) uses the

```
template <typename MatcherType>
auto MatchStoreOfMatrixC(MatcherType GEMMReduction, Value C, Value Alpha, Value
    Beta, PHINode iv_1, PHINode iv_2, Value LDC, Value GEP) {
  return m_Store(
    OneOf(ScaledVOrV(Alpha, GEMMReduction),
          ScaledVOrV(Beta, m_PHI(m_Value(), GEMMReduction)),
          m_c_FAdd(ScaledVOrV(Beta, m_Load(GEP)),
                   ScaledVOrV(Alpha, m_PHI(m_Value(), GEMMReduction)))),
    ArrayAccess(C, iv_1, iv_2, LDC));
}
```

(a)

```
inline auto GEMMReduction(Value AddLHS, Value MulLHS, Value MulRHS, PHINode iv^A_1,
    PHINode iv^A_2, PHINode iv^B_1, PHINode iv^B_2, Value Alpha, Value LDA, Value LDB) {
  return MultiplyAdd(
      Alpha, AddLHS, m_Load(ArrayAccess(MulLHS, iv^A_1, iv^A_2, LDA)),
      m_Load(ArrayAccess(MulRHS, iv^B_1, iv^B_2, LDB));
}
```

(b)

```
1 auto ArrayAccess(Value Op, PHINode iv_1, PHINode iv_2, Value LD) {
2   return OneOf(
3       m_GEP(m_Load(m_GEP(Op, m_PHI(iv_2))), m_PHI(iv_1)),
4       m_GEP(m_GEP(Op, PHITimesLD(iv_1, LD)), m_PHI(iv_2)),
5       m_GEP(m_GEP(Op, m_PHI(iv_2)), PHITimesLD(iv_1, LD)),
6       m_GEP(Op, m_PHI(iv_1), m_PHI(iv_2)),
7       m_GEP(Op, AffineFunctionOfPHI(iv_1, iv_2, LD));
8 }
```

(c)

Fig. 5. (a) Matcher of a store of *GEMMReduction* into matrix C, (b) *GEMMReduction* matcher, and (c) Matcher for an array access.

disjunctive node to match either $s * V$ or V, where V is any value represented by a KernelFaRer & PatternMatch matcher.

Figure 5(b) shows the matcher that captures the *GEMMReduction* itself, where MultiplyAdd matches expressions of the form $\alpha * A * B$.[4] ArrayAccess, shown in Figure 5(c), is the matcher that identifies accesses to arrays in different representations. Lines 3–6 are the sub-matchers representing different variants of access to two-dimensional (2D) arrays, while in line 7 the matcher identifies flat-arrays. m_GEP and m_PHI match their respective IR instructions, namely getelementptr and ϕ-nodes. Flat-arrays have indexing expressions that are affine functions of loop induction variables and such expressions are matched with AffineFunctionOfPHI. Two-dimensional-array accesses have an idiomatic expression that multiplies an induction variable by a matrix leading dimension. PHITimesLD captures two variants of this idiom, thus matching wether the multiplication is performed by a multiply (mul) or by a shift-left (shl) LLVM IR instruction.

4.1.1 Algorithm to Match a GEMM. The central idea of this article is that the loop information provided by the compiler can be used to constrain the search for the target idiom's components to the places where it is possible for them to occur. In LLVM, the nesting level, entry, exit, and latch basic blocks for each loop are available through the LoopInfo pass. However, there are two limitations: (1) In a kernel that has been optimized through blocking, the inner loops are not in canonical form, and (2) the induction-variable information is only available for canonical loops

[4] α is optionally matched by using PatternMatch's disjunctive node.

ALGORITHM 1: LLVM IR Pass to find GEMM candidates.

```
1:  function findGemmIRPass(Function F, LoopInfo LI)
2:    LoopList ← findInnerDeepLoops(F, LI)
3:    for all Loops L in LoopList do
4:      for all Values V in L do
5:        if GemmPattern.Match(V) then
6:          IVList ← {$iv_1^A$, $iv_2^A$, $iv_1^B$, $iv_2^B$, $iv_1^C$, $iv_2^C$}
7:          (OrderList, $iv_i$, $iv_j$, $iv_k$) ← MatrixAccessOrder(IVList)
8:          (M, N, D) ← LoopsUpperBound(LI, $iv_i$, $iv_j$, $iv_k$)
9:          if AllGemmValuesFound() then
10:           A ← Matrix(OrderList, M, N, D)
11:           B ← Matrix(OrderList, M, N, D)
12:           C ← Matrix(OrderList, M, N, D)
13:           Gemm ← (L, IVList, A, B, C)
14:           Candidates.insert(Gemm)
15: function match(Value V, Pattern P)
16:   return P.match(V)
17: interface Pattern<T>::match(Value V)
18: function Pattern<FAdd>::match(Value V)
19:   AddOper0 ← V.getOperand(0)
20:   AddOper1 ← V.getOperand(1)
21:   return FAdd_Match(AddOper0, AddOper1)
```

via the `LoopInfo` pass [10]. KernelFaRer's solution is to combine basic-block and loop-nesting information from LoopInfo with `PatternMatch`. In LLVM IR, induction variables are lowered to ϕ instructions with at least two incoming values: an initialization value; and a new value that comes from the loop's latch basic block. KernelFaRer's general idiom recognizer uses `PatternMatch` to match these ϕ instructions.

Algorithm 1 uses `PatternMatch` and the `LoopInfo` pass to identify GEMM candidates. First, using the LLVM `LoopInfo` pass, find all innermost loops that are nested at the third (or deeper) level and place these loops in a list (line 2). Then, iterate over all instructions (value V in line 4) inside those loops, invoking the method Match of a `GemmPattern` object on each one of them. The generic interface for Match is shown in line 17, where `T` can be any LLVM IR instruction type. Each instruction type in a pattern must have an implementation for the Match interface. For instance, the implementation of Match for `FAdd` for the GEMM pattern is shown on line 18.[5] To find the idiom, Match descends the IR tree matching the specified pattern.

Values from the IR are captured through matcher objects of type BindType. `GemmPattern` contains such objects to capture, for instance, values associated with a GEMM's induction variables and memory address instructions.

