Semiring Optimizations: Dynamic Elision of Expressions with Identity and Absorbing Elements

GUILHERME VIEIRA LEOBAS, UFMG, Brazil
FERNANDO MAGNO QUINTÃO PEREIRA, UFMG, Brazil

This paper describes a compiler optimization to eliminate dynamic occurrences of expressions in the format 
\[ a \leftarrow a \oplus b \otimes c. \]
The operation \( \oplus \) must admit an identity element \( z \), such that \( a \oplus z = a \). Also, \( z \) must be the absorbing element of \( \otimes \), such that \( b \otimes z = z \otimes c = z \). Semirings where \( \oplus \) is the additive operator and \( \otimes \) is the multiplicative operator meet this contract. This pattern is common in high-performance benchmarks—its canonical representative being the multiply-add operation \( a \leftarrow a + b \times c \). However, several other expressions involving arithmetic and logic operations satisfy the required algebra. We show that the runtime elimination of such assignments can be implemented in a performance-safe way via online profiling. The elimination of dynamic redundancies involving identity and absorbing elements in 35 programs of the LLVM test suite that present semiring patterns brings an average speedup of 1.19x (total optimized time over total unoptimized time) on top of clang -O3. When projected onto the entire test suite (259 programs) the optimization leads to a speedup of 1.025x. Once added onto clang, semiring optimizations approximates it to TACO, a specialized tensor compiler.

CCS Concepts: · Software and its engineering → Compilers; · Computing methodologies → Symbolic and algebraic manipulation; · Computer systems organization → Serial architectures.

Additional Key Words and Phrases: Compiler, Optimization, Profiling, Semiring

ACM Reference Format:

1 INTRODUCTION

Expressions that fit into the pattern 
\[ a \leftarrow a \oplus b \otimes c \]
are common in programs. In this pattern, \( \oplus \) and \( \otimes \) are binary operations of type \( T \times T \to T \). Operator \( \oplus \) has an identity element \( z \), which is the absorbing element of operator \( \otimes \). Therefore, \( z \oplus x = x \oplus z = x \) for any \( x \in T \), and \( z \otimes x = x \otimes z = z \). The multiply-add pattern 
\[ m[i,j] += p[i,k] \times q[k,j], \]
heart of matrix multiplication, is an example of this family of expressions. In Section 3.1 we show that this family is well-provided with a rich assortment of members. Indeed, any triple \( (T, \oplus, \otimes) \) forming an algebraic semiring [Hilbert 1904, Chapter IX] meets the required constraints. In this paper, we shall describe a code optimization technique that will be called semiring optimization, because it applies on the patterns mentioned above\(^1\).

\(^1\)Although we use the name “semiring optimization”, the preconditions required by the optimization can be satisfied by simpler algebras. For instance, the semiring axioms of associativity, commutativity and distributivity are not mandatory, because we do not change the order of operations—a desirable feature when dealing with floating-point arithmetics.

Authors’ addresses: Guilherme Vieira Leobas, UFMG, Brazil, guilhermel@dcc.ufmg.br; Fernando Magno Quintão Pereira, UFMG, Brazil, fernando@dcc.ufmg.br.
Because $z$ is the identity of $\oplus$, the operation $a \leftarrow a \oplus e$ is silent whenever $e = z$. Thus, this assignment can be dynamically avoided with a conditional, as Figure 1-i illustrates. Similarly, the operation $a \leftarrow b \otimes c$ reduces to $a \leftarrow z$ whenever $b = z$ or $c = z$; hence, enabling the optimization seen in Figure 1-ii. In its most general form, the expression $a \leftarrow a \oplus b \otimes c$ is silent whenever $b$ or $c$ are the absorbing element. Figure 1-iii outlines the general transformation.

\begin{verbatim}
(i) a ← a + e
1 t0 = ld a
2 t1 = ld e
3 t2 = t0+t1
4 st a t2
5 st a t2
6 if (t1) {
7 t0 = ld a
8 t4 = t3+t0
9 st a t4
}

(ii) a ← b × c
1 t0 = ld b
2 t1 = ld c
3 t2 = t0*t1
4 st a t2
5 st a t2
6 if (t0) {
7 t1 = ld c
8 if (t1) {
9 t2 = t1*t2
10 st a t4
}

(iii) a ← a + b × c
1 t0 = ld a
2 t1 = ld b
3 t2 = t1*t2
4 t3 = t1*t2
5 t4 = t3+t0
6 st a t4
7 if (t1) {
8 t2 = t1*t2
9 t0 = ld a
10 st a t4
}
\end{verbatim}

Fig. 1. Examples of the proposed optimization.

The Issue of Performance Safety. The code transformations seen in Figure 1 try to eliminate the execution of some instructions by guarding them with conditional checks. Depending on values loaded from memory, they can save several operations. For instance, whenever variable $b$ in Fig. 1-iii is zero, none of the instructions between lines 3 and 9 of the transformed program executes. However, the unrestricted application of such transformations might downgrade performance due to the insertion of branches into otherwise straight-line code. Branches burden the long-latency branch predictor [Jiménez 2003], compromise the formation of super-blocks [Hwu et al. 1993], complicate register allocation and hinder vectorization.

In this paper, we show how to implement semiring optimizations in a performance safe way. By performance safety, we mean that if a program runs for a sufficiently long time, then either the code transformation improves its speed, or does not change it in any statistically significant way. The key to achieve performance safety is profiling. Profiling, in this case, is applied online, that is, while the optimized program runs. We evaluate four different ways to carry out the optimization—two of which resort to online profiling:

- Section 4.1 discusses the simplest flavor of semiring optimization: the conditional elimination of silent stores due to operations involving the identity element.
- Section 4.2 discusses the conditional elimination of loads whose values are absorbing elements, and of the forward program slice that depends on such loads.
- Section 4.3 presents an online profiling technique implemented within loops that avoids checking loads that do not contain absorbing elements.
• Section 4.4 shows how to move the code in charge of online profiling outside the loop. This code hoisting harmonizes semiring optimization and vectorization.

Summary of Results. We have implemented the four variations of the proposed optimization in LLVM 6.0.1 [Lattner and Adve 2004]. Even though this optimization may appear, at first, innocuous, its effects are impressive when properly implemented. Section 5 supports this statement with experiments performed onto the LLVM test suite. Optimizable patterns appear within loops of 126 programs, 35 of which run for above one second when given their standard inputs. Our of this lot, 20 programs belong into PolyBench, and 13 to TSVC. The naïve optimization (Section 4.1) improves performance, with a significance level \( \alpha = 0.05 \), in 12 benchmarks (out of 35). These results use, as baseline, LLVM -O3. We have observed speedups of 1.48x, 1.45x and 1.11x in different LLVM benchmarks, for instance. However, we have also observed slowdowns of 1.26x and 1.25x in two benchmarks. This scenario improves as we move from the naïve elimination of silent stores towards the online profiler hoisted outside loops (Section 4.4). In its fourth, and most effective implementation, our optimization causes a maximum slowdown of 1.13x in one benchmark, while maintaining all the previous speedups. Overall, it delivers a performance improvement of 1.19x on the 35 benchmarks, when compared to LLVM -O3 -ffast-math. Additionally, we show that when equipped with semiring optimization, vanilla LLVM can output code for matrix multiplication on par with the code produced by TACO [Kjolstad et al. 2017], a code generation engine specialized in the compilation of linear algebra applications.

