GLORE: Generalized Loop Redundancy Elimination upon LER-Notation

YUFEI DING, XIPENG SHEN, North Carolina State University, United States

This paper presents GLORE, a novel approach to enabling the detection and removal of large-scope redundant computations in nested loops. GLORE works on LER-notation, a new representation of computations in both regular and irregular loops. Together with a set of novel algorithms, it makes GLORE able to systematically consider computation reordering at both the expression level and the loop level in a unified manner. GLORE shows an applicability much broader than prior methods have, and frequently lowers the computational complexities of some nested loops that are elusive to prior optimization techniques, producing significantly larger speedups.

CCS Concepts: • **Software and its engineering** → *General programming languages*;

Additional Key Words and Phrases: program optimization, loop redundancy elimination, operation minimization

ACM Reference Format:
Yufei Ding, Xipeng Shen. 2017. GLORE: Generalized Loop Redundancy Elimination upon LER-Notation. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 74 (October 2017), 28 pages.
https://doi.org/10.1145/3133898

1 INTRODUCTION

Removing redundant computations is an effective way to speed up applications. The traditional approach, loop redundancy elimination, detects and removes computations that are invariant across the innermost loop. Many redundancies, however, span a much larger scope and often remain hidden to prior methods. Detecting them would require some careful large-scope computation reordering and reassociation at both the expression level and the loop level. They are elusive to traditional methods for their small analysis scope and weaknesses in handling irregular loops and complex control flows and dependencies.

For instance, Example 1 in Figure 1 (a) shows a code containing a `while` loop and a nested `for` loop. If we only focus on the expression in the innermost loop body, we could not find any redundant computation: variable d gets updated in every iteration of the `for` loop, and w gets updated in every "while" loop iteration. However, taking a broader view, we can see that with some large-scope reordering and reassociation of the computations, the entire code is equivalent to the form in Figure 1 (b), in which, reduction loops $\sum_i a[i]$ and $\sum_i b[i]$ are both redundantly recomputed across the `while` loop. If we take the redundant computations out of the outer-level loop appropriately, we can save many computations and speed up the execution by orders of magnitude. Both the

Authors' Emails: Yufei Ding, yding8@ncsu.edu; Xipeng Shen, xshen5@ncsu.edu.
Authors' address: Department of Computer Science, North Carolina State University, Raleigh, North Carolina, 27606, United States, US..

2475-1421/2017/10-ART74
https://doi.org/10.1145/3133898

```
w = w0;
while (d > 0.01){
  d = 0;
  for(i = 0; i <= M; i++){
    d += a[i] + b[i] * w;
  }
  w = w - 0.001 * d;
}
```

(a) Example 1

```
w = w0;
while (d > 0.01){
  A = ∑ᵢ a[i];
  B = ∑ᵢ b[i];
  d = A + B * w;
  w = w - 0.001 * d
}
```

(b) A form equivalent to Example 1

```
for (i = 0; i <= M; i ++){
  r[i] = 0;
  for (k = 0; k <= i; k++){
    for (j = 0; j <= i; j++){
      r[i] += x[i,j] * y[j,k];
}}}
```

(c) Example 2

```
for (j = 0; j <= M; j++){
  temp[j,0] = y[j,0];
}
for (i = 1; i <= M; i++){
  for (j = 0; j <= i; j++){
    temp[j,i] = temp[j,i-1]+y[j,i];
}}

for (i = 0; i <= M; i++){
  r[i] = 0;
  for (j = 0; j <= i; j++){
    r[i] += x[i,j] * temp[j,i];
}}
```

(d) A form equivalent to Example 2

Fig. 1. Illustration of large-scope loop redundancies.

needed large-scope analysis and the presence of `while` loop prevent the traditional methods to find and remove such redundancies.

Example 2 in Figure 1 (c) shows large-scope redundancies in affine `for` loops. The code computes the products of elements in two arrays, reduces them along two axes (k and j), and then stores them in a new array r. Again, if we only focus on the expressions in the innermost loop, we could not find any redundant computations, as the expression $x[i,j]$*$y[j,k]$ computes different values across different loop iterations. But redundancies expose in a larger scope: switch the order of loop k and j; the inner loop would be computing the product of $x[i,j]$ and the prefix-sum of $y[j,k]$ along k dimension (up to i). We can have a separate loop to compute the prefix-sum. If we further notice that $temp[j,i]$ equals $temp[j,i-1] + y[j,i]$, the prefix-sum can be even simplified into the first two loops in Figure 1 (d). By reusing the prefix sums, the computation of r is simplified into the bottom loop in Figure 1 (d). The computation complexity reduces to $O(M^2)$ from the original complexity $O(M^3)$.

In both examples, the redundancies require large-scope (across multiple levels of loops) computation reordering to detect and remove. As noticed in numerous studies [Cooper et al. 2008; Deitz et al. 2001; Ding et al. 2017, 2015; Drake and Hamerly 2012; Elkan 2003; Fahim et al. 2006; Goldberg and Harrelson 2005; Greenspan et al. 2000; Gupta and Rajopadhye 2006; Gutman 2004; Hamerly 2010; Ngai et al. 2006; Wang et al. 2012; Wang 2011], such large-scope loop redundancies are common, especially in applications in computational physics, chemistry, data analytics, and other domains that involve relatively complex formulae or algorithms. When translating those complex formulae or algorithms into computer programs, the developers often intuitively follow the formulae or algorithms step by step, producing logically easy-to-understand and practically easy-to-maintain code rather than trying to minimize redundant computations.

There have been some efforts in extending the scope of traditional loop redundancy elimination [Cooper et al. 2008; Deitz et al. 2001; Gupta and Rajopadhye 2006]. They have made some significant contributions towards large-scope redundancy elimination, but they are subject to two major limitations. First, they all have some strict requirements on the forms of the loops. None of them can handle mathematical operations (e.g., *sin*, *mod*) or irregular loops with complex control flows (e.g., `while` loops with `breaks`) and complicated dependences. Second, almost none of them can systemically consider the combination of loop-level computation reordering and expression-level algebraic reordering, and deal with their interplays. Tensor contraction optimizations [Hartono et al. 2006, 2005] have considered both levels of reorderings, but they are designed specifically for tensor contraction in regular `for` loops with constant loop bounds, inapplicable to common loops. As a result, both examples in Figure 1 are elusive to all prior techniques.

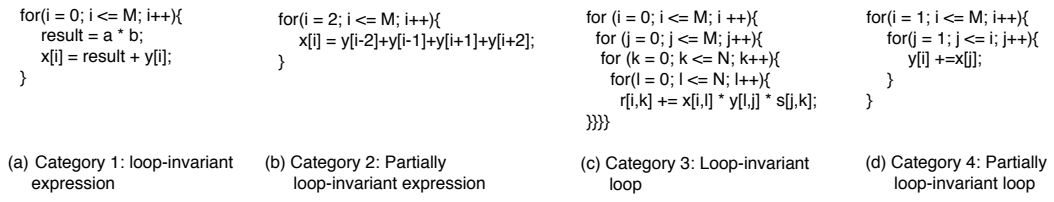


Fig. 2. Examples of the four main categories of loop redundancies.

A key observation made in our work is that the limitations of the prior techniques fundamentally stem from the lack of a proper representation of loops of various forms. For instance, a prior work [Gupta and Rajopadhye 2006] uses a high-level equational form to represent loops. As a result, it cannot accommodate `while` loops or data dependences. Moreover, it focuses on loop reordering but ignores its interplay with expression-level reordering, causing it to even miss many redundancies hidden in regular loops that it can represent.