4.1.2 *Determination of the Matrix Access Order.*

A crucial step to replace a GEMM idiom with a call to a library is to determine the access order used by the idiom for each matrix (Algorithm 1, line 7). These access orders are required by the library call that will replace the detected idiom (see Section 4.3). This analysis differs from the work of Wolfe et al. where the iteration domain is analyzed to identify dependencies in a loop [37]. Here the goal is to identify the access order of each matrix in the target pattern.

[5]The FAdd_Match in line 21 is an algorithmic simplification of an object-oriented code structure in LLVM. It abstracts the matching of both operands of a `fadd` instruction referenced by `V` (line 18) using the `FAdd_Match` object within `GemmPattern`.

Table 1. Access Offset Expressions for All Combinations of Column-major (CM) and Row-major (RM) Order

Access Order			Offset Expressions		
C	A	B	C	A	B
RM	RM	RM	$iv_i \times ld_C + iv_j$	$iv_i \times ld_A + iv_k$	$iv_k \times ld_B + iv_j$
CM			$iv_j \times ld_C + iv_i$		
RM	CM		$iv_i \times ld_C + iv_j$	$iv_k \times ld_A + iv_i$	
CM			$iv_j \times ld_C + iv_i$		
RM	RM	CM	$iv_i \times ld_C + iv_j$	$iv_i \times ld_A + iv_k$	$iv_j \times ld_B + iv_k$
CM			$iv_j \times ld_C + iv_i$		
RM	CM		$iv_i \times ld_C + iv_j$	$iv_k \times ld_A + iv_i$	
CM			$iv_j \times ld_C + iv_i$		

Table 1 shows the offset expressions according to the access order of the matrices assuming that the index variables are iv_i, iv_j, and iv_k. When a multiply-and-add operation that is a candidate to be a GEMM reduction is encountered, the expressions for the access into the matrices are identified as $l_C \times ld_C + o_C$, $l_A \times ld_A + o_A$, and $l_B \times ld_B + o_B$, where l_C is the index variable for the leading dimension of matrix C and o_C is the offset into that dimension of C.[6] The same logic is applied to the matrices A and B. The access order can be deduced by examining which of the index variables are identical in the multiply-and-add idiom candidate.

Table 2 shows the equality conditions that determine the access order for matrices A and B. For instance, if $o_A = l_B$, then both matrices A and B are accessed in row-major order as shown in the first two rows of Table 1. Given that all the possible combinations for access orders for a GEMM are given in Table 1, it follows that if none of the equality constraints shown on the second column of Table 2 are met, then the multiply-and-add cannot be a GEMM reduction.

To illustrate how to determine the access order of C, let us examine the two top rows of Table 1 where both matrices A and B are accessed in row-major order. There are two cases. If $l_C = l_A$ and $o_C = o_B$, then C is accessed in row-major order. If $l_C = o_B$ and $o_C = l_A$, then C is accessed in column-major order. If neither of these conjunctions are satisfied, then the multiply-and-add is not a GEMM reduction. Similar sets of conjunctions can be written for the other three combinations of access order for A and B as shown in Table 2.

This strategy can also be used to detect the access order of matrices in other computational kernels (e.g., SYR2K).

4.1.3 Loop Upper Bounds. The intuition to determine loop upper bounds is that the index variable of a loop must be initialized prior to starting the execution of the loop and must be updated within the loop body. Therefore, in the SSA representation there must be a ϕ instruction that merges the initialization path with the update path at the first basic block of a loop. First, add all ϕ instructions that are in the first basic-block of the loops and that use iv_i, iv_j, or iv_k to a work list. Then, match each ϕ instruction in this list against a pattern tree that represents the instruction sequence for the loop index update and loop comparison.[7] The upper bound of a loop L is the operand compared with the value defined by the ϕ instruction if the pattern belongs to the latch block of L. All GEMM instances matched by `GemmPattern` that have a valid access order and for

[6] l_X and o_X are also known as the angular and linear coefficients of a linear function, such as those in indexing expressions used to access array elements.
[7] This instruction sequence is idiomatic of loop-exit conditions.

Table 2. Conditions to Determine the Access Orientation

Access Order		Equalities to derive access order of		
A	B	A and B	C	
			RM	CM
RM	RM	$o_A = l_B$	$l_C = l_A \wedge o_C = o_B$	$l_C = o_B \wedge o_C = l_A$
CM	RM	$l_A = l_B$	$l_C = o_A \wedge o_C = o_B$	$l_C = o_B \wedge o_C = o_A$
RM	CM	$o_A = o_B$	$l_C = l_A \wedge o_C = l_B$	$l_C = l_B \wedge o_C = l_A$
CM	CM	$l_A = o_B$	$l_C = o_A \wedge o_C = l_B$	$l_C = l_B \wedge o_C = o_A$

ALGORITHM 2: Data-Dependence Analysis IR Pass.

```
1:  function AnalysisPass(Function F, Kernel K)
2:    L ⟵ K.getAssociatedLoop()
3:    for all Instructions I in L do
4:      if I ∈ K.Values or I ∈ K.Stores then continue
5:      if I.mayWriteToMemory() then
6:        return False
7:      if I.mayHaveSideEffects() then
8:        return False
9:      for all Users U of Instruction I do
10:       B ⟵ BasicBlock of I
11:       if B ∉ L then
12:         return False
13:     return True
```

which loop nests upper bounds can be determined are candidates for replacement with a call to a library (see Section 4.3). Now, a dataflow analysis must establish the legality of the transformation.

In the triangular iteration space of SYR2K two out of tree induction variables are traversed in the same way as in GEMM and the third variable's (J) upper (lower) bound is dependent on another variable (I). To detect this pattern, KernelFaRer checks for the presence of I in the ϕ instruction for J. This detection method can be thwarted by the insertion of auxiliary induction variables.

4.2 Data-Dependence Analysis

Idiom replacement must preserve program behavior, including writes to memory and outputs. Also, any inner-loop code that is not part of the GEMM reduction must be safely moved out of the loop nest. A liveness and side-effects analysis of the use-def chains in the idiom's loop nest can be combined to determine if any intermediate value is live after the idiom or produces side-effects during the computation. This analysis differs from the loop dependence analysis of Wolfe et al. [37].