2 OVERVIEW

In this section, we will explain the optimization that we propose in this paper, and we will introduce three of its variants. Our exposition will use the naïve matrix multiplication algorithm\(^3\) seen in Figure 2. All the examples in this section are written in C, for the sake of readability; however, our optimization is meant to be implemented in the back-end of a compiler. Indeed, the implementation that we shall evaluate in Section 5 was implemented in the LLVM code generator, and requires no intervention from users.

```
void mul_ORG(float *restrict a, float *restrict b, float *restrict c, int n)
{
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
      c[i*n + j] = 0;
  for (int k=0; k<n; k++)
    c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```

Fig. 2. Matrix multiplication taken from Cormen et al. [page 332]. We have added the restrict keyword to run the experiments in this section.

Elimination of Silent Stores. The store at line 7 of Figure 2 is silent whenever the product \( a[i*n+k] * b[k*n+j] \) happens to be zero. "Silent Store" is a term coined by Lepak and Lipasti [2000a] to denote a store operation that writes in memory a value that was already there. As recently demonstrated by Pereira et al. [2018], expressions involving the value zero are a common source of silent stores. In the context of this paper, silentness happens because zero is the identity element of the addition

\(^2\)This number is the result of dividing the total running time of programs compiled without semiring optimizations by the running time of the programs compiled with it. The geometric mean of speedups is 1.06x.

\(^3\)The "naïve matrix multiplication algorithm", which orders the loops in the i-j-k sequence of induction variables, is typically adopted in textbooks [Cormen et al. 2009, p.332]. In Section 5.4 we shall analyze the faster i-k-j implementation.
operation. Thus, we can avoid the store operation by checking if \(a[i*n+k] * b[k*n+j]\) is zero. Figure 3 shows code that exercises such possibility.

```c
void mul_ESS(float *restrict a, float *restrict b, float *restrict c, int n) {
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++) {
            c[i*n + j] = 0;
            for (int k=0; k<n; k++) {
                float aux = a[i*n + k] * b[k*n + j];
                if (aux)
                    c[i*n + j] += aux;
            }
        }
}
```

Fig. 3. Naïve mat-mul after Elimination of Silent Stores.

At the first optimization level of clang, e.g., -O0, the code in Figure 3 saves one addition, plus one load and one store of \(c[i*n+j]\). At clang -O3, it saves only one addition, due to the scalarization of \(c[i*n+j]\). Non-surprisingly, the new version of matrix multiplication, i.e., function \(mul_ESS\), has worse runtime than function \(mul_ORG\) at that optimization level. Figure 4 shows this comparison, considering input matrices with increasing probability of having cells with the value zero\(^4\). Each cell has, independently, the same chance to be set to zero. Each input matrix is independently produced. In this scenario, function \(mul_ESS\) has two disadvantages, when compared with \(mul_ORG\). First, the conditional at line eight downgrades the performance of the branch predictor, as the density of zeros in the input matrices increases. Second, clang does not vectorize the innermost loop of \(mul_ESS\). In contrast, it unrolls the innermost loop of \(mul_ORG\) 40x, and parallelizes it using 8-word vectors. These shortcomings lead to the gap marked as region “1” in Figure 4.

**Elision of absorbing elements.** The guard of Figure 3 encompasses too narrow a region to be of much benefit. To widen it, we notice that either \(a[i*n+k] = 0\) or \(b[k*n+j] = 0\) is a sufficient condition for a silent store, as zero is the absorbing element of multiplication. Figure 5 uses this observation to optimize the code originally seen in Figure 2. The chart in Figure 4 reveals that this optimization starts paying off when about 35% of the elements of matrix \(A\) are the value zero. Its benefit increases noticeably with the density of zeros. Once over 90% of the elements of \(A\) are zeros, we observe a performance boost over the original matrix multiplication (compiled with clang -O3) of almost 70% (Gap "4" in lower chart of Figure 4). On the other hand, at lower densities we observe important slowdowns, which gap “2” highlights. This slowdown is due, again, to the lack of vectorization, and to poor branch prediction –both negative consequences of the conditional at line 8 of Figure 5.

**Pre-Loop Profiling.** Semiring optimization hinders vectorization because its implementation requires inserting conditional tests into straight line code that, if left untouched, would be easy to vectorize. Nevertheless, it is still possible to benefit from vectorization and our optimization. Key to this possibility is profiling. Figure 6 shows a possible way to apply profiling in this scenario. Because the profiler runs online, immediately before the program flows into the loop that contains the semiring pattern, we call this technique Pre-Loop Profiling. We adopt this name to contrast this technique with Intra-Loop Profiling, a similar –albeit simpler– methodology that applies profiling within the loop of interest. Intra-loop profiling shall be discussed in Section 4.3.

The function `sampling` invoked at line 3 of Figure 6 reads a few positions of an array to count occurrences of a value. In Figure 6, the array is matrix \(A\), and the value of interest is zero. If the

\(^4\)Data produced with LLVM 10.0.0, in an Intel Core i5 at 1.4GHz, running OSX 10.14.4, and 1000 \(\times\) 1000 matrices.
Fig. 4. Runtime of different implementations of matrix multiplication ($C = A \times B$), given different probabilities that cells from matrices $A[10^3 \times 10^3]$ and $B[10^3 \times 10^3]$ contain zero. Programs were compiled with clang -O3. The bottom figure shows the same information as the upper figure, albeit with a reduced scale along the Y-axis. The runtime of the original implementation of matrix multiplication has been removed from the bottom figure, to improve its readability. We use the following keys: ORG: the original program, without any form of semiring optimization, compiled with clang -O3 -ffast-math. EAE: program optimized with elision of absorbing elements (for details, see Section 4.2). PLP: program optimized with a pre-loop profiling that guards the elision of absorbing elements against unprofitable inputs (for details, see Section 4.4).

ratio of zeros exceeds a threshold—in this case, 0.5, then we suppress computation dependent on absorbing elements; otherwise, we run the code without any semiring optimization. Figure 4 shows that this approach recovers the performance of the original code when the density of zeros is low. Additionally, it matches the performance of elision of absorbing values when this density is high. The choice of an adequate threshold is important. In this example, 0.35 would be a better threshold than 0.5. The gap labeled "3" highlights a region where semiring optimization is profitable, but the high threshold prevents it from happening. Profiling imposes a small overhead onto the program, which the gap labelled "5" outlines. In this example, we sample 1,000 cells of matrix $A$. Nevertheless, Section 5 will show situations in which sampling has a positive effect, due to data prefetching. In
1 void mul_EAE(float *restrict a, float *restrict b, float *restrict c, int n)  
2 {
3  for (int i=0; i<n; i++)
4    for (int j=0; j<n; j++) {
5      c[i*n + j] = 0;
6      for (int k=0; k<n; k++) {
7        float t0 = a[i*n + k];
8        if (t0 != 0.0)
9          c[i*n + j] += t0 * b[k*n + j];
10      }
11    }
12 }

Fig. 5. Elision of Absorbing Elements applied onto the implementation of naïve matrix multiplication. In this example, we check if \(a\) is non-zero; however, it would also be possible to check if \(b\) is non-zero, or if their product is non-zero. More details are given in Section 3.3.

1 void mul_PLP(float *restrict a, float *restrict b, float *restrict c, int n)  
2 {
3  if (sampling(a, n, 0.0F) > 0.50)
4    mul_EAE(a, b, c, n);
5  else
6    mul_ORG(a, b, c, n);
7 }

Fig. 6. Pre-Loop Profiling applied onto the implementation of naïve matrix multiplication.

Section 4.4 we shall explain how we generate code to carry out sampling for any loop containing the semiring pattern.

3 GENERALIZING SEMIRING OPTIMIZATIONS

Section 3.1 presents a list of optimizable patterns. Section 3.3 introduces a code transformation technique that optimizes programs where these patterns occur.

3.1 A Family of Optimizable Patterns

Figure 7 shows examples of operators and values that enable the proposed optimization. Some of these operators, when combined with the proper type, form true algebraic rings. Examples include bitwise operations such as (int, OR, AND); and logical operations such as (bool, OR, AND). Other combinations do not yield even semirings, such as (float, ADD, MUL), as floating-point arithmetic lacks associativity [Knuth 1998, Sec.4.2.2]. Nevertheless, these patterns are optimizable, because they present identity and absorbing values: the pattern \(OP_1(a, OP_2(b, c))\) is optimizable whenever the absorbing value of \(OP_2\) is the identity of \(OP_1\). SHR\(_L\) stand for logical shift right, and SHR\(_A\) denotes arithmetic shift right; SHL\(_L\) and SHL\(_A\) denote the logical and arithmetic shift left equivalents. We let sz be the size of the type, in bits.

Example 3.1. The tuples (ADD, MUL, int), (ADD, MUL, float), (AND, OR, float), (OR, AND, int), (OR, AND, bool), and (ADD, SHL, int8), are examples of optimizable patterns.

Operations DIV and MOD in Figure 7 do not form semirings. However, we include them, because they contain identity (Id) and absorbing (Ab) elements, albeit position dependently. Thus, \(\theta\) is an absorbing element in DIV(\(\theta\), n), as long as n \(\neq 0\). Similarly, 1 is an identity in DIV(n, 1), regardless of the value of n. We can fit these two operations into the same optimization algorithm that Section 3.3 introduces.
Example 3.2. The following identities are examples that follow from applying operations seen in Figure 7:

\[ a == \text{ADD}(a, \text{MOD}(b, 1)) \], or \[ a == \text{OR}(a, \text{DIV}(0, b)) \].

Figure 7 includes three operators that the implementation discussed in Section 5 does not currently handle, but that also enable the proposed optimization: \text{MAX}, \text{MIN} and \text{ADD}_\text{s}. The latter represents 

\[ \text{saturating addition} \], which, in 8-bit arithmetics, can be implemented as either \((\text{uint8})(\text{MIN}(255, \text{ADD}((\text{uint16})a, (\text{uint16})b))))\), or \((a > 255 - b ? 255 : a + b)\). Minimum and maximum form a semiring on integers. Sato and Iwasaki explain how to identify these operations in program representations that lack direct implementations of them [Sato and Iwasaki 2011]. We do not handle them today out of pragmatism: these operators do not exist in the LLVM intermediate representation, and we have not found the pattern \(a = a \oplus b \otimes c\) using them (even as combinations of multiple instructions) in the benchmarks evaluated in this paper.

3.2 Safety of Semiring Optimizations in the Floating Point Domain

Semiring optimizations must be implemented with care, when applied onto floating-point types. To be safe, semiring optimizations must be restricted to finite arithmetics. Finite arithmetics is assumed in mainstream compilers such as gcc and clang if the flag \(-\text{ffinite-math-only}\) is enabled. The implementation in this paper guards the optimization with \(-\text{ffast-math}\), which enables \(-\text{ffinite-math-only}\). There are four issues that must be considered to understand the limits of this optimization when applied onto the floating-point domain:

- Zero is not the absorbing element in IEEE 754 arithmetics, due to two special values, NaN and INFINITY [Hough and Cowlishaw 2019]. The former propagates through almost every operation involving floating-point numbers. The latter also propagates throughout these operations, except when they involve NaN. So, if c is NaN or INFINITY and b is zero, then the multiplications in Figure 1-ii and Figure 1-iii will not be zero.
- Again, considering Figure 1, if a is \(-0\), b is +0, and c is +0, then \(a + b \times c\) will return +0. However, the optimization will keep a as \(-0\). Therefore, semiring optimization on the floating-point domain requires the flag \(-\text{fno-signed-zeros}\). This flag is enabled by default with \(-\text{ffast-math}\).
- the comparison at Line 8 in Figure 5 will be skipped when t0 is either minus or plus zero. If t0 is a very small nonzero value, e.g. a subnormal number, then the computation will still...
run although it might not change the value of $c[i*n+j]$. In this case the optimization is innocuous; albeit safe.

- Associativity is missing in the IEEE 754 standard. However, in this case semiring optimizations are safe, because they do not change the order in which operations happen.

3.3 Identification of Optimization Points

The function `optimize`, in Figure 8, identifies optimization points in a program for some operations seen in Figure 7. This function is invoked onto expressions that fit into the pattern $a = a \oplus b$. Figure 8 represents such patterns as $st_a (a \oplus b)$. In this case, $\oplus$ is any operation that has an identity element and $st$ is a store. Given this pattern, `optimize` invokes function `fix` over $b$. This procedure will traverse –backwardly– the data-dependence graph of $b$, looking for operations whose absorbing element is the identity of $\oplus$.