This paper presents a new solution that addresses those challenges through the development of GLORE, which stands for generalized loop redundancy elimination. This new method introduces a notation scheme named *loop-reduction notation* (LER-notation), which provides the first unified symbolic abstraction for systematically conducting computation reordering across both loops and expressions upon the laws of associativity, commutativity, and distributivity. LER-notation equips GLORE with an applicability much broader than prior methods have, covering both regular and irregular loops and applicable to code with complex control flows, complicated dependences, and math operations. At the same time, LER-notation offers a form more friendly for the explorations of both loop and expression reordering.

To translate the increased flexibility to actual redundancy removal, we further propose a set of novel transformations and algorithms, including *operand folding*, *alternating form generation*, *minimum-union algorithm*, a linear-time *closure-based algorithm*, and so on. These techniques allow GLORE to treat the various loop complexities with ease, and to effectively detect loop redundancies by exploring loop and expression reordering and reassociations in a unified, comprehensive manner.

Experiments on 21 benchmarks from four sources show that GLORE excels in both generality and effectiveness. Working as an end-to-end framework, GLORE is able to detect and remove the most common cases in four major categories of loop redundant computations. Those cases include some large-scope redundancies that have been elusive to prior techniques, on which, GLORE gives orders of magnitude speedups by lowering their computational complexities. On loops that prior methods can handle, GLORE produces similar or significantly higher speedups.

2 FOUR CATEGORIES OF LOOP REDUNDANCY

In general, any computations that occur in multiple iterations of a loop is a loop redundancy. We classify loop redundancies into four major categories according to their granularities and repetition patterns, and GLORE is designed to tackle programs with these four major loop redundancies.

- *Category 1: Loop-invariant expressions.* If an expression's operands and operations are invariant across all iterations of a loop, the expression is a loop-invariant expression, illustrated by $a * b$ in Figure 2 (a).
- *Category 2: Partially loop-invariant expressions.* If an expression is recomputed across some but not all iterations of a loop, that expression is a partially loop-invariant expression of that loop. Such redundancy could be the outcome of array expressions that appears in some aligned formats. Figure 2 (b) offers such an example, in which, $y[i+1] + y[i+2]$ in the i^{th}

Loop invariant removal [Allen and Kennedy 2001]: category 1 redundancies only.

ASE [Deitz et al. 2001]: for sum-of-products in stencils only (part of category 2), requiring single-rank array references as operands and index expressions in a stringent form.

ESR [Cooper et al. 2008]: for common array subexpressions in categories 1 and 2.

REDUCTION [Gupta and Rajopadhye 2006]: for reductions only. No quantitative results reported. No support of "while" loops, imperfectly nested loops, and complex operations (e.g., *sin*, *cos*, etc.) in an expression.

Paige [Paige and Koenig 1982]: incremental computation across function calls.

Fig. 3. Summary of prior methods.

iteration of the loop executes the same computation as $y[i-2]+y[i-1]$ does in the $(i+3)^{th}$ iteration.

- *Category 3: Loop-invariant loops.* Let loop L_2 be a loop nested in loop L_1. If every invocation of L_2 contains exactly the same set of computations, we say that L_2 is a *loop-invariant loop* of L_1. Figure 2 (c) gives such an example: The reduction over products of $x[i,l]$ and $y[l,j]$ along axis l is repeatedly computed across loop k, and the reduction over products of $y[l,j]$ and $s[j,k]$ is repeatedly computed across loop i.
- *Category 4: Partially loop-invariant loops.* Let L_2 be a loop nested in loop L_1. If the computations by some invocations of L_2 form a subset by some other invocations of L_2, we say that L_2 is a *partially loop-invariant loop* of L_1. Figure 2 (d) shows an example. The i^{th} iteration of the outer loop computes $\sum_{j=1}^{i} x[j]$. That computation is repeated in the first part of every later iteration of the outer loop. It differs from Category 3 in that later iterations have some extra computations.

The examples in Figure 2 are intentionally made simple for understanding. The actual code could involve data dependences, irregular loops, and other complexities—as the motivating examples in Figure 1 show. One important ability of GLORE is to systematically conduct computation reordering and reassociation across both loops and expressions, so that these four categories of redundancies, if hidden in the original program, could still be exposed and removed.

3 RELATED WORK

Figure 3 summarizes the main methods developed in prior studies on removing loop redundancies. Traditional loop invariant removal [Allen and Kennedy 2001] is designed for only category-1 redundancy. Efforts to expanding the scope have each been designed for a special type of redundancy. Without establishing a general flexible way to analyze and reorder computations in a large scope, these efforts show limited applicability and effectiveness. Array Subexpression Elimination (ASE) [Deitz et al. 2001] is designed only for sum-of-products computation—stencils, where the operands must be single-rank array references and the index expressions must be of a stringent form. ESR [Cooper et al. 2008] combines value numbering and scalar replacement to explore common subexpressions made up of array references. It is designed for some loop redundancies in only categories 1 and 2, and misses redundancies that require sophisticated computation reordering.

REDUCTION [Gupta and Rajopadhye 2006] is specially designed for simplifying reductions. It utilizes polyhedral models to explore computations shared among multiple reductions. It might be able to find some redundancies in categories 3 and 4, but cannot handle "while" loops, imperfectly

nested loops, and complex operations (e.g., *sin*, *cos*, etc.). Moreover, its description stops at a theoretical level, giving neither implementation nor guidelines for implementation; no quantitative results have been reported on that technique. Hartono et al. [Hartono et al. 2006, 2005] tries to identify the most cost-effective common subexpressions for tensor contraction in electronic structure calculations so that the total number of operations could be minimized. They have strict requirements on the loop type and expression format: only `for` loop with constant loop bound, and the expressions must be products of array references and their index expressions must be of a particular form. Paige [Paige and Koenig 1982] studies how finite differencing can be used to optimize incremental computations. The optimization is at function level based on a predefined transformation library, without considering sophisticated computation reordering.

CLARITY [Olivo et al. 2015] is a recent work that shows promising results in detecting repeated traversals of arrays. Even though repeated traversals could hint on possible (not necessarily true) redundant computations, they are insufficient for precisely detecting or removing redundant computations.

Some tiling techniques for imperfect loops [Song and Li 1999] may expose some possible redundant computations as a result of the loop tiling transformations. But their main purpose and effects are on improving cache performance by restricting data footprint size, rather than detecting loop redundancies.

A recent work [Luporini et al. 2017] manages to reduce the number of operation counts for a class of finite element integration loop nests by exploiting fundamental mathematical properties of finite element operators. It can find some redundancies in categories 3, but does not handle imperfectly nested loops or complex operations (e.g., sin, cos, etc.). It neither gives any systemic considerations of the combination of loop-level computation reordering and expression-level algebraic reordering, and thus may miss some large-scope optimization opportunities.

Overall, for lack of a general approach to analyzing large-scope redundancy and comprehensive reordering, these methods are each limited to a special type of redundant computations with relatively narrow applicabilities. Even merging them together still leave lots of cases uncovered and opportunities untapped as Section 9 will show.

4 GLORE OVERVIEW

GLORE overcomes the limitations of the previous studies through two features: an LER-notation to enable flexible analysis and reordering of large-scope computations on both regular and irregular loops, and a series of novel algorithms to effectively determine the appropriate orders that minimize the amount of redundant computations.

As a result, GLORE treats the most common cases in all the four categories of redundancy—a much broader range of loop redundancies than any prior method does, finds better computation orders, and is amenable to the aforementioned various code complexities: reduction loops, regular loops, and irregular loops that are perfectly or imperfectly nested, carrying dependences or not, involving simple or complex operations (e.g., *sin*, *cos*, *log*).

We next present LER-notation and then explain how the algorithms in GLORE leverage the flexibility by the notation to analyze and reorder computations to remove each of the four categories of redundancies. We describe the conversion between code and LER-notation at the end[1].

[1]This paper uses C language terminology as GLORE is currently implemented for C programs, but the principled technique should be extensible to code in some other languages.