The Data-Dependence Analysis in Algorithm 2 receives the IR for a function F and a Kernel object found within F. It returns a *Boolean* indicating if the idiom replacement is legal. At the matching stage, the algorithm gathers two data structures. ($K.Values$) contains matrix pointers, offsets and intermediate values; ($K.Stores$) contains the stores in the Kernel. These data structures afford flexible access to the Kernel parameters and enable checking that the only extra stores in the loop nest are initializations stores.

The Data-Dependence Analysis algorithm iterates over all instructions in the kernel's associated loop (line 3) skipping those that are in the kernel (line 4). If an instruction may store to memory (line 5) or produces side-effects such as throwing an exception (line 7), then return *False*. The

```
1 for (long i = 0; i < M; i++)
2   for (long j = 0; j < N; j++) {
3     C[i][j] = 0.0;
4     for (long k = 0; k < D; k++)
5       C[i][j] +=
6         alpha * A[i][k] * B[k][j];
7   }
```

(a)

```
1 for (long i = 0; i < M; i++)
2   for (long j = 0; j < M; j++)
3     for (long k = 0; k < N; k++) {
4       C[i][j] +=
5         alpha * A[i][k] * B[j][k];
6       C[i][j] +=
7         alpha * B[i][k] * A[j][k];
8     }
```

(b)

```
1 for (long r = 0; r < R; r++)
2   for (long q = 0; q < Q; q++) {
3     for (long p = 0; p < P; p++) {
4       sum[r][q][p] = 0.0;
5       for (long s = 0; s < P; k++)
6         sum[r][q][p] +=
7           A[r][q][s] * C4[s][p];
8     }
9     for (long p = 0; p < P; p++)
10      A[r][q][p] = sum[r][q][p];
11  }
```

(c)

Fig. 6. (a) Naïve GEMM, (b) symmetric rank-2k operations (syr2k), and (c) Multiresolution analysis kernel (doitgen).

algorithm also checks for data flow from definitions in the kernel innermost loop to uses after the loop (lines 9–12).

The kernels in Figure 6 illustrate when instructions may or may not be moved out of a GEMM loop. Two of them, Figure 6(b) and (c), are from benchmarks in PolyBench [38]. In spite of its simplicity, the naïve GEMM implementation in Figure 6(a) cannot be directly rewritten because of the initialization of matrix C. This initialization must first be moved to a separate loop, because all high-performance libraries assume that matrix C is initialized prior to calling the function that implements GEMM. Another possibility is to simply delete the GEMM reduction store that is in line 5 and to insert a library call at the exit-block of the loop nest—dead code can be removed later by LLVM passes. Side-effects in the initialization of the matrix C could also prevent rewrite. KernelFaRer recognizes this idiom and replaces it if no side-effects exist.

Figure 6(b) shows a symmetric rank-2k operation from the SYR2K kernel. This kernel accesses matrix A in row-major order and B in column-major order in line 4 and then matrix B in row-major order and A in column-major order in line 5. C is accessed in row-major order. The replacement of the idioms in SYR2K is safe and the access order is detected by the algorithm. No copy operations are required, because a matrix stored in row-major order can be accessed in column-major order by specifying to the library that the matrix is transposed. However, floating-point arithmetic is not associative and, depending on the values in A and B, the result after and before the transformation might differ for a given error tolerance. In this case there are three options: (1) do not replace it, (2) replace it with two calls to `cblas_gemm` if the the user allows it, or (3) replace it with a call to `cblas_syr2k`. KernelFaRer replaces this idiom with a call to `cblas_syr2k`.

The multiresolution analysis `doitgen` kernel shown in Figure 6(c) has `R` GEMM instances, one for each resolution `r`. The GEMM in each resolution can only be rewritten after the update of row `q` in A with the values from `sum` (lines 8 and 9) is moved to a new loop-Q after the original one (line 2). Loop distribution is not currently available in KernelFaRer, because LLVM only distributes innermost loops.

4.3 Idiom Rewrite

The prototype of KernelFaRer evaluates two replacement options: a custom library that leverages Eigen [29] or the CBLAS [6] interface, which is widely used by OpenBLAS [7] and by

Table 3. Machine Configuration Used in the Evaluation

Component/Feature	Intel	IBM	AMD
Processor	2x Xeon 8268	2x Power9	1x EPYC 7742
Cores/Threads	24/2	20/4	64/2
L1 instruction cache	32 KiB	32 KiB	32 KiB
L1 data cache	32 KiB	32 KiB	32 KiB
L2 (unified)	256 KiB	512 KiB	512 KiB
L3 (unified+shared)	35.75 MiB	10 MiB	16 MiB
RAM	755 GiB	1 TiB	995 GiB
Memory bandwidth	131.13 GiB/s	140 GiB/s	190.7 GiB/s
FMA Latency	4	7	5
FMA Throughput	2	2	2

vendor-specific libraries such as MKL [26], ESSL [25] and BLIS[8] [28]. This interface allows the evaluation of multiple libraries without recompiling the program. CBLAS expects that matrices be stored contiguously in memory, thus KernelFaRer does not replace any GEMM that access the matrices through double pointer indirection, e.g., `float**`, because the contiguity constraint cannot be proven. In addition, instaces where only sub-blocks of 2D arrays are accessed to compute GEMM are also not replaced, since the contiguity constrain is violated.

Unlike BLAS, Eigen [29] uses a minimum amount of assembly code; is portable; and supports all basic types from C/C++ languages, not just single and double-precision floating-point types. KernelFaRer creates a slim wrapper routine for each data type in Eigen. Eigen heavily exploits C++ template meta programming and other object-oriented features (e.g., polymorphism). In Eigen's template type hierarchy the compiler chooses which methods to instantiate based on the target platform (see Section 2.3).