```plaintext
fun fix (VAR s, v) ⇒ (s == v)
| fix (ORB (a, b), ~0) ⇒ fix(a, ~0) U fix(b, ~0)
| fix (MUL (a, b), 0) ⇒ fix(a, 0) U fix(b, 0)
| fix (ANDB (a, b), 0) ⇒ fix(a, 0) U fix(b, 0)
| fix (ORL (a, b), T) ⇒ fix(a, T) U fix(b, T)
| fix (ANDL (a, b), F) ⇒ fix(a, F) U fix(b, F)
| fix (DIV (a, b), 0) ⇒ fix(a, 0)
| fix (MOD (a, b), 0) ⇒ fix(b, 1) U fix(a, 0)
| fix _ ⇒ Ø

fun optimize (st a ORB(a, b)) ⇒ fix(b, 0)
| optimize (st a ANDB(a, b)) ⇒ fix(b, ~0)
| optimize (st a MUL(a, b)) ⇒ fix(b, 1)
| optimize (st a ORL(a, b)) ⇒ fix(b, F)
| optimize (st a ANDL(a, b)) ⇒ fix(b, T)
| optimize (st a ADD(a, b)) ⇒ fix(b, 0)
| optimize (st a SUB(a, b)) ⇒ fix(b, 0)
| optimize (st a XOR(a, b)) ⇒ fix(b, 0)
| optimize (st a SHL(a, b)) ⇒ fix(b, 0)
| optimize (st a SHR(a, b)) ⇒ fix(b, 0)
| optimize (st a DIV(a, b)) ⇒ fix(b, 1)
| optimize (st a MOD(a, b)) ⇒ fix(b, 1)
```

Fig. 8. Identification of optimization points.

The output of function `fix` is a set of relations in the form `{s == v}`, where `s` is a variable name, and `v` is the absorbing element of some operation that uses `s`. This relation means that whenever `s` has the value `v`, then the entire expression that uses `s` can be replaced by a known constant. These two procedures, `optimize` and `fix`, have been designed to operate on programs in the Static Single Assignment format (SSA) [Cytro et al. 1989]. Therefore, every relation `{s == v}` is unambiguous, because every program variable `s` has just one definition point. The next example illustrates how these two functions work.
**Example 3.3.** Figure 9 shows the invocation of `optimize` on the program `St(a, ADD(VAR a, MUL(AND MOD(VAR b, VAR c), VAR c), VAR d)))`. Function `optimize` produces four equalities for this code snippet, namely: `{(b == 0), (c == 1), (c == 0), (d == 0)}`. The point where each equality is created is marked in gray in Figure 9. If any of these equalities hold at run time, then the store to variable `a` will be silent.

![Figure 9. Example of relations produced by `optimize`.](image)

To keep our report concise and reproducible, in this paper, we focus on the elimination of silent stores only. Nevertheless, identity patterns enable many other optimizations that we will not explore. For instance, instead of removing silent stores, we can simply remove operations like `a ⊕ b`, whenever either `a` or `b` are the identity element of `⊕`.

**Correctness.** In this section, we lay out the key invariants of functions `optimize` and `fix`. These invariants are stated on top of the semantics of the language of logical and arithmetic expressions that `fix` traverses. In this context, we define the store environment `σ : VAR → VALUE` as a function that maps variable names to values. Next, we define an evaluation function `eval : st × σ → σ`, which receives a store instruction, like those seen in Figure 8, plus an environment `σ`, and produces a new environment `σ'`. The implementation of `eval` is standard; hence, instead of defining it formally, we will only illustrate it with Example 3.4. Function `eval` lets us state the core invariant of function `fix`, which Theorem 3.5 proves.

**Example 3.4.** If `σ = {a → 7, b → 4}`, then we have that `eval(st a ADD(b, a), σ) = σ[a → 11]`, and `eval(st a ADD (DIV(a, b), a), σ) = σ[a → 8]`. We use `[]` to denote function updating, i.e.: `σ[s → v] = λx.(x = s)?v : σ(x).

**Theorem 3.5.** If `fix(b, z) = C`, then, for any `(s := v) ∈ C` and any store environment `σ`, `eval(b, σ[s → v]) = z`.

The proof is by induction on the derivation tree of `fix`. We shall consider a few cases:
- if `fix(VARs, v)`, then `C = {s := v}`. We have that `eval(VARs, σ[s → v]) = v;`
• if \( \text{fix}(\mathsf{OR}(a, b), \emptyset) \), then \( C = C_1 \cup C_2 \), where \( C_1 = \text{fix}(a, \emptyset) \) and \( C_2 = \text{fix}(b, \emptyset) \). If \( s = v \in C_1 \) (the case for \( C_2 \) is analogous), then, by induction, \( \text{eval}(b, \sigma[s \mapsto v]) = \emptyset \). Because \( \emptyset \) is the destructor of \( \mathsf{OR} \), we have that \( \text{eval}(\mathsf{OR}(a, b), \sigma[b \mapsto \emptyset]) = \emptyset \).

Corollary 3.6. Let \( \text{optimize}(\mathsf{st}(\oplus(a, b))) = C \). If \( v \in C \), then \( \text{eval}(\oplus(a, b), \sigma[s \mapsto v]) = a \), whenever \( \sigma(s) = v \).

We have that \( \text{optimize}(\mathsf{st}(\oplus(a, b))) = \text{fix}(b, z) = C \), where \( z \) is the absorbing element of \( \oplus \). From Theorem 3.5, if \( b = z \in C \), then \( \text{eval}(b, \sigma[s \mapsto z]) = z \). Thus, \( \text{eval}(\oplus(a, b), \sigma[s \mapsto v]) = \text{eval}(\oplus(a, b), \sigma[s \mapsto v, b \mapsto z]) = a \).

4 FOUR VARIATIONS OF THE OPTIMIZATION

This section presents four ways to eliminate silent stores related to semiring patterns, from the simplest (Section 4.1) towards the most complex (Section 4.4).

4.1 Version 1: Elimination of Silent Stores (ESS)

The simplest form of semiring optimization guards a store with a conditional test. We have implemented this transformation as the application of the rewriting rule earlier seen in Figure 1-i. Function \( \text{optimize} \) (Fig. 8, Page 8) gives us a set of expressions that can be guarded. Our current implementation guards only the outermost of these expressions. Thus, once we insert a conditional test in the program, the code guarded by such check is not further optimized.

Notice that the semiring pattern makes two trivial optimizations possible, which Figure 10 illustrate. In case variable \( t_0 \) is alive past the store at line 4 of Figure 10-i, then we must settle for the more conservative transformation seen in Figure 10-ii. Otherwise, we can also avoid the load of \( t_0 \), using the transformation seen in Figure 10-iii, for even greater profit. The transformations seen in Figure 10 preserve program semantics. Although trivial, we state this fact formally in Theorem 4.1, for the sake of completeness.

(i) \[
1 \quad t_0 = \text{ld } a
2 \quad t_1 = \text{ld } e
3 \quad t_2 = t_0 \oplus t_1
4 \quad \text{st } a \ t_2
\]
(ii) \[
1 \quad t_1 = \text{ld } e
2 \quad t_0 = \text{ld } a
3 \quad \text{if } (t_1 \neq z) \{ \\
4 \quad t_2 = t_0 \oplus t_1 \\
5 \quad \text{st } a \ t_2 \\
6 \quad \} \\
7 \quad \text{is\_alive}(t_0)
\]
(iii) \[
1 \quad t_1 = \text{ld } e
2 \quad \text{if } (t_1 \neq z) \{ \\
3 \quad t_0 = \text{ld } a \\