5 LER-NOTATION

With LER-notation, a nested loop can be represented concisely captured in some formulae, making symbolic analysis easier to apply. Our survey finds that no existing notations of loops can handle all the complexities mentioned in the previous section, whereas LER-notation solves the problem.

In LER-notation, a nested loop is represented in one or more formulae (called *LER-formulae*). Although LER-formulae can represent calculations in an arbitrary level of loops, in this work, we use them to represent the computations in the innermost loop along with all the levels of loops enclosing the computations, because of the innermost computations typically being the most costly part of a nested loop. There could be data dependences flowing into the innermost loop from other levels of loops, which would be captured by some subscripts of operands in LER-notation as described at the end of this section.

The general format of a formula in LER-notation is as follows[2]:

$$\mathrm{L\ E} \Rightarrow \mathrm{R},$$

where, L represents a sequence of loop notations, E represents an expression inside those loops, "$\Rightarrow R$" represents that the computation results are stored into variable R. The LER-representation of a nested loop is a collection of such formulae. We next explain E and L in more detail.

E and Operands Folding. The expression E may contain arbitrary mathematical computations (e.g., $sin^2(x[i])$) as long as the computations do not alter the value of the operands or other variables (i.e., free of side effects). For computations using operators beyond the common basic mathematical operators (+,-,*,/), the computations are folded into a single synthetic operand with a unique ID and with all the loop indices used in the original computations included in the operand's indexing subscript. For instance, $sin(a[i] + b[j])$ is represented as $synthetic\_ab1[i, j]$, where $synthetic\_ab1$ is the unique ID of the created synthetic operand and $[i, j]$ is its indexing subscript. We call this transformation *operands folding*. By hiding the detailed complexities in expressions but explicitly exposing the connections with the enclosing loops, operands folding makes it possible for GLORE to handle loops with complex expressions.

L and Dependence Subscripts. The loop sequence L is a combination of $\mathfrak{L}$, $\sum$, $\prod$, and $\mathfrak{W}$, which each represents one kind of loops:

(a) Regular `for` loops ($\mathfrak{L}$): $\mathfrak{L}_i^{l,u}$ represents a regular for loop with i as the loop index variable. It is assumed that the loops have already gone through normalization such that the index goes from a lower bound (l) to an upper bound (u) (which are affine expressions of loop index variables) with 1 as the step size. The following code, for instance, is represented as $\mathfrak{L}_i^{1,N}\mathfrak{L}_j^{1,M}(a[i] \cdot b[i] \cdot c[j]) \Rightarrow x[i, j]$ in LER-Notation:

```
for(i = 1; i ≤ N; i++){
    for(j = 1; j ≤ M; j++){
        x[i,j] = a[i] · b[i] · c[j];}}
```

(b) `Reduction` loops ($\sum$, $\prod$): If the loop conducts a reduction operation (e.g., summation or product) across iterations, the loop is represented as a reduction loop. In the current implementation, GLORE considers just summation ($\sum$) and product ($\prod$), which are the most commonly seen Semirings. Other Semirings are possible to be handled with minor extensions. The notation of a reduction loop is the same as a regular `for` loop except that $\mathfrak{L}$ is replaced with either $\sum$ or $\prod$.

[2]The name "LER" comes from this general form.

Loop normalization and affine loop bounds are also assumed. So, $\sum_i^{l,u}$ represents a loop in which a summation is done across its iterations with i as the loop index and l and u as the loop bounds. The following code, for example, can be represented as $b + \sum_i^{1,N} \sum_j^{1,M} a[i] \Rightarrow x$:

```
x = b;
for(i = 1; i ≤ N; i++){
    for(j = 1; j ≤ M; j++){
        x = x + a[i];}}
```

Note, if the innermost loop contains multiple statements, multiple formulae could be created with each corresponding to one of the statements. When there are values flowing between those statements, some vectorizations of scalar variables may be necessary. For instance, if there is a statement "c[i,j] = x" in the previous example loop right after the "x=x+a[i]" statement, the LER-notation would be as follows:

$$b + \sum_i^{1,N} \sum_j^{1,M} a[i] \Rightarrow x[i,j]$$
$$\mathfrak{Q}_i^{1,N} \mathfrak{Q}_j^{1,M} x[i,j] \Rightarrow c[i,j].$$

(c) `While` loops and other irregular loops ($\mathfrak{W}$): Unlike the previous two kinds of loops, a `while` loop has no loop index variable or lower or upper bound. To help identify a particular `while` loop in the notation, LER-notation gives a unique identity to each `while` loop, represented as a subscript of $\mathfrak{W}$. For instance, $\mathfrak{W}_t$ represents a `while` loop whose identity is set as t. Irregular `for` loops and `reduction` loops (e.g., with non-affine loop bounds or control flow statements—such as `break`—that may cause early termination of the loops) are represented and treated in the same way as the `while` loops, except that their loop indices are used as their identities.

When the lower bound of a *for* loop or *reduction* loop is 1, the lower bound can be omitted. In LER-notation, the subscript of $\mathfrak{Q}$, $\sum$, $\prod$, or $\mathfrak{W}$ is called the ID of that loop.

If a variable (say x) gets assigned and a loop (either *for* or *while* loop) with ID i is the innermost loop that contains that assignment, the variable carries a subscript i (e.g., x_i) in the LER-representation to explicitly indicate possible data dependences caused by the update to that operand. For instance, the w in Figure 1 (a) gets updated in the while loop; so its subscript shall carry the ID of the while loop to indicate the possible cross-loop dependences. (In this paper, to simplify the representation, we explicitly write out such subscripts only when it is necessary.) Such dependences propagate: Variables whose value comes from calculations involving a variable with such a subscript would carry that subscript themselves. To obtain these subscripts, the examination starts from the operands in the outermost loop, and gradually moves to the inner loops and propagates the subscripts throughout the process. A synthetic operand carries subscript i if any of its original operands would carry subscript i.

Examples. In LER-notation, the imperfectly nested loop in Figure 1(a) is expressed in the following formula:

$$\mathfrak{W}_t \sum_i^{1,N} (a[i] + b[i] \cdot w_t) \Rightarrow d_t. \tag{1}$$

It explicitly represents the statement in the innermost loop as that is the focus of optimization. It uses the subscript t to capture the dependences of w and d over the iterations of the *while* loop due to the statements in the outer loop.

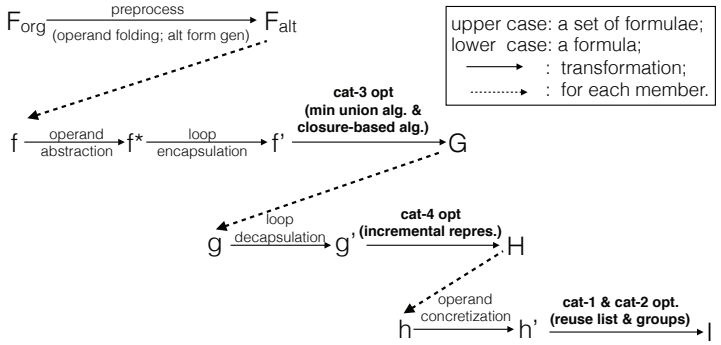


Fig. 4. GLORE transforms formulae through a series of steps to remove its loop redundancies. A final cross-formula optimization step is omitted.

The example in Figure 1(c) is expressed as

$$\Omega_i^{1,M} \sum_k^{0,i} \sum_j^{0,i} x[i,j] \cdot y[j,k] \Rightarrow r[i], \tag{2}$$

LER-notation offers a way to concisely represent both regular and irregular loops and explicitly encode possible data dependences into the representation. These properties prove essential for GLORE to achieve a much broader range of applicability and to more effectively explore computation reordering of all scopes to detect and remove large-scope redundancies than prior methods do.