The following steps replace a GEMM-idiom loop nest with a library call: (1) Insert a call instruction in the exit-block of the idiom's loop nest (the values captured in the matching phase (Section 4.1) are passed as arguments to the library call); (2) delete the idiom's store to the resulting matrix. This makes the computations in the idiom's loop nest dead-code, which can be removed by LLVM passes scheduled afterwards; (3) run LLVM's loop deletion, dead-code elimination, and CFG simplification passes.

5 EXPERIMENTAL EVALUATION

This experimental evaluation assesses (1) the effect on performance of replacing native code idioms with high-performance library calls on three platforms: Intel x86, AMD x86, and IBM PowerPC (Section 5.2); (2) the robustness of KernelFaRer's pattern matching in comparison with Polly and IDL [5] (Section 5.3); and (3) the effects of pattern matching and analysis on compilation times (Section 5.5).

5.1 Experimental Setup

This section presents the detailed experimental setup, methodology, and all tools used in the experimental evaluation presented next.

5.1.1 Machine Setup. This experimental evaluation uses the three platforms shown in Table 3 to determine if the performance trends are platform specific.[9, 10] All platforms run Linux with a 64-bit

[8]BLIS is actually an open-source project but AMD provides support and advertises it as their platform-specific solution: https://developer.amd.com/amd-aocl/blas-library/.
[9]Core counts are per socket and thread counts are per core.
[10]Cache sizes are per core or per core pair.

kernel at version `4.15.0-115-generic`. The processor's frequency was locked to 3.5 GHz on the Intel machine and to 2.3 GHz on the IBM machine. Issues with the `acpi-cpufreq` module prevent the locking of the frequency on the AMD machine and thus the frequency fluctuates between 1.8 and 2.2 GHz.

5.1.2 Compilation Pipeline. All benchmarks are compiled using Clang and LLVM tools from the proprietary LLVM 12.x branch at `-O3` with `-mtune=native` and the options `-march=native` for x86 (Intel and AMD) and `-mcpu=native` for IBM PowerPC. IDL [5] and KernelFaRer need to run prior to loop unrolling and vectorization passes. The solution is to disable these passes via `-fno-unroll-loops -fno-vectorize -fno-slp-vectorize` before the idiom-identification pass and to reenable them afterwards by running the `opt` LLVM tool using `-O3`. All binaries, including the baselines, were compiled with `-ffp-contract=fast` and `-ffast-math` for a fair comparison with libraries that assume a relaxed floating-point contract.

5.1.3 Backends. This evaluation uses the following backends: BLAS (via the OpenBLAS implementation [7, 27]), Eigen [29], MKL [26], BLIS [28], and ESSL [25]. It uses the latest stable release, from source, of BLAS[11] and Eigen.[12] It uses the most recent, pre-built, binaries available from the vendors for the platform-specific backends—MKL,[13] BLIS,[14] and ESSL.[15]

Libraries are unmodified except for setting flags to enable multithreading. The number of threads in each platform is chosen such that all computation is performed on a single socket and no hyperthreading takes place. This criteria results in 24 threads for the Intel platform, 64 for the AMD, and 20 for the PowerPC (see Table 3). The same number of threads is used in each platform for all backends.

5.1.4 Polly. This evaluation uses the version of Polly available in the proprietary LLVM 12.x branch. The target-specific instruction scheduling in Polly's GEMM optimization pass requires the following platform information: size and associativity of first and second level caches, latency, and throughput of vector **fused multiply-add instruction (FMA)**. This information was extracted from each vendor's software optimization guides [39, 40] and Agner Fog's instruction tables [41] and included the architecture-specific information in Table 3. For the evaluation, Polly's code uses the same thread counts as the other approaches. Even though pattern-matching options are always enabled in Polly, it is only successful in four of seven programs (see Section 5.3).

5.1.5 IDL. IDL [5] does not provide a replacement pass (see Section 4.3). This evaluation focuses on the idiom identification mechanism of IDL. The performance improvement of replacing idioms identified by IDL should be the same as when using KernelFaRer. Normally, IDL also has the ability to detect multiple idioms at once, but in this evaluation, all other idioms have been disabled, leaving only GEMM.

5.1.6 PolyBench. This evaluation uses all of the benchmarks in the PolyBench/C 4.2 benchmark suite [38], a set of benchmarks originally curated for the testing of Polly. Choosing the extra-large datasets ensures that the input and output matrices do not fit in cache, thus allowing assessment of the cache and memory-hierarchy awareness of each evaluated library and strategy. This study evaluated all the benchmarks in the suite but presents results only for benchmarks where at least

[11]Default branch at 1f62a8278983f7afec8c9c28ecbb2f4892f7ce52.
[12]Default branch at b5df8cabd7b9dcaf3eb0ab93416f3f25352c55f2.
[13]Version 2020.0 build 20191122.
[14]Version 1.3 build 20190901.
[15]Version 6.2.1.

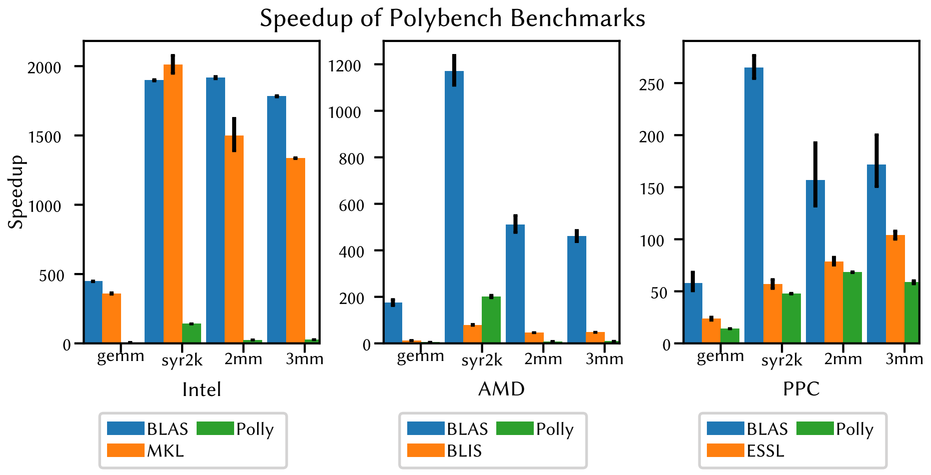


Fig. 7. The speedup of benchmarks when compared to the same benchmark run at `-O3` on the respective platform.

one of the approaches recognized a GEMM pattern. Section 5.2 discusses the impact on the other benchmarks.