4 \quad t_2 = t_0 \oplus t_1 \\
5 \quad \text{st } a \ t_2 \\
6 \quad \} \\
7 \quad \text{is\_dead}(t_0)
\]

Fig. 10. Implementation of silent store elimination when \( z \) is the identity of \( \oplus \). (i) Original program. (ii) Program optimized when \( t_0 \) is used in instructions other than the store. (iii) Program optimized when \( t_0 \) is used only once. Annotations at Line 7 are not part of the language used to write the computations—rather, they are pseudo-code indicating that variables are either alive or dead past the point where they appear.

Theorem 4.1. If \( t_0 \) is only used at line 3 of Figure 10-i, then Fig. 10-i and Fig. 10-iii are equivalent. Otherwise, Fig. 10-i and Fig. 10-ii are equivalent.

We show equivalence from Fig 10-i and Fig 10-iii. The second part of the theorem follows from similar reasoning. If \( t_1 \neq z \), then both programs execute the same set of assignments. Otherwise, we have that \( t_2 = t_0 \oplus z = t_0 = a \), and the store is silent.
4.2 Version 2: Elision of Absorbing Elements (EAE)

The conditional elision of expressions that depend on absorbing elements is based on the invariant
stated by Theorem 3.5. Thus, if \((s==v) \in \text{fix}(b, z)\), the evaluation of \(b\) yields \(z\) whenever the
symbol \(s\) holds the value \(v\). From Corollary 3.6, \((s==v)\) is enough to yield the store \(\text{st} \ a \oplus (a, b)\)
silent. To capitalize on these observations, we proceed in the three steps below, where the pattern
\(\text{st} \ a \oplus (a, b)\) is called \(t_s\):

1. We decorate the load of \(s\) with a guard \(d\) that checks if \(s\) receives the value \(v\).
2. We move \(t_s\) to the false branch of \(d\). Thus, \(t_s\) will happen only when \(d\) is false.
3. We move to inside the false branch of \(d\) any other instruction \(i\) that is only used to compute
\(t_s\), or some other instruction \(i'\) already inside the false branch.

Example 4.2. Fig. 11-i shows the tree in Fig. 9 written in three-address format. Fig. 11-ii shows
the guard used to check if the value loaded from \(b\) is an absorbing element. Fig. 11-iii displays
the optimized program, provided that none of the temporary variables is used past the last store
instruction.

\[
\begin{array}{c}
\text{(i)} & 1 \quad t0 = ld b \\
2 & t1 = ld c \\
3 & t2 = t0 \% t1 \\
4 & t3 = ld c \\
5 & t4 = t2 \& t3 \\
6 & t5 = ld d \\
7 & t6 = t4 \ast t5 \\
8 & t7 = ld a \\
9 & t8 = t6 + t7 \\
10 & \text{st} \ a \ t8 \\
\end{array}
\]

\[
\begin{array}{c}
\text{(ii)} & 1 \quad t0 = ld b \\
2 & t1 = ld c \\
3 & t2 = t0 \% t1 \\
4 & t3 = ld c \\
5 & t4 = t2 \& t3 \\
6 & t5 = ld d \\
7 & t6 = t4 \ast t5 \\
8 & t7 = ld a \\
9 & t8 = t6 + t7 \\
10 & \text{if} \ (t0 \neq 0) \{ \\
11 & \quad \text{st} \ a \ t8 \\
12 & \} \\
\end{array}
\]

\[
\begin{array}{c}
\text{(iii)} & 1 \quad t0 = ld b \\
2 & \text{if} \ (t0 \neq 0) \{ \\
3 & \quad t1 = ld c \\
4 & \quad t2 = t0 \% t1 \\
5 & \quad t3 = ld c \\
6 & \quad t4 = t2 \& t3 \\
7 & \quad t5 = ld d \\
8 & \quad t6 = t4 \ast t5 \\
9 & \quad t7 = ld a \\
10 & \quad t8 = t6 + t7 \\
11 & \quad \text{st} \ a \ t8 \\
12 & \} \\
\end{array}
\]

Fig. 11. Checking if \(b\) is an absorbing element.

In Example 4.2, we emphasize that instructions can only be moved into the guarded region if they
are not used in expressions other than the silent store. In other words, the region that guards the
execution of a potentially silent store \(t_s\) will contain every instruction \(i\) that is part of the backward
slice of \(t_s\) –except if \(i\) is used to compute values that do not belong into this slice. Example 4.3
clarifies these observations.

Example 4.3. Figure 12 shows three optimized versions of the code snippet from Figure 11. Each
version differs on the instructions that can be placed inside the region guarded by the check on \(t0\).
Notice how, in Figure 12-iii, the liveness of \(t6\) past the store prevents several other instructions
from being moved within the guarded region.

4.3 Version 3: Intra-Loop Profiling (ALP)

Semiring Optimization is speculative, because its performance depends on program inputs. Thus,
the unrestricted application of semiring optimization might lead to runtime regression. As explained
in Section 2, regression is due to the fact that the guards that implement the optimization hinder
vectorization and complicate branch prediction. It is possible to circumvent these shortcomings via
profiling techniques.
In this section, we introduce a form of profiling of easy implementation: its deployment does not require any static program analysis. Profiling happens online, that is to say, during the execution of the profiled program already in production mode. The version of profiling that we describe in this section inserts code within the loops that contain semiring patterns; hence, we call it Intraloop Profiling (ALP). It improves branch prediction, but still hinders vectorization. Later, in Section 4.4 we will show how to hoist the profiling code outside the loop; thus, enabling also vectorization.

Figure 13 shows how ALP augments a loop with profiling code. The body of the loop is cloned twice. In other words, there will be three versions of the loop in the optimized program. Thus, in addition to the original loop, ALP creates a body with an iteration counter, that performs profiling, and another body optimized with the elision of absorbing elements (see Section 4.2). Profiling is guided by two constants. The first, NUM_ITER, determines the number of times that stores are inspected, to check if they are silent or not. The second, THRESHOLD, determines when semiring optimization should take place. One important detail of the online profiler is that it alternates.
between the versions of the target loop only once: after profiling takes place, we have enough information to decide if the execution flow should be diverted either to the original or to the optimized loop. Once a decision is made, there is no come back: the code will iterate that version of the loop, until the loop itself terminates.

Example 4.4. Figure 14 shows the implementation of ALP on the program used as an example in Section 2. For the sake of clarity, we show code written in C; however, just like all the other optimizations described in this paper, ALP is implemented at the LLVM Intermediate Representation level.