6 GLORE ANALYSIS AND OPTIMIZATIONS

This section describes GLORE and explains how it works on the LER representations of loops to find and remove the four categories of loop redundancies.

6.1 Overview

Figure 4 outlines the main steps of the GLORE algorithm. The input to GLORE is a set of LER-formulae corresponding to a nested loop. Its output is a new set of LER-formulae with all redundancies GLORE finds removed.

The algorithm first preprocesses the input formulae. This step includes two main operations. The first is *operand folding* (described in Section 5), after which, operations beyond the basic algorithmic operations are replaced with synthetic operands. The second is *alternating form generation*, in which, minus is turned into negative signs associated with each of the relevant operands, and division is folded into operands as inverse. After that, the formula contains only plus or times, a form we call *alternating form*. In such a form, the expression can be regarded as a hierarchy with the levels alternating between PLUS and TIMES, as illustrated in Figure 5. The hierarchical view allows a divide-and-conquer strategy to be used, with redundancy detected and removed at each level of the hierarchy. Because the composition at each level involves either only plus or only times, it allows free reassociation and communication of the operations among the children of an arbitrary node in the tree.

GLORE then optimizes each of the original formulae individually to remove loop redundancies contained in each. After that, it examines the new set of formulae and removes redundancies that exist across the formulae.

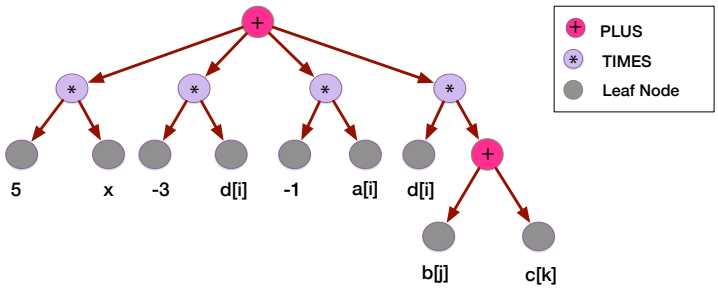


Fig. 5. The alternating form of formula $5x - (3d[i] + a[i] - d[i] \cdot (b[j] + c[k])$ is $5x + (-3d[i]) + (-a[i]) + d[i] \cdot (b[j] + c[k])$, regarded as a hierarchy with the levels alternating between PLUS and TIMES.

When optimizing an individual formula, GLORE takes a series of steps, and through the process, a formula is transformed into a series of forms. As Figure 4 shows, for a given formula f, GLORE first conducts *operand abstraction*, which replaces the index of each operand with a set of the IDs of its relevant loops. A loop is relevant to an operand if its ID appears in the index of the operand. We use *relLoops(x)* to denote the set of loops relevant to an operand x. For instance, for the following formula:

$$\mathfrak{Q}_i^{1,N} \sum_j^{1,i} \sum_k^{1,N} x[i+j] \cdot y[i, j+k] \cdot z[k] \Rightarrow w[i], \tag{3}$$

the relevant loop set is {i,j} for x, {i,j,k} for y, and {k} for z. After operand abstraction, the formula becomes

$$\mathfrak{Q}_i^{1,N} \sum_j^{1,i} \sum_k^{1,N} x\{i, j\} \cdot y\{i, j, k\} \cdot z\{k\} \Rightarrow w[i]. \tag{4}$$

This step simplifies the removal of loop redundancies of categories 3 and 4, in which, what is relevant is the set of loop indices in the indexing expressions of each operand rather than the indexing expressions themselves. We denote the resulting formula with f^*. (The concrete indexing expressions of each operand is restored in later steps.)

GLORE then applies *loop encapsulation* on f^* to convert it into a new form f', which, through the use of pseudo-bounds, hides the complexities in loop bounds such that every loop in f', other than *while* loops, has only constant bounds. GLORE then uses *minimum union algorithm* to detect and remove category-3 redundancies (loop-invariant loops) from f', yielding a new set of formulae G (Section 6.2). For each formula in G, say g, GLORE decapsulates it to get a form g' with the complexities of the loop bounds restored. GLORE then finds and removes category-4 redundancies (partially loop-invariant loops) by converting g' into an incremental representation, resulting in a new set of formulae H. GLORE then restores the concrete index expressions of operands (*operand concretization*), and removes the other two categories of redundancies by building up reuse lists and reuse groups of the expressions in the formulas.

We next give a detailed explanation of the algorithms for the removal of loop-invariant loops (category 3). As the most complex category to handle, it demonstrates how LER-notation facilitates the large-scope analysis and computation reordering for redundancy removal. The treatments to other categories follow a similar approach; we describe them briefly at the end.

6.2 Removal of Loop-Invariant Loops (Category 3)

To help understanding, we start with the case where every loop bound is constant across the iterations of the nested loop of interest, there are no loop-carried data dependences (expect regular reductions), and all loops are interchangeable in order. We discuss the other complexities later in Section 6.2.3.

A loop-invariant loop can be either a reduction loop or a "for" loop. GLORE treats redundant reduction loops first and then treats redundant regular loops.

6.2.1 Loop-Invariant Reduction Loops. We will draw on the example in Figure 2 (c) in this section. In LER notation, it can be expressed in the following formula:

$$\mathfrak{Q}_i^{1,M} \sum_j^{1,M} \mathfrak{Q}_k^{1,N} \sum_l^{1,N} x[i,l] \cdot y[l,j] \cdot s[j,k] \Rightarrow r[i,k]. \tag{5}$$

RelLoops. The removal of redundant reductions is based on *relLoops* of operands and *relLoops* of reduction loops. Recall that the *relLoops* of an operand is the set of loop IDs that appear in the indexing expressions of that operand (as defined in section 6.1). The *relLoops* of a reduction loop R is the union of the *relLoops* of all its operands whose *relLoops* contains R. Formally, it is defined as follows:

$$relLoops(R) = \bigcup_{\substack{o:o \in operands(R) \\ R \in relLoops(o)}} RelLoops(o). \tag{6}$$

For instance, the relLoops($\sum_j$) in Formula 5 is {j,l,k} because j appears in the indexing expressions of operands $y[l,j]$ and $s[j,k]$ and the union of their loop index sets is {j,l,k}. Relevant loops of a reduction tells what loops must be involved when doing the corresponding reduction. Our following discussion assumes that the reduction is a summation. The principle design is the same for multiplication-based reduction.

Formula Simplification. The input to the step for removing cat-3 redundancies is the form after the preprocessing steps and the operand abstraction. Array indexing expressions are already replaced with the relLoops of the array access. For instance, for formula $\sum_i^N \sum_j^N \sum_k^N d[2*k] + 2 \cdot a[i] \cdot sin(d[i]) \cdot (a[3*i+3]b[j] + c[k]) \Rightarrow r$, its input form to this step is $\sum_i^N \sum_j^N \sum_k^N d\{k\} + 2 \cdot a\{i\} \cdot d\_\{i\} \cdot (a\{i\}b\{j\} + c\{k\}) \Rightarrow r$, where the indexing of each array only indicates the relevance of the loops rather than the exact location of the element to access, and $d\_$ is a synthetic operand.