5.1.7 Experimental Methodology. The methodology proceeds as follows. (1) Compile each benchmark, for each platform, with aggressive optimization flags (see Section 5.1.2) using five strategies: baseline, Polly, and replacement with BLAS, Eigen, and a platform-specific library (MKL for Intel, BLIS for AMD, and ESSL for PowerPC). (2) Create a list containing all the executables. (3) Measure the execution time of each element of the list once. (4) Randomize the list of executables before the next set of measurements. (5) Repeat until there are twenty measurements for each executable. This methodology ensures that changes to the execution environment that may affect performance manifest in a higher variance between executions of the same executable rather than introduce bias in the results of the experimental evaluation. All speedups are relative to the platform-specific, baseline application code compiled with `-O3` and are the average of these 20 measurements. The confidence intervals (95%) are computed using Kalibera et al.'s formulation [42].

5.2 Performance Comparison

Figure 7 shows the speedup for each library or strategy for each of the platforms for the four PolyBench/C 4.2 benchmarks where the idioms are recognized and replaced. The outstanding performance of the BLAS libraries is due to a macro/micro-level design strategy. Such strategy enables higher performance gains with SYR2K than with other kernels as only half of the output matrix is accessed.

KernelFaRer never negatively affects performance in the case where no GEMM is detected, because no transformation is imposed. Polly improves performance of other benchmarks in the PolyBench/C 4.2 suite that do not contain a GEMM idiom, but it occasionally causes a degradation. For instance, the performance for `atax` is only 70% performance of the baseline in the AMD environment.

Replacing a native GEMM idiom with a call to the BLAS library on the Intel platform leads up to a 2,000 times speedup for the `2mm` benchmark. Polly also produces non-trivial performance gains—up to 200× for SYR2K on AMD. Polly only targets the first two levels of cache but the third-level cache is significantly larger (see Table 3) and its effective utilization is key for performance [1]. Polly

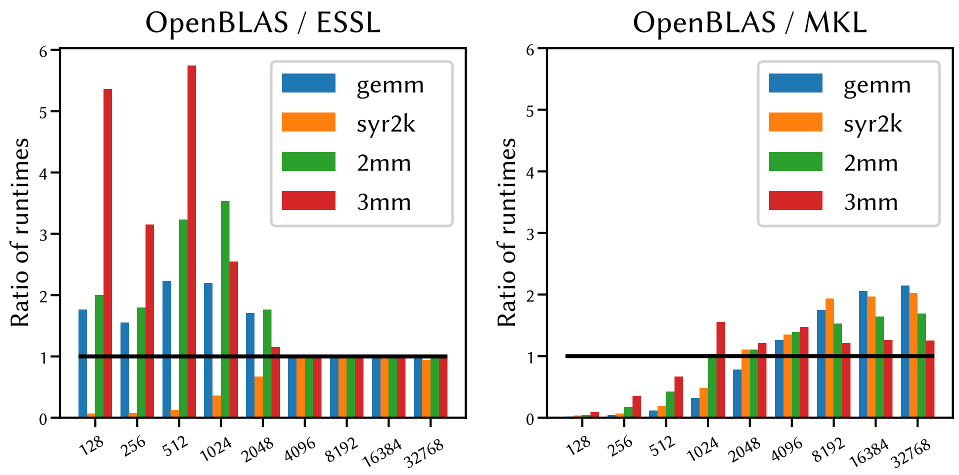


Fig. 8. The ratio of runtimes of OpenBLAS and vendor libraries.

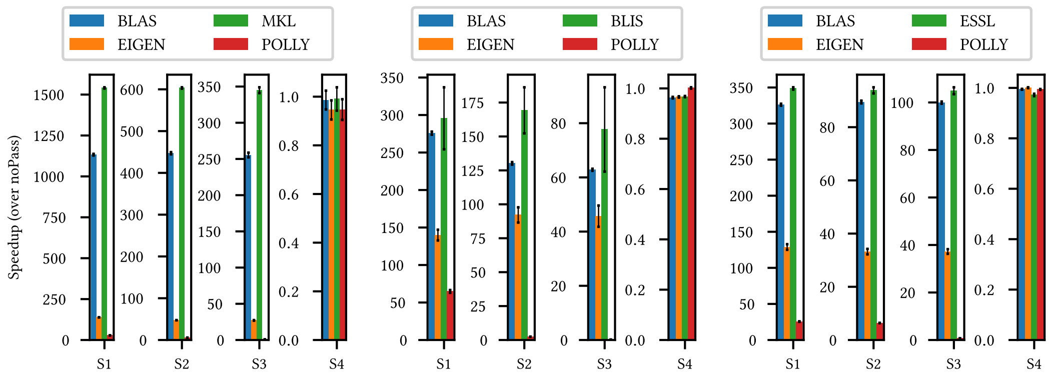
also only applies thread-level parallelism to the outermost loop in a GEMM [9] while the libraries implement a dynamic runtime strategy to decide how to divide the *GEMMReduction* across multiple threads [28]. Polly's current GEMM-packing strategy could be improved by carefully choosing packing parameters for multithreading [43].

In the results shown in Figure 7, OpenBLAS [27] outperforms MKL, BLIS, and ESSL, the vendor-specific solutions from Intel, AMD and IBM, respectively. This result contradicts the expectation that the performance of vendor-specific libraries should be on par with, or superior to, OpenBLAS. This is due to the largest dataset sizes in PolyBench/c 4.2 — $M = 2000$, $N = 2300$, and $K = 2600$—not being in the optimal range for the vendor-specific libraries. Figure 8 shows the ratio of running times between a vendor-provided library (ESSL and MKL) and OpenBLAS. The results were measured for for matrix sizes $M = K = N = 2^i, i \in [7, 15]$ and averaged over 10 runs. ESSL is up to 5× faster for `3mm` and on average 2× faster for GEMM and `2mm` than OpenBLAS for sizes smaller than 4K. The *cblas_syr2k* routine is better optimized in OpenBLAS than in ESSL, which explains the significantly poorer performance of ESSL. PPC's baseline binares (compiled with only `-O3`) also ran three times faster than Intel's baseline, which could explain its lower library speedups. MKL has been optimised for larger matrix operations, visible in its consistent improvment as dimension sizes increase. This is undoubtedly due to the insight that saving a percentage of time from an operation that takes several hours to complete is much more impactful than that same percent taken from one that takes a fraction of a second. Starting with the matrix sizes of 4K, all libraries show comparable performance for all kernels.