```c
void mul_ALP(float *restrict a, float *restrict b, float *restrict c, int n) {
  int silent = 0;
  int target = PROFILE;
  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
      for (int k=0; k<n; k++) {
        switch (target) {
          case Original:
            c[i*n + j] += a[i*n + k] * b[k*n + j];
            break;
          case Optimize:
            float t0 = a[i*n + k];
            if (t0 != 0.0) c[i*n + j] += t0 * b[k*n + j];
            break;
          default:
            float t = a[i*n + k] * b[k*n + j];
            silent += (t == 0.0) ? 1 : 0;
            iter += 1;
            if (iter == NUM_ITER)
              target = silent > NUM_ITER/THRESHOLD ? Optimize : Original;
            c[i*n + j] += t;
        }
      }
    }
  }
}
```

Fig. 14. Naive matrix multiplication augmented with code to implement Intra-Loop Profiling. The PROFILE constant that initializes target sends the execution flow to the default clause of the switch.

4.4 Version 4: Pre-Loop Profiling (PLP)

The online profiling technique discussed in the previous section improves the hit rate of the branch predictor, as it tends to reduce the number of branches dynamically executed. However, it is still hard to vectorize the optimized code. As an example, neither LLVM 8.0 nor gcc 6.0 can vectorize the assignment at line 9 of Figure 14. The culprit is the switch statement at line 6, which leads to three very different variations of the original loop body. To enable vectorization, we must hoist the profiling code outside the loop. This new version of online profiling shall be called Pre-Loop Profiling.

**Different hoisting strategies.** While designing the online profiler, we considered three different approaches. Although we have implemented prototypes for every one of them, only the last technique became fully functional. Nevertheless, we discuss them all, lest those who intend to expand our ideas start implementations that are difficult to conclude successfully:
• **Loop peeling:** we can split the target loop into two iterators: the first iterates `NUM_ITER` times (the number of samplings performed by the profiler); and the second completes the rest of the loop. In this form of loop peeling [Mahlke et al. 1992], both loops do the useful work present in the original program, but only the first samples memory. This strategy was our first approach to hoist the profiler outside the loop. However, although LLVM provides support to peel the innermost loop, no such support exists for the whole loop nest, and we found it difficult to craft a correct implementation.

• **Symbolic range analysis:** we can use symbolic range analysis to obtain bounds to the arrays present in LLVM’s intermediate representation. To this end, we have built a prototype that reuses DawnCC’s parametric range analysis [Mendonça et al. 2017], which is publicly available for LLVM. This implementation is an extension of a first prototype designed by Nazaré et al. [2014], and has been used to place pointer disambiguation checks in code produced by LLVM [Sperle Campos et al. 2016]. Although it handles the arrays in Polybench, the suite that we evaluate in Section 5, it is too conservative when given larger benchmarks. Hence, often it reports that data-structures range over $[0, \infty]$.

• **Program slice:** a program slice with respect to a statement $i$ in a program $P$ is the subset of $P$ that contributes to the execution of $i$ [Weiser 1981]. To profile a memory location $s$, loaded by an instruction $i_{ld}$, we extract the program slice of $i_{ld}$. Because a backward slice considers data and control-dependences, it gives us every loop nest that contributes to compute the address used in $i_{ld}$. In this paper, we have been able to successfully reuse a slicing algorithm available for the LLVM compiler [Rodrigues et al. 2016]. Hence, this was our approach of choice.

**Example 4.5.** Figure 15(i) shows the backward slice of the access $a[i*n + j]$, originally at line 7 of Figure 2. This memory access depends on the indexing variables, e.g., $i$, $n$ and $k$. Control dependences add to the slice the outermost loop (which controls variable $i$), and the innermost (which controls variable $k$). Notice that the middle loop is left out of the slice, as variable $j$ bears no influence on the memory access.

**Sampling stride.** In addition to enabling vectorization, hoisting the profiling code outside the loop of interest via backward slicing brings another advantage: we are free to choose different strides to sample memory locations. The *sampling stride* is the spatial distance between successive addresses inspected via profiling. The stride used in Section 4.3 always follows the pattern in which memory is accessed within the original loop. As an example, in Figure 14, sampling happens at line 16. Array $a$’s sampling stride is 1, and array $b$’s is $n$.

Sampling indexed by the loop’s trip count might lead to bad decisions. For instance, one of the benchmarks that we analyze in Section 5 is Cholesky decomposition. This benchmark receives a triangular matrix, in which the elements below the main diagonal are zeros. However, sampling based on the trip count, even with a large number of profiling iterations, touches only a few cells under that diagonal. Consequently, profiling misses a substantial region that contains only zeros—multiplication’s absorbing element.

**Example 4.6.** Figure 15(ii) shows the sampling function that we built after the slice in Figure 15(i). The induction variables within the two loops that constitute the slice is incremented by a parameterized interval $STRIDE$. The implementation evaluated in Section 5 sets the value of this variable at compilation time. However, nothing hinders another implementation from having this value defined per program, for instance.

**On the number of profiling iterations.** There exists a tradeoff between the number of samples collected by the pre-loop profiler, and the accuracy of the information that it reports. The larger the
void mul_ORG(float *a, float *b, float *c, int n) {
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            c[i*n + j] = 0;
    for (int k=0; k<n; k++)
        c[i*n + j] += a[i*n + k] * b[k*n + j];
}

double sampling(float *a, int n, float v) {
    int tx = 0;
    int num_iter = 0;
    for (int i=0; i<n; i += STRIDE)
        for (int k=0; k<n; k += STRIDE) {
            tx += (a[i*n + k] == v) ? 1 : 0;
            if (num_iter++ > NUM_ITER)
                break;
        }
    return tx/(NUM_ITER*NUM_ITER);
}

Fig. 15. (i) Backward slice that determines the memory access a[i*n+k]. (ii) The sampling function that is derived from this slice. Figure 6 shows the final, optimized code, with a call to function sampling. Figure 16 will show how this code is implemented in practice in the low-level representation of a program.

number of samples, the better this accuracy. However, more samples contribute towards a heavier overhead that the profiler imposes onto the programs that it aims to optimize. Our implementation of the pre-loop profiler limits the number of samples in 1,000 per loop. This value, like the sampling stride, is defined at compilation time, although users have the option to define it in a per-program basis. In Section 5 we shall consider the same limit of 1,000 samples for all the programs that we analyze.

Low-level implementation. Example shown thus far in this paper are written in C; however, the implementation of all the techniques that it introduces happens at the level of LLVM’s intermediate representation. In the case of the pre-loop profiling, there is one further difference between our presentation and our implementation: the profiler, which is seen in Figures 6 and 15(ii) as a separate function, is, in practice, inlined in the optimized code. Example 4.7 provides a more faithful view of this implementation.

Example 4.7. Figure 16 shows the backward slice of the access a[i*n + j] at line 7 of Figure 2. The third part of Figure 16 shows the code of the sampling function built out of that slice, augmented with profiling. This sampling function is the same that is invoked at line 3 of Figure 6.

5 EVALUATION
Research Questions. In this section, we shall investigate the following research questions:

RQ1 How often does the semiring pattern appear in typical benchmarks?
RQ2 What is the runtime benefit of the different versions of Semiring Optimization?
RQ3 What is the overhead of Semiring Optimization on compilation time?
RQ4 How does Semiring Optimization compare with a specialized tensor compiler?
RQ5 What is the overhead of the two profilers that we use in this paper?