Before detecting redundant reductions, we further simplify the formula. If the top level of the expression in the formula is a plus node (like Figure 5 shows), the formula is broken into several formulae with each corresponding to a child of the root node. Moreover, for each of the new formula (a times node), its operands are grouped to further simplify the representation: The operands represented by its immediate child nodes are grouped into a single synthetic operand if they have the same relLoops, and each non-leaf child node turns into a single synthetic operand. For instance, the simplification result of formula $\sum_i^N \sum_j^N \sum_k^N d\{k\} + 2 \cdot a\{i\} \cdot d\_\{i\} \cdot (a\{i\}b\{j\} + c\{k\}) \Rightarrow r$ becomes

$$\sum_i \sum_j \sum_k d\{k\} \Rightarrow tmp1,$$
$$\sum_i \sum_j \sum_k 2 \cdot group\_a\_d\{i\} \cdot group\_a\_b\_c\{i,j,k\} \Rightarrow tmp2,$$
$$tmp1 + tmp2 \Rightarrow r,$$

where, the first two formulae correspond to each of the two terms of the original plus expression. The operands in the second formula are further grouped: $group\_a\_d\{i\}$ is derived from $a\{i\} \cdot d\_\{i\}$

for their identical *relLoops*, and *group_a_b_c*$\{i,j,k\}$ is derived from the single term $(a\{i\}b\{j\}+c\{k\})$. The final formula adds the results of the previous formulae to get the final result. This simplification puts each formula into a product form, offering conveniences for removal of redundant reductions as shown next.

Detecting Redundancy. Based on the simplified product form and the concept of relLoops, the detection of loop-invariant loops becomes easy:

Under the assumption that all loops in a formula are interchangeable, if loop $i \notin relLoops(R)$, where R is $\sum_j$ then the reduction loop R is invariant to loop i, which means that we may move out of loop i the reduction R of the subexpression consisting of operands that include j in their relevant loop sets.

An Example.

In Formula 5, for instance, loop i is not in relLoops($\sum_j$), which equals {j,l,k}. The formula is hence equivalent to the following:

$$\begin{aligned} &\mathfrak{Q}_l^{1,N}\mathfrak{Q}_k^{1,N}\sum_j^{1,M} y[l,j]\cdot s[j,k] \Rightarrow temp[l,k] \\ &\mathfrak{Q}_i^{1,M}\mathfrak{Q}_k^{1,N}\sum_l^{1,N} x[i,l]\cdot temp[l,k] \Rightarrow r[i,k]. \end{aligned} \tag{7}$$

The equivalence is intuitive: Because the calculation of the sum of the product of y and s has nothing to do with loop i, it does not need to be repeatedly computed inside loop i. Putting it out removes the redundant computations. The cost of the reduction is reduced from $O(N^2M^2)$ to $O(N^2M)$.

Another way to understand the benefits is that the transformation essentially changes the order of computation involved in the two reductions by leveraging the distributive property of multiplication. Given i and k, the original formula computes $r[i,k]$ as

$$\sum_j\sum_l x[i,l]\cdot y[l,j]\cdot s[j,k],$$

while the new formula computes it as

$$\sum_l x[i,l]\cdot\sum_j y[l,j]\cdot s[j,k].$$

With the inner summation being moved into a separate formula, the computational complexity decreases.

For that example, an alternative form of the resulting formulae is as follows:

$$\begin{aligned} &\mathfrak{Q}_i^{1,M}\mathfrak{Q}_j^{1,M}\sum_l^{1,N} x[i,l]\cdot y[l,j] \Rightarrow temp[i,j] \\ &\mathfrak{Q}_i^{1,M}\mathfrak{Q}_k^{1,N}\sum_j^{1,M} temp[i,j]\cdot s[j,k] \Rightarrow r[i,k]. \end{aligned} \tag{8}$$

This form differs from the previous form in the orders of computing the reduction loops, and hence the computational complexity ($O(NM^2)$ v.s. $O(N^2M)$).

This example demonstrates a critical aspect in removing redundant loops: finding the best order of the computations. We solve the problem through an algorithm named *minimum union algorithm.*

Minimum Union Algorithm.

When there are many nested loops and operands whose indexing expressions each cover some subsets of the loop indices, finding the best computation order can be a difficult problem. In fact, a previous paper [Chi-Chung et al. 1997] shows that even a much simplified version of the problem[3] is already NP-complete.

We design a heuristic algorithm, called *minimum union algorithm*, to help quickly determine a good order of reduction loops (regular loops discussed later). It produces a forest, which encodes the desired order of the reduction loops. Each node in the forest corresponds to a reduction loop. Loops on separate trees can have an arbitrary order in the produced formulae, while the loops in one tree will follow a post order (children before parent) in the produced formulae. We call the forest an *ordering forest*.

Consider the following example:

$$\sum_{i}^{N}\sum_{j}^{N}\sum_{k}^{N}\sum_{t}^{N} a[i][j]\cdot b[i][k] \Rightarrow result \tag{9}$$

Figure 6 shows the produced *ordering forest*. The t loop can be either before or after the other three loops; reduction loop j and loop k should be computed before reduction loop i.

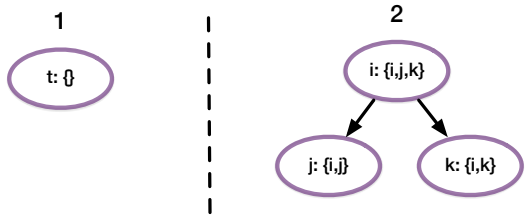


Fig. 6. The ordering forest produced by the minimum union algorithm for Formula 9 (the brackets show the *relLoops* of each reduction).

Minimum union algorithm leverages from insights. First, if the *relLoops* of reduction loop i is a subset of that of reduction loop j and does not include j, computing loop i first will allow loop j to use its results (rather than recompute its results in every iteration of loop j), lowering the computational complexity. Second, if the *relLoops* of two reduction loops i and j have no overlap, then the two reduction loops do not need to use the results from each other, and hence their order does not matter to the cost.

Figure 7 outlines our algorithm. Its input is the set of reduction loops in a given LER-formula. GLORE stores with each reduction loop its *relLoops*, and estimates its cost as the product of the ranges of the index values of all loops in its *relLoops* (symbolically represented). It produces separate trees in the result forest based on the second insight given in the previous paragraph. It takes a greedy strategy, attempting to maximize the amount of reuse. In each iteration of the while loop in Figure 7, the loop with the minimum cost is selected and added into the forest. It is temporarily put as the children of all the yet-to-process reduction loops that use its results (they are its "temporary parents"), and the algorithm (lines 10-21 in Figure 7) updates those loops' costs by considering the replacement of the (re)computation of that loop with the reuse of its result. The parent of a node is later clarified: The first of its "temporary parents" that is put into the forest is set as its actual parent (lines 22-31 in Figure 7). Note we do not need to undo the cost reduction to other

[3]In the simplified problem, only regular "for" and "summation" loops with constant bounds are allowed, and the operands must be the product of arrays whose indices must follow some strict form—for instance, a[i,j] is allowed while neither a[i,i] nor a[2i,j] is.

"temporary parents" because they will happen after this loop and can still use its results thanks to the post order in the generation of new formulae. When several orders could be the best depending on the actual loop bounds values, the analysis records all of them and their respective favorable conditions.

Application of the algorithm to Formula 9 gives the forest as shown in Figure 6.

```
1   /* inputs:  a set of reduction loops S;
2               each element has an estimated computation cost (cost)
3               and relevant loops (relLoops) recorded.
4     outputs: a forest F that records an optimized order to compute
5               the reductions.
6   */
7   worklist = createInitialNodes (S);
8   // a tree node is created for each loop with children and
9                                       // parent setting to NULL;
10  while(worklist != ∅){
11          thisLoop = the loop with the minimum cost in worklist;
12          remove thisLoop from worklist and add it into F;
13          foreach l in worklist{
14              // add thisLoop into the children list of loops
15                                      //that rely on its results
16              if  ( ( l.ID ∩ thisLoop.relLoops) != ∅) {
17                  add thisLoop into the children list of l;
18                  // update the cost of l
19                  l.cost /= thisLoop.indexRange;
20              }
21          }
22          // confirm the parent relation with its children
23          foreach c in thisLoop.children {
24              if (c.parent != NULL){  // c has a parent already
25                  remove c from the children list of thisLoop
26              }
27              else{
28                  c.parent = thisLoop;
29              }
30          }
31  }
```

Fig. 7. Minimum Union Algorithm for selecting an optimized order for reduction loops.