5.3 Robustness of Pattern Matching

Sophisticated programmers may attempt to improve the performance of a native-code implementation of GEMM by applying source-code level transformations. The BLISLab tutorial [44], made available by the Science of High-Performance Computing Group [45], provides a sequence of increasingly more sophisticated, high-level-code-only implementations of GEMM.[16] `S0` is the simplest naïve form of GEMM where neither α nor β are used, all matrices are accessed in column-major order and can be expressed as $C = C + A * B$. In `S1` the matrix A is transposed. In `S2`, instead of transposing A, the loop nest (i, j, k) is interchanged to (j, k, i). `S3` both interchanges the loop

[16]Pattern matching assembly code is platform specific and out of the scope of this article.



(a) Intel's x86 (b) AMD's x86 (c) IBM's PowerPC

Fig. 9. The speedup relative to each hand optimization by replacing GEMM in each platform.

nest, as in S2, and tiles the loop nest by 128, 64, and 128, respectively. S4 interchanges the loop nest as in S2, tiles as in S3, and performs a gather operation to pack matrices A and B into blocks of tiles. The values used for tiling and packing are chosen to allow effective cache usage.

The more complex source code makes idiom recognition more challenging as shown in Table 4. KernelFaRer idiom recognition is robust in comparison with the two alternatives. Outside of the patterns in Polybench, the test suite used in its development, Polly recognizes only S3. IDL's recognition expects the result of the *GEMMReduction* to be stored into a scalar variable that is then later used to update the destination matrix—in all benchmarks the reduction is performed directly into the destination matrix. IDL is not expressive enough for the writing of an alternative specification to capture variants of GEMM. Capturing such variants would require modifications to the DSL compiler. This limitation highlights the brittleness of IDL's approach: A programmer must specify in a DSL the exact form of the code to be matched. A different specification is need for each subtle variation in the way the pattern is written. Operating in the LLVM IR, KernelFaRer avoids this gap between the pattern description and the code to be matched, because nodes in the pattern tree are explicitly made to match LLVM IR.

Figure 9 shows that even when a programmer spends significant effort to improve the source code of a GEMM kernel, highly specialized library implementations still deliver significantly superior performance. Note the different scales in the vertical axis of each of the graphs; the amount of work required to calculate the GEMM does not change, so the only varying factor is the effectiveness of the hand optimized version. For S4 KernelFaRer identifies a `GEMM reduction` but the additional memory access needed to gather tiles of matrices A and B lead the KernelFaRer analysis to conclude that it is not safe to replace the GEMM idiom with library calls.

Polly improves the performance of the kernels S1 and S2 through loop tiling, unrolling and fusion. In the case of S3, however, after applying GEMM specific optimizations, Polly produces a result matrix C that differs from that computed by Eigen and BLAS for the same input matrices A, B, and C.[17]

In addition to the programs listed in Table 4, we also created 16 variants of the GEMM idiom by varying the presence of the scaling factors α and β, and the choice of first storing the result of the GEMM reduction into a scalar variable or storing it directly into matrix C.[18] KernelFaRer

[17] A communication to the authors resulted in no response up to the time of this writing.
[18] All the code used in this evalution is part of an Artifact submitted together with this manuscript.

Table 4. Comparison of Pattern Matching Tool Robustness to Different Patterns

	gemm	syr2k	2mm	3mm	S1	S2	S3	S4
KernelFaRer	X	X	X	X	X	X	X	M
Polly	X		X	X			X	
IDL								

KernelFaRer is the presented method. Cells marked "X" indicate that the tool recognized and replaced the kernel idiom. "M" indicates instances where the tool only matched the kernel but was not able to replace it.

recognizes all 16 variants of GEMM; IDL only recognizes the four instances that accumulate into a scalar; and Polly does not recognize any of these variant as a GEMM. Polly always fails, because all variants have an extra loop that stores the expression $C = \beta \times C$, or variants of it, generated by the compiler to guarantee correctness for cases where the innermost loop trip count is zero.

5.3.1 No False Positives Occurred. An important question is whether or not KernelFaRer ever detects a GEMM idiom where it does not exist. No such occurrences were detected while compiling the following benchmark suites: Rodinia 3.1 [46], SPEC CPU 2006 [47] and 2017 [48], and Nekbone [49]. None of the approaches found any instances of the GEMM idiom in these suites, though Nekbone already uses the CBLAS interface to perform GEMM operations. Both its design and the compilation of this extensive corpus of code provides confidence that KernelFaRer does not detect a GEMM idiom when none exists.

5.4 Flexibility

The matching code for GEMM is largely reused to match SYR2K pointing to the potential to extend this pattern matching approach to other kernels. The flexibility of this pattern-matching approach is underscored in this demonstration, because SYR2K is different from GEMM in two principal ways: two matrix multiplication operations instead of one and a triangular output matrix iteration space. Matching SYR2K requires additional checks for the induction variables and their bounds and matrix layouts (similar to Table 1). Integrating the SYR2K pattern in KernelFaRer consists simply of adding a match invocation to the *if* statement of the Algorithm 1. When extending the matching to other kernels, developer's time will be spent on filtering out false positive cases (e.g., rectangular iteration space) and matching corner cases, such as the addition of temporary variables by earlier compiler passes.