Fig. 16. (i) Three-address code version of the naïve matrix multiplication algorithm seen in Fig-2. The grey triangle shows the instruction that we will slice out from the loop. (ii) The grey boxes mark the backward slice of the load of a[i*n+k]. (iii) The sampling function built out of the slice. Grey boxes show code present in the original loop.

RQ6 What is the effect of input variation on semiring optimizations?

RQ7 What is the effect of semiring optimizations on performance counters?

Runtime Setup. We have implemented Semiring Optimization onto LLVM 6.0.1 and use LLVM -O3 -ffast-math as the baseline. The optimization is applied at the level of LLVM’s intermediate representation, via a pass plugged in the LLVM’s opt tool. We run experiments on an Intel(R) i7-3770 at 3.40GHz, with 16GB of RAM with Ubuntu 16.04.

Benchmarks. We have applied semiring optimization on the programs available in the LLVM test suite, a collection with 259 programs. Semiring patterns occur in 126 of these programs; however, most of them run for a very short time to yield reliable measurements. Thus, we report runtime numbers (RQ2) for 35 programs, which run for more than one second. For the sake of space, we present detailed figures answering RQ1 and RQ3 only for the PolyBench programs. We run PolyBench [Pouchet and Yuki 2018] with the EXTRALARGE_DATASET. PolyBench consists of 30 programs written in C; 20 of them contain the pattern $a = a \oplus b$. To avoid very short runtimes, we had to increase the largest input size of six benchmarks: gemver (5x), gesummv (7x), atax (15x), mvt (5x) and trisolv (10x). We use the original data generator of each benchmark in every experiment. Thus, we have not changed the density of identity elements in these inputs. We report experimental results in vanilla PolyBench/C programs. Hence, said results are driven by the dense data initialization function that is implemented for each benchmark and may not correspond to
realistic levels of sparsity in the input data. For an in-depth study of the relationship between input sparsity and performance, we refer the reader to Figs. 4 and 21.

Measurement methodology. Runtime results are the arithmetic mean of 5 runs of each version of each program. When summarizing results across multiple benchmarks, we report geometric means and absolute ratios (total running time of original programs divided by total running time of optimized programs). Measurements are taken using the apparatus provided by LLVM in its test suite. Runtime numbers are per program, meaning that individual function invocations are not measured separately. We adopt a significance level $\alpha = 0.05$. Thus, if the results reported by original and optimized programs cannot be distinguished with a confidence of more than 95% (via Student’s Test), then we consider them as originating from the same population. Our baseline is LLVM -O3. At this level, LLVM performs vectorization, unrolling and inlining. LLVM does not perform loop transformations such as tiling, skewing and fission.

A note on static and dynamic instances. A static instance of a store instruction is the syntactic occurrence of it. A dynamic instance, in turn, is its execution. Hence, a static instance can have many corresponding dynamic instances. We call instructions involved in a semiring pattern as “marked” instructions. Given these definitions, we shall use Dyn to refer to the number of dynamic instances of store instructions in a benchmark, M-Dyn for the number of dynamic stores marked and MS-Dyn for the number of times a store marked was silent. The proportion of marked instances is the ratio \( \frac{M\text{-Dyn}}{Dyn} \). Similarly, the ratio \( \frac{MS\text{-Dyn}}{Dyn} \) gives the percentage of silent instances. Finally, the ratio \( \frac{MS\text{-Dyn}}{M\text{-Dyn}} \) measures the silentness level of marked instructions.

5.1 RQ1: Prevalence
We have measured the prevalence of semiring patterns in 259 programs from 36 benchmark suites. Out of the 259 programs, 126 (49%) contain the semiring pattern. Figure 17 shows the frequency of these patterns among them. The standard execution of these benchmarks exercise 101,107 static instances of store operations, out of which 1,569 (1%) belong into a semiring pattern. Henceforth, we shall call these instructions marked. About 41% of programs contains one or two marked stores, but this number varies significantly across benchmarks. If we average the number of marked store instructions per program, then we obtain 12 with a standard deviation of 39. This high standard deviation is due to the variability in the workload of the test suites we use. Some programs are designed for computer-intensive tasks, whereas others are not. In total, we observed 988,223,728,822 dynamic instances of store instructions (Dyn), out of which 276,072,024,712, i.e., 27%, were from marked stores (M-Dyn). Out of this lot, 29,765,251,126, i.e., 3%, were silent (MS-Dyn). Thus, 1% of all static instances contributed to 27% of all execution of store operations of which 11% were silent.

Figure 18 shows the prevalence of the semiring pattern on the PolyBench test suite. The transparent bar is the ratio \( \frac{M\text{-Dyn}}{Dyn} \) while the gray bar is \( \frac{MS\text{-Dyn}}{Dyn} \). The numbers on top of each bar are the order of magnitude of M-Dyn. Numbers inside circles report the number of static instances. Most of the benchmarks have \( M\text{-Dyn} \approx 1.0 \) and three benchmarks (cholesky, 1u, 1udcmp) have \( \frac{MS\text{-Dyn}}{Dyn} \geq 0.5 \). On 1udcmp, one static store was responsible for almost all $10^9$ dynamic instances. Approximately 60% of these instances were silent.

---

https://llvm.org/docs/TestSuiteGuide.html
Fig. 17. Histogram with the number of static store instances marked for the set of benchmarks used. The first column corresponds to the number of marked store instructions equals to 1. We have omitted some buckets in this histogram, to keep the image intelligible. The omitted number of patterns are mentioned in the figure.

Fig. 18. Prevalence of the semiring pattern on the PolyBench suite. Powers on top of bars denote absolute number of dynamic instances of stores (order of magnitude).

5.2 RQ2: Speedup

Figure 19 shows the speedup (average of five samples) achieved by the four optimizations from Section 3 over LLVM -O3. The four semiring optimizations are applied independently and exclusively, before LLVM -O3. The longest runtime of a program compiled with LLVM -O3 belongs to PolyBench’s simm: 97.34 seconds.
Fig. 19. Speedups caused by semiring optimizations onto LLVM -O3. The higher, the better. Numbers in the gray box are the runtime of all the 35 programs, with different semiring optimizations. We use the following keys to refer to benchmark suites: TV = Test Suite for Vectorizing Compilers (TSVC) [Callahan et al. 1988]; PB = PolyBench; PA = DOE_ProxyApps_C; VB = VersaBench.

Considering a confidence interval of 95%, we observed statistically significant speedups in 9 out of 35 benchmarks using the pre-loop profiler of Section 4.4. The four largest speedups were observed in VersaBench’s bmm (2.90x), and in three PolyBench programs: Cholesky (2.05), Ludcmp (1.75x), and Lu (1.71x). The simple elimination of stores discussed in Section 4.1 is less effective: speedups in the same four programs were 0.95x (a slowdown), 1.11x, 1.09x, and 1.08x. Hoisting profiling code outside the loop is essential for performance. The intra-loop profiler (ALP) of Section 4.3 gives less performance improvement than PLP: 1.92x, 1.03x, 1.04x, and 1.04x. The average speedup (geo-mean) of PLP across the 35 benchmarks is 1.061x. If we divide the running time of programs optimized with semiring optimizations by the running time of the programs without it, then we observe a speedup of 1.19x. The absolute runtime reduction is 108 out of 659 seconds.

The benefit of profiling over the unrestricted elision of code (Section 4.2) is clear once we consider vectorization. Although EAE gives us speedups in some benchmarks, it also yields large slowdowns whenever it disables vectorization. As an example, the original version of gemm is 1.66x faster than the version optimized with EAE. Pre-loop profiling recovers this slowdown in its totality: there is no statistically significant difference between the original version of gemm and the version optimized with Intra-Loop Profiling.

These Numbers in Perspective. Figure 19 reports results for 35 programs taken from the LLVM test suite (259 programs). As mentioned in the beginning of this section, these programs were

---

7Speedups observed in two PolyBench programs, Lu and Ludcmp, are due to semiring patterns in data initialization routines, not in the kernels proper. These initialization routines run much longer than the kernels.