Removing Redundancy through Formula Generation. After getting the *ordering forest*, GLORE generates new formulae with the redundancy removed. This step involves not just the reduction loops, but also all other loops and all operands in the original formula. The generation works on the trees in the forest one after another; the order makes no difference. We explain the algorithm first and then provide an example.

When starting working on a tree T, the algorithm fills a list A with all the operands that appear in the original formula. It traverses T in a post order (children before parent). Consider a node corresponding to reduction loop R_i in an ordering forest. The algorithm creates a formula "L E $\Rightarrow$ r". L is a sequence of loop representations corresponding to the loops in $relLoop(R_i)$. E represents the product $\alpha \cdot \beta$, where α is the product of the results produced by the children of this node in the ordering forest, and β is the product of all the operands in A that have i in their *relLoops*. Those operands are then removed from A. For a single-node tree with no relevant operands (e.g., node 1 in Figure 6), E is just 1; the formula is replaced with the computation of the range $(ub - lb)$ of the loop index. The right-hand-side notation r represents a new name created by the algorithm to record the result; if L contains some regular loops, then r has index as $[d_1, d_2, \cdots, d_k]$, where d_i is the ID of the i^{th} regular loop in L. After all trees in the forest have been processed, a final formula is generated to get the product of all those results.

Applied to Formula 9, the algorithm generates the following formulae based on the ordering forest shown in Figure 6.

$$N \Rightarrow tmp0;$$
$$\mathfrak{Q}_i^N \sum_j^N a[i][j] \Rightarrow tmp1[i];$$
$$\mathfrak{Q}_i^N \sum_k^N b[i][k] \Rightarrow tmp2[i];$$
$$\sum_i^N (tmp1[i] \cdot tmp2[i]) \Rightarrow tmp3;$$
$$tmp0 \cdot tmp3 \Rightarrow result.$$

The first formula corresponds to the reduction loop t, which contains no relevant operand or child. The second formula comes from the left child of the second tree which corresponds to the reduction loop j. As its *relLoops* contains only i and j, the formula contains loop i as a regular loop and the reduction loop j. That node has no children and hence the expression in the formula contains only the product of all the operands that have j in their *relLoops*, which are just $a[i][j]$. The result is stored into a new name $tmp1$, whose index contains only i. The third formula comes from the right child of the second tree in a similar manner. The fourth formula comes from the root of the second tree. Because after the generation of the second and third formulae, both $a[i][j]$ and $b[i][k]$ have been removed from the operand list A, there is no operand in A that has i in its *relLoops*. Therefore, the expression of this third formula contains only the results from its two children nodes, $tmp1[i]$ and $tmp2[i]$. The final formula gets the final result by multiplying the results from different trees. The overall computational complexity reduces from $O(n^4)$ to $O(n^2)$.

6.2.2 Loop-Invariant Regular Loops. Redundant regular loops can be detected and removed upon LER-notation in a similar manner, through a different algorithm named *closure-based algorithm*. It works on each of the formulae produced in the previous step.

We use the following example for our discussion.

$$\mathfrak{Q}_i^N \mathfrak{Q}_j^N \mathfrak{Q}_k^N x[i] \cdot d[j,k] \cdot c[i,j] \Rightarrow w[i,j,k]. \tag{10}$$

Such computations are "cross products" that are commonly seen in computational physics, where each operand represents some values in a lower dimensional space, and the result gives the values in a higher dimensional space.

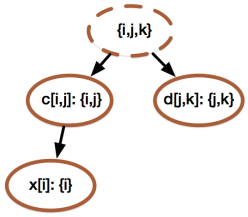
There are two kinds of optimization opportunities for such regular loops.

The first is about synthetic operands. If $x[i]$ in Formula 10, for instance, is a synthetic operand (defined in Section 5) that involves non-trivial computations (e.g., $sin^2(t[i])$), then computing x[i] cross loops j and k would be redundant. It could be avoided if we put the computations of all $x[i]$ $(1 \le i \le N)$ into a separate formula. Transformations to exploit this kind of opportunities is easy to do: Just put the operand and its relevant loops into a separate formula.

The second is about reuses across subexpressions. When there are multiple operands, their computations could be split into multiple steps (each as a separate formula), such that later steps can reuse, rather than repeatedly compute, the results of earlier steps. For example, the following formulae compute the same results as Formula 10 does, but requires only $N^2 + N^3$ multiplications, rather than the $2N^3$ multiplications needed by the original formula.

$$\mathfrak{Q}_i^N \mathfrak{Q}_j^N x[i] \cdot c[i,j] \Rightarrow temp1[i,j]$$
$$\mathfrak{Q}_i^N \mathfrak{Q}_j^N \mathfrak{Q}_k^N temp1[i,j] \cdot d[j,k] \Rightarrow w[i,j,k]. \tag{11}$$

The complexity in exploiting this kind of opportunities is again on ordering: There may be many possible orders in which the expressions could get computed. For Formula 10, a form alternative to



(a)

```
1  // nodeSet: nodes that share a parent in the closure tree;
2  foreach nd in nodeSet{
3      nd.remainingIndexSpace = nd.indexSpace;
4  }
5  seenIndexSet = ∅;
6  while (nodeSet is not empty){
7      thisNode = the node having the smallest remainingIndexSpace;
8      generateFormula(thisNode);
9      remove thisNode from nodeSet;
10     extraIndexSet = thisNode.indexSet - seenIndexSet;
11     extraIndexSpace = computeSpace (extraIndexSet);
12     if (extraIndexSpace > 0) {
13         // update the remainingindexSpace of the other nodes
14         foreach nd in nodeSet {
15             nd.extraIndexSpace /= extraIndexSpace;
16         }
17     }
18     seenIndexSet = seenIndexSet ∪ extraIndexSet;
19 }
```

(b)

Fig. 8. (a) The operand closure tree for Formula 10. (b) Closure-based algorithm for finding a good order for the operands that share a parent in an operand closure tree.

Formula 11 is as follows:

$$\begin{aligned} \mathfrak{L}_i^N \mathfrak{L}_j^N \mathfrak{L}_k^N x[i] \cdot d[j,k] &\Rightarrow temp1[i,j,k] \\ \mathfrak{L}_i^N \mathfrak{L}_j^N \mathfrak{L}_k^N temp1[i,j,k] \cdot c[i,j] &\Rightarrow w[i,j,k], \end{aligned} \tag{12}$$

which is more costly than Formulae 11 due to the order in which it involves the operands in the computation.

For an arbitrary expression, finding the optimal order is NP-complete in general [Chi-Chung et al. 1997]. We design a linear-time closure-based heuristic algorithm to solve the problem. It is based on a concept we introduce, *operand closure tree*. Each node in the tree, except the root, corresponds to an operand in the expression of the LER-formula to optimize, and carries the *relLoops* of that operand in it. The root is an artificially added node to put all nodes into one tree structure; its *relLoops* consists of all the loop IDs in the formula to optimize. An important property of the tree is that a child's *relLoops* must be a subset of its parent's—hence the name "operand closure tree". This property helps GLORE find good orders. Figure 8 (a) shows the operand closure tree of Formula 10. Figure 8 (b) outlines the closure-based algorithm, which finds a good order through a post-order walk over the closure tree. Before the walk, each non-root node has an *indexSpace* computed, which equals the product of the ranges of all the loops in its *relLoops*. Through the post-order walk, the algorithm uses a greedy strategy to iteratively decide the order of the children of each node. Its design tries to make the union of the index sets of the selected operands enlarge slowly, which helps maximize the amount of result reuse and hence effectively avoid unnecessary computations. Through the ordering process, new formulae are generated to incrementally compute the product of the children of a node (putting one more children into each new formula), and then creates a formula to compute the multiplications between that product and the parent node.