5.5 Effect on Compilation Times

Pattern matching consumes compilation time whether a program contains the pattern or not. Using LLVM's built-in pass-timing mechanism, this evaluation applies each of the idiom recognition methods to each benchmark and sums the wall-clock times for each of the passes added as part of the recognition method. Some analysis or transformation passes that are already performed for high levels of optimizations also support the recognition passes but these are not included in the summation, because they are performed whether or not the idiom recognition is also performed. Table 5 shows the averages of 20 compilation-time measurements on PPC. Intel and AMD times differ by no more than 20% on average. The presented times include the time it takes to perform the corresponding pass (KernelFaRer, Polly or IDL) plus the time for -O3 for KernelFaRer and Polly (because Polly requires -O3).

KernelFaRer has low impact in the compilation pipeline, scaling slowly as the complexity of the program increases. IDL's constraint solver method is inherently more costly than tree matching

Table 5. Comparison of Time Spent in LLVM Passes Implementing a GEMM and SYR2K Detection Method

	gemm	syr2k	2mm	3mm	S1	S2	S3	S4
KernelFaRer	26.1	31.1	30.6	31.5	99.0	99.0	99.5	102.4
Polly	348.0	225.1	898.2	1400.5	390.0	616.3	2130.3	3268.5
IDL	255.0	—	959.6	2204.7	40.8	40.7	50.0	451.5

Times are in milliseconds.

and must also analyse all function code, unlike KernelFaRer, which knows that the idiom can only occur in loops of at least depth three and so limits its scope. The times from IDL are also solely for detection of the GEMM idiom, because there is no replacement strategy. The higher compilation time of Polly includes more than idiom recognition as it also computes the polyhedral model of the loops, performs extra transformations beyond idiom recognition, and finally code generation back to IR. The additional compilation time in this evaluation are all from singular compilation units; these impacts will be magnified in larger programs or with more files. For instance, both PolyBench's gemm and `S1` are simple GEMM kernels; the only difference between the two is that `S1` transposes *A*. However, a verification loop elsewhere in the file that prints the result matrix in `gemm` significantly increases the compilation time for Polly and IDL.

As Table 5 shows, the time KernelFaRer spends to detect and replace GEMM and SYR2K is about the same. This indicates that KernelFaRer's matching strategy runtime is not significantly affected by the complexity of the kernel. Polly does not have a specific matcher for SYR2K, but it is able to optimize SYR2K's loop nests in 225.1 ms. The compilation time for IDL is not shown, because, at the time of writing, there is no description of the SYR2K idiom in IDL.

6 CONCLUSION

This article presents KernelFaRer, an idiom recognizer coupled with a dataflow analysis that finds and replaces idiomatic instances of a computing kernel with calls to high-performance libraries. KernelFaRer is implemented entirely within LLVM's compiler framework and extends its `PatternMatch` namespace to build a robust and effective tree-matching based pattern recognizer. The idiom recognition in KernelFaRer is robust: For GEMM it recognizes many more idioms than state-of-the-art approaches, e.g., Polly and IDL, including more complex instances of GEMM that were hand-optimized following well-known source-code transformations. No false-positives were produced when KernelFaRer analyzed extensive codebases from SPEC CPU 2016 & 2017, Rodinia, and Nekbone. This article also introduced a novel strategy to identify the access order of matrices in LLVM IR. Access-order information is central to correctly replacing kernels with calls to high-performance BLAS libraries. KernelFaRer is a sophisticated idiomatic recognizer that only relies on standard dataflow analysis and extensions to tree-matching support in LLVM, a production-ready compiler framework.

REFERENCES

[1] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.* 34, 3, Article Article 12 (May 2008), 25 pages. DOI: http://dx.doi.org/10.1145/1356052.1356053

[2] Perlis, Alan J. and Rugaber, Spencer. 1979. Programming with idioms in APL. *SIGAPL APL Quote Quad* 9, 4-P1 (May 1979), 232–235. DOI: http://dx.doi.org/10.1145/390009.804466

[3] Tobias Christian Grosser. 2011. *Enabling Polyhedral Optimizations in LLVM.* Ph.D. Dissertation. Universität Passau.

[4] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation.* Tucson, AZ, USA, 101–113.

[5] Ginsbach, Philip and Remmelg, Toomas and Steuwer, Michel and Bodin, Bruno and Dubach, Christophe and O'Boyle, Michael F. P. 2018. Automatic matching of legacy code to heterogeneous APIs: An idiomatic approach. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. Association for Computing Machinery, New York, NY, 139–153. DOI: http://dx.doi.org/10.1145/3173162.3173182

[6] L. Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R. Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw.* 28, 2 (2002), 135–151.

[7] Z. Xianyi, W. Qian, and Z. Yunquan. 2012. Model-driven level 3 BLAS performance optimization on loongson 3A processor. In *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems*. 684–691. DOI: http://dx.doi.org/10.1109/ICPADS.2012.97

[8] cuBLAS–NVIDIA Developer. Retrieved January 2020 from https://developer.nvidia.com/cublas.

[9] Chelini, Lorenzo and Zinenko, Oleksandr and Grosser, Tobias and Corporaal, Henk. 2019. Declarative loop tactics for domain-specific optimization. *ACM Trans. Archit. Code Optim.* 16, 4, Article 55 (Dec. 2019), 25 pages. DOI: http://dx.doi.org/10.1145/3372266

[10] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO'04)*. IEEE Computer Society, Washington, DC, 75.

[11] Callahan, David. 1992. Recognizing and parallelizing bounded recurrences. In *Languages and Compilers for Parallel Computing*. Springer, Berlin, 169–185.

[12] Pinter, Shlomit S. and Pinter, Ron Y. 1994. Program optimization and parallelization using idioms. *ACM Trans. Program. Lang. Syst.* 16, 3 (May 1994), 305–327. DOI: http://dx.doi.org/10.1145/177492.177494

[13] Menon, Vijay and Pingali, Keshav. 1999. High-level semantic optimization of numerical codes. In *Proceedings of the 13th International Conference on Supercomputing (ICS '99)*. Association for Computing Machinery, New York, NY, 434–443. DOI: http://dx.doi.org/10.1145/305138.305230

[14] João P. L. de Carvalho, Braedy Kuzma, and Guido Araujo. 2020. Acceleration opportunities in linear algebra applications via idiom recognition. In *Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE'20)*. Association for Computing Machinery, New York, NY, 34–35. DOI: http://dx.doi.org/10.1145/3375555.3383586

[15] Kenneth E. Iverson. 1962. A programming language. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*. ACM, 345–351.