chosen because they: (i) contain semiring patterns and (ii) run for more than one second. If we consider all the programs in the LLVM test suite, then we obtain approximately 4,260s without semiring optimizations (clang -ffast-math -O3) and 4,140s with the pre-loop profiling version of it (clang -ffast-math -O3 -semiring-plp). The resulting speedup lays between 1.020x and 1.030x. Considering only the 35 programs, we obtain 659s vs 551s (with pre-loop profiling semiring optimization); hence, obtaining a speedup of about 1.20x. The 12 programs that run for the longest time in the LLVM test suite do not contain semiring patterns within loops (the runtime is not directly related to the size of the program, in LoC). Their runtime alone accounts for approximately 2,720s. This time dominates the results.

5.3 RQ3: Impact on Compilation Time

Figure 20 shows the compilation time spent by clang -O3 plus our optimization on PolyBench. For each benchmark, we show: (i) the time taken by the optimization passes in clang -O3; (ii) the time taken to implement semiring optimization and (iii) the time taken by the LLVM analyses and transformations needed to enable semiring optimization. This category includes the following LLVM passes: instcombine, early-cse, indvars and loop-simplify. These passes are necessary to simplify the control flow graph. Notice that the time taken by clang -O3 varies according to the version of semiring optimization that we use. Variation happens mostly between the approaches that use profiling (ALP and PLP) and those that do not (ESS and EAE). This difference is due to the fact that the profiling code is inserted before LLVM -O3 runs; thus, there will be more code to be optimized.

Inspection of Figure 20 reveals that semiring optimization is practical. In absolute terms, clang -O3 takes approximately 0.30 seconds on average to compile the 20 programs in Figure 20. This number increases by 0.01, 0.01, 0.10 and 0.23 seconds when considering, respectively, SSE, EAE, ALP and PLP. The time taken by semiring optimization itself is usually shorter than the time taken by its supporting optimizations.

5.4 RQ4: Comparison with TACO

Semiring patterns are common in linear algebra, as our evaluation on PolyBench indicates. There are compilers specialized in the generation of code for this kind of applications. We believe that the current state-of-the-art approach in the field is TACO (short for Tensor Algebra Compiler) [Kjolstad et al. 2017]. Figure 21 compares TACO with clang (plus our Pre Loop Profiler). We emphasize that this figure relates two different compilers and two different optimization methodologies. TACO uses special matrix representations, whereas semiring optimization is a general code transformation that avoids unnecessary computations. These techniques are complementary, as semiring optimizations could also be implemented in TACO, like typical optimizations for sparse matrices [Akbudak et al. 2018; Liu et al. 2019].

TACO provides different data-structures to represent matrices. We have experimented with dense and sparse representations. In Figure 21, DD means that the two dimensions in the input matrices, i.e., rows and columns, are dense; SD means that the first dimension is sparse, and the second is dense in both input matrices. The other representations, DS and SS, follow similar nomenclature. ORG (short for ORiGinal) is clang -O3, and PLP is Pre-Loop Profiling, with a threshold of 50% of absorbing elements.

Figure 21 uses the same matrix multiplication algorithm seen in Figure 2, except that loops ranging on variables j and k have been switched. This is the code that TACO generates by default, because loop inversion improves cache locality and enables vectorization of the expression k*n+j. Figure 21 shows that clang -O3 plus PLP is faster than TACO using dense matrices, and approximates its speed on sparse matrices, when the density of zeros is high. Notice, however, that TACO’s code,
when using sparse matrices, has better asymptotic behavior, because a multiplication with a sparse data structures does no work where there are zeros, whereas our optimization still implements a test in this case. Thus, for very large matrices, TACO tends to outperform clang+PLP; however, considering square matrices with 1,500 rows (the largest we could fit), these two approaches are equally as fast when the density of zeros is maximal.

5.5 RQ5: Overhead

Figure 22 shows the overhead of the profiling techniques proposed in Sections 4.3 and 4.4. Results are given in terms of percentage of the original runtime. To build the figure, we made the "optimized code section" the same as the unoptimized code; hence, any change in runtime is due to profiling. When augmented with the ALP profiler (Section 4.3), the execution time of the PolyBench programs has varied within the range $[-6\%, +27\%]$. The few speedups produced by ALP are due to prefetching: perf reveals that cache misses decreased in cholesky, lu and ludcmp. PLP was more stable: we could not consistently measure any runtime variation outside the interval $[-0.4\%, +0.5\%]$. That's less than one percent of runtime variation, and cannot be considered statistically significant within any reasonable confidence interval. Notice that each program runs for at least 5s, whereas profiling accounts for milliseconds of execution. Therefore, we conclude that for long-running applications, the overhead of online pre-loop profiling is negligible.
5.6 RQ6: The Impact of Program Inputs

The optimizations proposed in this paper are dependent on input values—a fact already observed in Figures 4 and 21. Figure 23 provides more insight on this behavior. To prepare this experiment, we have changed the init_array routine used in three PolyBench kernels to insert zeros into the input matrices with the probabilities seen in the X-axis of Figure 23. We show results for three
programs: Cholesky, Gemm and Gramschmidt. We chose these three programs because they are representative of the kinds of behaviors that our optimization tends to produce.

![Figure 23. Impact of inputs onto three PolyBench programs. X-axis shows probability of a cell being initialized with a zero, in case its value can vary. Cells are independently set to zero, and matrices are produced independently.](image)

Cholesky, for instance, already manipulates diagonal matrices. Any flavor of semiring optimization is already advantageous in this case, as more than half of every matrix is set to zero. Nevertheless, Figure 23 shows that the code produced by EAE and PLP runs faster as the density of zeros increases. ESS also enhances performance, but, this improvement tends to remain constant, regardless of the input. PLP, once past the optimization threshold, is not strictly equivalent to EAE. For instance, in Gramschmidt, it yields faster code, because it reduces misses in the data cache (as observed via perf). In Gemm, the profiling overhead leads to slower code, compared to EAE. Finally, the effect of ESS: saving one addition and one store is small compared to the elimination of loads that we obtain with PLP and EAE.

5.7 RQ7: Performance Counters

Figure 24 uses performance counters to measure the effect of the different flavors of semiring optimizations onto the i-k-j version of matrix multiplication that was evaluated in Section 5.4. In this case, the expression $c[i*n+j] += a[i*n+k]*b[k*n+j]$ admits vectorization over all the memory accesses. Figure 24 clarifies why simple elimination of silent stores (ESS, Section 4.1) is
not an effective optimization for this particular benchmark, whereas EAE (Section 4.2) and PLP (Section 4.4) are. The conditional that ESS inserts within the innermost loop hinders vectorization completely—LLVM -O3 is not able to vectorize none of the memory accesses due to that branch.

Both EAE and PLP still support vectorization in this benchmark. To understand why, we refer the reader to the code in Figure 21. When applied onto this code, EAE will insert a conditional test.

Fig. 24. The effect of semiring optimizations on the ikj version of matrix multiplication, seen in Figure 21. The X-axis shows the probability that input .
guarding the load at line 3. The second part of this optimization moves the forward slice of that load to within the guard. This transformation will ensure that the innermost loop will be entirely moved inside this guard. Therefore, the innermost loop can still be vectorized.