The search algorithm is for general cases. For loops with only a small number of operands and loops, exhaustive search could be used to find the best.

6.2.3 Extra Complexities. This subsection describes how GLORE handles non-constant loop bounds and data dependences when it removes category-3 redundancies.

Non-Constant Bounds. It uses *loop encapsulation* to handle loops with non-constant bounds. Consider the following example.

$$\mathfrak{Q}_i^{1,N} \sum_{j}^{1,i} \sum_{k}^{1,N} x[i] \cdot y[j] \cdot z[k] \Rightarrow w[i] \tag{13}$$

where, the upper bound of loop j is i.

The basic idea of loop encapsulation is to use a pseudo-loop with constant bounds to replace a group of loops that may contain non-constant bounds and have dependences among their indices. Its application to Formula 13 gives

$$\sum_{t,\{i\}}^{1,N^2} \sum_{k}^{1,N} x\{t\} \cdot y\{t\} \cdot z\{k\} \Rightarrow w[i],$$

where, loop t is a pseudo-loop for the group of loops {i,j}, and the subscript {i} records the regular loop (loop i) in that group; the upper bound of loop t (N^2) is a simple rough estimation of the size of the combined iteration space; the operand x and y both have t in their *relLoops* because they are relevant to some (or all) loops in that loop group. After encapsulation, the formulae turn into a form amenable for the previously described optimization algorithms to apply. Technical report([TR 2017] contains the full algorithm of loop encapsulation.)

Data Dependencies. When there are loop-carried data dependences, there may be certain restrictions on loop reordering in the optimizations. Many classic techniques have been developed before for detecting loop-carried data dependences, and to recognize the legality of a new order of loops according to the data dependences [Allen and Kennedy 2001]. These techniques can be used to reveal the data dependences in a nested loop.

Based on these analysis results, GLORE can ensure that its transformations produce legal formulae. Specifically, GLORE avoids dependence violations through an annotation scheme and two principled rules. The annotation scheme is the subscripts of operands for specifying that the operands are subject to some data dependence across certain loops. An example is the subscript t in w_t in Formula 1, which indicates the data dependence of w across the *while* loop in Figure 1(a). Such annotations apply to "for" and reduction loops as well. The annotations allow GLORE to follow two conservative rules to prevent any dependences violations:

(1) If a new formula's expression contains no operands that have dependence subscripts, the formula is safe to create. The correctness comes from the classic *loop transformation theorem* [Allen and Kennedy 2001]: Loop reordering is safe if there are no loop-carried data dependences.

(2) Whenever GLORE tries to create a new formula containing operands with dependence subscripts, it must include into the formula all the loops that any of the operands in the new formula depends on, and at the same time, loop reordering is allowed only among the loops inside the innermost dependence-carrying loop to avoid dependence violations.

Control Flow Statements. Our technique applies regardless of whether the loops contain "if", "continue", "break", or other control flow statements. These statements are not explicitly expressed in our LER notations. But the dependences they induce are kept in the optimizations through dependence subscripts of variables in the notations and some constraints on loop reordering. If the computations in the expression included in an LER-notation has control or data dependences on one of such control flow statements, the variables in those expressions are marked with dependence subscripts of all the loops enclosing that control flow statements, which prevents the reordering involving those loops to observe the dependences.

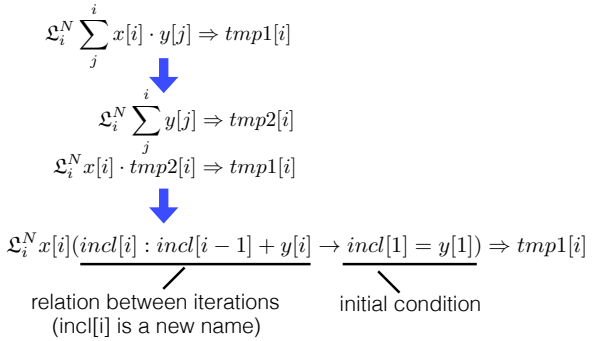


Fig. 9. Example for removing redundancies of categories 4.

6.3 Other Categories

This section describes the algorithms for other categories briefly. Readers may refer to our technical report [TR 2017] for details.

Category 4: GLORE first decapsulates the encapsulated loops. It then detects possible category 4 redundancies (*partially loop-invariant loops* as Figure 2 (d)) by matching some common patterns (e.g., affine loop bounds) with the formulae. From the innermost to the outermost loops, it moves an operand out of a loop if its *relLoops* do not contain the loop's index, as the top step in Figure 9 illustrates. It then reformulates the partially loop-invariant loops with an incremental form as the bottom step in Figure 9 (a) shows. In the code generation step, it removes the original redundant computations by replacing them with incremental computations. When there are multiple reduction loops, GLORE applies the minimum union algorithms discussed in Section 6.2 to decide the reduction order. After that, for each reduction formula it generates, GLORE checks, in the LER-notation, whether the union of the relLoops of all the operands in "R" is a proper subset of the union of the loop IDs in 'L". If so, it tries to reformulate the partially loop-invariant loops into an incremental form. This category of redundancy is often seen in cases where so-called *incremental computing* [Hammer et al. 2015, 2014] has been applied. What GLORE contributes in this part is a mechanism for incorporating the detected incremental computations into the optimization of LER-formulae for automatically removing the redundancies.

Category 2: In all the cases we have discussed, some operands use only a subset of the loop indices, creating the redundancies. When all operands cover all loop indices, there can still be redundant computations. One typical example is stencil-like computations, such as

$$\mathfrak{L}_i\mathfrak{L}_j(A[i][j-1] + A[i-1][j] + A[i+1][j] + A[i][j+1]) \Rightarrow C[i,j],$$

where, the first half of the expression (A[i][j-1] + A[i-1][j]) in iteration (i, j) conducts the same computations as the second half of the expression (A[i+1][j] + A[i][j+1]) does in iteration $(i-1, j-1)$. GLORE deals with these redundancies after removing the redundancies of categories 3 and 4.

GLORE first takes an *operand concretization* step to reverse the effects of *operand abstraction* such that the operands in the formulae now have their concrete indexing expressions.

We draw on the following example to explain the algorithm for removing category-2 redundancies:

$$\begin{aligned} &\mathfrak{Q}_i\mathfrak{Q}_j 6 \cdot \underline{A[i,j]} \\ &+2 \cdot \underline{(A[i,j-1] + A[i-1,j] + A[i+1,j] + A[i,j+1])} \\ &+\underline{A[i+1,j-1] + A[i+1,j+1] + A[i-1,j+1] + A[i-1,j-1]} \\ &\Rightarrow B[i,j]. \end{aligned} \tag{14}$$

We introduce a term—*reuse direction vector*. It is represented by a tuple $(\delta_1, \delta_2, \cdots, \delta_k)$, where, δ_i indicates the change to the ith level of the loop index when one moves by one step along the direction represented by the tuple. For instance, (1,1) represents the direction in which the two levels of loop both increase by one for one-step movement along the direction of interest. Consider the second underlined expression in Formula 14. It shows clear redundant computations along direction (1,1): The expression "A[i,j-1]+A[i-1,j]" is identical to the expression "A[i+1,j]+A[i,j+1]" in the previous iteration along the reuse direction.

Our observations show that two reuse directions cover most common cases: One is along the innermost loop, the other is along the diagonal direction. GLORE explores each of these two reuse directions on the target formula and settles on the direction that helps remove more redundancies.

We next explain the algorithm we have designed to detect such repeated computations. The algorithm runs in a top-down traversal of the hierarchical representation of a formula. It contains four steps.