[16] Lawrence Snyder. 1982. Recognition and selection of idioms for code optimization. *Acta Inf.* 17, 3 (01 Aug 1982), 327–348. DOI: http://dx.doi.org/10.1007/BF00264357

[17] Sato Hiroyuki. 2000. Array form representation of idiom recognition system for numerical programs. *SIGAPL APL Quote Quad* 31, 2 (Dec. 2000), 87–98. DOI: http://dx.doi.org/10.1145/570406.570418

[18] Hiroyuki, Sato. 2009. Idiom recognition and program scheme recognition based program transformations for performance tuning–beyond compiler optimizations. In *Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, 272–279.

[19] Jiahua He, Allan E. Snavely, Rob F. Van der Wijngaart, and Michael A. Frumkin. 2011. Automatic recognition of performance idioms in scientific applications. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*. IEEE, 118–127.

[20] Kawahito, Motohiro and Komatsu, Hideaki and Moriyama, Takao and Inoue, Hiroshi and Nakatani, Toshio. 2013. Idiom recognition framework using topological embedding. *ACM Trans. Archit. Code Optim.* 10, 3, Article 13 (Sept. 2013), 34 pages. DOI: http://dx.doi.org/10.1145/2512431

[21] Jens Palsberg and C. Barry Jay. 1998. The essence of the visitor pattern. In *Proceedings of the 22nd Annual International Computer Software and Applications Conference (Compsac'98)*. IEEE, 9–15.

[22] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers, Principles, Techniques* (2nd Ed.). Addison wesley.

[23] Roger A. Horn and Charles R. Johnson. 2012. *Matrix Analysis*. Cambridge University Press.

[24] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High performance convolutional neural networks for document processing. In *Proceedings of the 10th International Workshop on Frontiers in Handwriting Recognition*. Université de Rennes.

[25] IBM. 2020. *ESSL Guide and Reference (Version 5, Release 5)*.

[26] Intel. 2020. *Intel Math Kernel Library: Developer Reference Manual (Revision 26)*.

[27] OpenBLAS: An Optimized BLAS Library. Retrieved January 2020 from https://www.openblas.net/.

[28] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Softw.* 41, 3 (Jun. 2015), 14:1–14:33. http://doi.acm.org/10.1145/2764454

[29] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. Retrieved from http://eigen.tuxfamily.org.

[30] Clang: A C language family frontend for LLVM. Retrieved January 2020 from https://clang.llvm.org.

[31] N. Birkbeck, J. Levesque, and J. N. Amaral. 2007. A dimension abstraction approach to vectorization in Matlab. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'07)*. 115–130.
[32] Fu, James Jianghai. 1997. Directed graph pattern matching and topological embedding. *J. Algor.* 22, 2 (1997), 372–391.
[33] Daniele G. Spampinato, Diego Fabregat-Traver, Paolo Bientinesi, and Markus Püschel. 2018. Program generation for small-scale linear algebra applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 327–339.
[34] Norman A. Rink, Immo Huismann, Adilla Susungi, Jeronimo Castrillon, Jörg Stiller, Jochen Fröhlich, and Claude Tadonki. 2018. CFDlang: High-level code generation for high-order methods in fluid dynamics. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*. 1–10.
[35] Philip Ginsbach, Bruce Collie, and Michael F. P. O'Boyle. 2020. Automatically harnessing sparse acceleration. In *Proceedings of the 29th International Conference on Compiler Construction (CC'20)*. Association for Computing Machinery, New York, NY, 179–190. DOI: http://dx.doi.org/10.1145/3377555.3377893
[36] Verdoolaege, Sven and Guelton, Serge and Grosser, Tobias and Cohen, Albert. 2014. Schedule trees. In *Proceedings of the International Workshop on Polyhedral Compilation Techniques*.
[37] David A. Padua and Michael J. Wolfe. 1986. Advanced compiler optimizations for supercomputers. *Commun. ACM* 29, 12 (1986), 1184–1201.
[38] Louis-Noël Pouchet and Tomofumi Yuki. 2019. PolyBench/C 4.2.1: The Polyhedral Benchmark Suite. Retrieved from http://polybench.sf.net.
[39] IBM. 2018. *Power9 Processor User's Manual (Version 2.0)*.
[40] AMD. 2020. *Software Optimization Guide for AMD Family 17th Models 30h and Greater Processors (Revision 3.01)*.
[41] Agner Fog. 2019. Instruction Tables: Lists of Instruction Latencies, Throughputs and Micro-operation Breakdowns for Intel, AMD and VIA CPUs. Technical University of Denmark (08 2019), 383. Retrieved March 2020 from https://www.agner.org/optimize/instruction_tables.pdf.
[42] Tomas Kalibera and Richard Jones. 2013. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM'13)*. Association for Computing Machinery, New York, NY, 63–74. DOI: http://dx.doi.org/10.1145/2491894.2464160
[43] Tyler M. Smith, Robert Van De Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. 2014. Anatomy of high-performance many-threaded matrix multiplication. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 1049–1059.
[44] Jianyu Huang and Robert A. Van de Geijn. 2016. BLISlab: A sandbox for optimizing GEMM. arXiv:1609.00076. Retrieved from https://arxiv.org/abs/1609.00076.
[45] The Science of High-Performance Computing Group. Retrieved March 2020 https://shpc.oden.utexas.edu/.
[46] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC'09)*. 44–54. DOI: http://dx.doi.org/10.1109/IISWC.2009.5306797
[47] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Comput. Arch. News* 34, 4 (2006), 1–17.
[48] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE'18)*. Association for Computing Machinery, New York, NY, 41–42. DOI: http://dx.doi.org/10.1145/3185768.3185771
[49] P. Fischer and K. Heisey. 2013. *NEKBONE: Thermal Hydraulics Mini-Application.* Quick Starter Guide, Release 2.1, 1st ed., 2013.

Received October 2020; revised March 2021; accepted March 2021