Figure 24 evidences the point in which the online profiler activates semiring optimization. In this experiment, we adopt a threshold of 50%; that is, one the probability of finding a zero in the input matrix reaches this point, the sampling routine seen in Figures 6 and 15 activates the optimized version of matrix multiplication. It is possible to observe an abrupt change in PLP’s behavior in Figure 24 at this moment: PLP stops following ORG’s pattern, and starts following EAE’s. Figure 24 also provides some idea on the overhead of the profiler, once we observe the number of last-level-cache accesses: the sampling compromises locality during profiling; hence, more misses in the lower levels of the cache are, indeed, expected.

5.8 Discussion

The experiments described in this section show that when semiring patterns dominate computations, their elision leads to statistically significant speedups (Section 5.2). As Figure 19 indicates, gains can be dramatic, reaching three-fold levels, as observed in VersaBench’s bmm. Furthermore, online profiling tends to preserve performance-safety when applying the optimization: the overhead of the pre-loop profiler is insignificant (Section 5.5), and its benefits noticeable (Figure 19). Nevertheless, it tends to increase compilation time, although less than two-fold, when compared to the other variants of semiring optimization discussed in this paper (Section 5.3).

We emphasize that silent semiring expressions are relatively uncommon, at least in the benchmarks that we have evaluated. They appeared in 126, out of 259 benchmarks; however, most dynamic occurrences of semiring patterns were not silent—hence, non-optimizable. As a consequence, the optimization that we advocate in this paper is not as general as classic compiler techniques such as constant propagation or global-value numbering. Our experience indicates that semiring patterns are most common in linear-algebra applications. Yet, there are programs outside this domain that also tend to present semiring patterns. Examples include implementations of SAT solvers over linked lists of booleans or computation of transitive closures via boolean matrix multiplication.

Because semiring patterns are relatively uncommon, when applied onto large programs, our optimizations are unlikely to produce the large gains observed in Section 5.2. We have applied it onto the integer benchmarks in SPEC CPU2006, using the same experimental setup of that section. The elision of absorbing elements led to statistically significant gains only in perlbench. When using the reference input, the optimized version of perlbench runs in 49.80 seconds. Without the elision of absorbing elements, but still at the -O3 optimization level, perlbench runs in 51.50 seconds. No statistically significant running time difference was observed in the other 11 benchmarks, except in namd. In that case, without the guard inserted by the pre-loop profiler, we perceived a slowdown (262.9 seconds without the optimization vs 293.80 with it).

6 RELATED WORK

This paper is related to the literature concerning the study and optimization of silent stores, and also concerning value-dependent optimizations. Section 6.1 explores the first direction of related work. Section 6.2, the second.

6.1 Work Related to Silent Stores

Inspiration for this work came from recent developments in the investigation of silent stores. The term silent store was coined by Lepak and Lipasti in the early 2000’s. A silent store deposits in memory a value that was already there. Together with his collaborators, Kevin Lepak showed that silent stores are prevalent among well-known benchmarks, and that it is possible to build
hardware that mitigates their overhead [Bell et al. 2000; Lepak and Lipasti 2000b, 2002]. The growing popularization of non-volatile memory architectures, brought renewed attention to this theme [Boehm and Chakrabarti 2016]. Different research groups showed that it is possible to use profiling techniques to help developers to uncover and remove silent stores [Khan et al. 2017; Wen et al. 2017, 2018]. In 2018 Pereira et al. [2018] showed how to predict store operations that are likely to be silent statically.

The present paper differs from this foregoing literature in two ways. First, none of these previous works attempt to remove silent stores automatically via code generation techniques. In contrast, we propose a compiler optimization that affects the target program without any intervention from users. Second, although we chose to eliminate only semiring patterns that can lead to silent stores, the theoretical framework that we propose in this paper goes beyond that. As an example, Figure 1(ii) shows an example of optimization that is not associated with any silent store.

6.2 Work Related to Value-Dependent Optimizations

Much work has been done to optimize programs based on the values that they process. The bulk of this literature was published in the late nineties [Calder et al. 1997; Grant et al. 1999; Muth et al. 2000]. However, more up-to-date work also exists in the field, such as Zeroploit, an optimization technique recently published by Rangan et al. [2020]. To give the reader a glimpse on the state-of-the-art approaches on value-dependent optimizations, we summarize the differences between ours and Rangan et al.’s work. We believe that the notion of semiring optimizations generalizes the ideas introduced in Zeroploit, for while in that setting the authors focus on specializing for dynamic occurrences of zeros in the program flow, we focus on absorbing elements in general—zero being a particular case. In terms of implementation, the optimizations discussed in Zeroploit are performed manually; our work, in turn, runs as a compilation pass, and is fully automatic. Finally, as a last difference, whereas Zeroploit relies on an off-line profiler, our profiling technique is online.

The kind of patterns that we optimize, such as multiply-adds, are common in tensor algebra. Identification of algebraic objects such as groups [Rocha et al. 2016, 2019] and semirings [Sato and Iwasaki 2011] is also common practice in automatic parallelization. This kind of algebra has become very fashionable in recent years. Today, there are specialized compilers that generate high-quality code for tensor products, such as TACO [Kjolstad et al. 2017], Sparso [Rong et al. 2016] and TVM [Chen et al. 2018]. In Section 5 we showed that the optimizations that we introduce in this paper can bring a general compiler closer to TACO, which is probably the state-of-the-art tool in terms of tensor compilation. However, we do not see semiring optimizations as a competing approach. On the contrary, we believe that this technique could be used to enhance even further these specialized tensor compilers.

The online profiler that we presented in Sections 4.3 and 4.4 supports a kind of “guess-and-check” algorithm, similar to the one introduced by Sharma et al. [2013]. This family of profilers probes the nature of data, rather than execution points. This is in contrast to profiling techniques used, for instance, to carry out partial redundancy elimination [Cai and Xue 2003]. Sharma et al.’s implementation is not explicit enough to let us draw a precise comparison with our work. Furthermore, their guess-and-check algorithm is not used to implement an adaptive optimization; rather, it is used to infer loop invariants.

7 CONCLUSION

This paper has described four variations of a code optimization that avoids operations that, depending on input values, are redundant. These so called semiring optimizations can be implemented in classic compilers, and work in commodity hardware. We have observed scenarios in which semiring
optimizations speeds up the code produced by standard clang -O3 (LLVM 6.0.1) by 2x, on a single core machine. At the -O3 level, LLVM applies 58 passes onto the target program; thus, the fact that such speedups are possible was a pleasant surprise in the course of this project. We believe that semiring optimizations can have a positive impact on tensor-based computations where matrices are sparse but not sparse enough to benefit from compression. Examples of such applications include dense neural networks with dropouts.

Software. Our implementation is available at https://github.com/guilhermeleobas/phoenix.

ACKNOWLEDGMENT
This work has been made possible by grants from different research agencies, namely CNPq, CAPES, FAPEMIG and ANR (the CONTINUUM project: grant ANR-15-CE25-0007-01). Guilherme Leobas developed part of this project while interning at the Laboratoire d’Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM). We thank Abdoulaye Gamatié and the other LIRMM researchers with whom we have interacted for many insightful discussions, which have greatly contributed to increase the quality of this work. We are also indebted to OOPSLA’s referees, who made much effort to bring rigor and elegance to this paper.

REFERENCES


Donald Knuth. 1998. *Arithmetic*. Addison-Wesley, Boston, MA, USA, Chapter 4, 194–252.