Step-1: Grouping. This step puts the operands in the subexpression denoted by the leaf nodes into a number of *reuse groups*. Two operands fall into the same group if they meet two conditions: (1) They have the same identifier and access pattern—that is, they have the same indexing expression except that the constant offsets could differ. For instance, A[2i, 2j] and A[2i, 2j+3] could get grouped, but no two of A[i,j], A[2i,j], A[j,i] B[i,j] qualify. (2) The reuse of any two operands in one group must be along the currently explored reuse direction. For instance, A[i,j] and A[i-2,j-2] would be grouped if the reuse direction is (1,1), but not if it is (0,1). Take the third underlined subexpression in Formula 14 as an example: In direction (0,1), those four operands fall into two groups: $\{A[i+1, j-1]$, $A[i+1, j+1]\}$, and $\{A[i-1, j-1]$, $A[i-1, j+1]\}$.

Step-2: Sorting. This step sorts operands in a reuse group according to the order in which they access a particular address (along the reuse direction). An *ordered list* is created, from the earlier access to the latest. For $A[i+1, j-1]$ and $A[i+1, j+1]$, the ordered list is {A[i+1,j+1], A[i+1,j-1]} (assuming that reuse direction (0,1) is being studied). Suppose that the first element in the *ordered list* accesses a particular memory location x, and that the i^{th} element of the list access the same memory location d iterations later along the reuse direction, we record d as the *reuse distance* of this i^{th} element in the list. Based on the indexing expressions in an *ordered list*, GLORE derives a *reuse distance list* with the i^{th} element being the reuse distance of the i^{th} element in the *ordered list*. The first element in a *reuse list* is always set to 0. For instance, for $A[i+1, j+1]$ and $A[i+1, j-1]$, along direction (0,1), the *reuse distance list* is [0,2].

Step-3: Inner-group redundancy elimination. When a reuse group contains more than three operands, those operands could form multiple subexpressions. For example, a group with four operands $A[i+1]$, $A[i]$, $A[i-1]$, $A[i-2]$, which could have come from the expression $A[i+1] + A[i] + A[i-1] + A[i-2]$, can be regarded as the sum of two subexpressions $A[i+1] + A[i]$ and $A[i-1] + A[i-2]$. GLORE tries to recognize regular strides in the reuse distance list, based on which, it detects repeated computations among the subexpressions. For instance, based on the reuse distance list of the four operands in our example, [0,1,2,3], GLORE would discover that (2,3) is a repetition of (0,1) with offset 2, and further examine the corresponding subexpressions to validate that the second subexpression repeats the calculations of the first regularly.

Step-4: Inter-group redundancy elimination. The subexpressions from different reuse groups may also form some repeated computations. Take the third underlined subexpression in Formula 14 again as an example: It has two reuse groups, $\{A[i+1,j-1], A[i+1,j+1]\}$, and $\{A[i-1,j-1], A[i-1,j+1]\}$, as we have analyzed. Their reuse distance lists are both [0,2]. Having the same reuse list (more generally, the same stride in the reuse lists) is a clue for possible inter-group redundancies. In this example, the inter-group subexpression $A[i+1,j-1]+A[i-1,j-1]$ repeats the work of $A[i+1,j+1]+A[i-1,j+1]$, forming a redundancy. The detection is similar to step-3.

For each of the redundant subexpression detected, GLORE creates a new formula to store its computation result to an intermediate operand, and then replaces the corresponding computations in the original formulae with the intermediate operands. The optimized formulae for Formula 14 are as follows:

$$\begin{aligned} \mathfrak{L}_i\mathfrak{L}_j A[i+1,j]+A[i-1,j] &\Rightarrow tmp1[i,j]. \\ \mathfrak{L}_i\mathfrak{L}_j 6\cdot A[i,j]+2\cdot(A[i,j-1]+A[i,j+1]+tmp1[i,j]) & \\ +tmp1[i,j-1]+tmp1[i,j+1] &\Rightarrow B[i,j]. \end{aligned}$$

Most previous work [Hammer et al. 2015, 2014] for extended redundancy eliminations have focused on this category of redundancies. Compared to those work, our contributions are three-fold: First, the LER-notation introduces a more structured format such that the associativity and communication can be explored separately; second, we propose a way to group indexed operands, which helps better expose inter-subexpression redundancies; finally, with the groups, we employ a layered search strategy, finding more types of redundancies efficiently. Without a systematic format of formula representations, previous studies can handle much limited cases. For example, some work [Deitz et al. 2001] targets only sum-of-product stencils of array computations. Some work [Cooper et al. 2008] uses value numbering to explore redundant computation and relies on compiler's def-use chain to group array operands, but their grouping is coarse-grained, and redundancies within a group are missed. Section 9 will give some quantitative comparisons.

Category 1: This is the easiest category to detect and remove. Traditional compiler techniques are already doing that. We just note that they can be directly handled by GLORE in the process of optimizing loop-invariant loops (category 4) as they are simply a special, simpler case of such redundancies: It can be regarded as having a pseudo loop inside the innermost loop, having only one iteration.

7 CODE-LER CONVERSION

This section briefly describes the conversion between code and LER-notation. Based on the Clang Libtooling Library [Lattner and Adve 2008] included in LLVM, we build a prototype to derive LER-notation from code. At the core are two data structures. *LoopNode* stores loop indices, bounds, depths, break conditions for a "whileâĂİ loop, and a pointer to its parent LoopNode. *ExprNode* stores information for a variable assignment statement, a pointer to the loop in which the assignment happens and pointers to the statements that define some values used in this statement. The tool emits LER-formulae for the relevant statements inside the inner-most loops based on these info, and replacing those statements in the original code with some placeholders, leaving a code frame. The converter employs the default alias analysis in LLVM. Complex aliases could cause difficulties for the conversion. But GLORE is mostly for computing-intensive scientific or analytics programs, in which, complex pointer or aliases usage is uncommon.

After the LER-formulae get optimized, a code generator generates code from the formulae and inserts them into the code frame left in the LER conversion. Our prototype is developed on the code

generator in the symbolic tool Sympy [Joyner et al. 2012]. The key point to notice is that when an LER-formula is produced from the original code by our conversion tool, the tool also keeps a copy of the original code; in that copy, it replaces the statements represented by LER-formula (loop control statements and some relevant statements in the innermost loop) with some placeholders and keeps the remaining code in the copy to form a code frame, which helps the code recovering. For the computation in an LER-formula, the generator puts the computation code at its corresponding placeholder in the code frame if that placeholder and the computation have the same series of enclosing loops. It helps ensure the necessary control and data dependences. If there is no such location, new loops for that computation can be safely created and inserted before the original loops.

8 AN END-TO-END EXAMPLE

This section uses Example 1 in Figure 1(a) to show the complete lists of formulae that GLORE creates during its optimizations and the final optimized codes that GLORE generates.

As outlined in Figure 4, GLORE starts with Example 1 in the LER-notation form, F_{org}, as shown in the following list of formulae. As there is only one single formula, and it is already in the *alternating form* with plus as the leading operator, GLORE directly proceeds to the *operand abstraction* step. The f^* formula gives the result after transformation, where the index of each operand is replaced with a set of the IDs of its relevant loops. The next step, *loop encapsulation*, is skipped as all loops (except the *while* loop) already have only constant bounds. After applying the *minimum union algorithm* to detect and remove category-3 redundancies (loop-invariant loops) from f^*, GLORE yields a new set of formulae G. GLORE then restores the concrete index expressions of operands (operand concretization) and produces the set of formulae $\{h'\}$. The optimization finishes as there are no other categories of redundancies; the computational complexity is lowered from $O(N * W)$ to $O(N + W)$ (where W is the number of iterations of the *while* loop).

$$F_{org}:$$

$$\mathfrak{W}_t \sum_{i}^{1,N} (a[i] + b[i] \cdot w_t) \Rightarrow d_t. \tag{15}$$

$$f^*:$$

$$\mathfrak{W}_t \sum_{i}^{1,N} (a\{i\} + b\{i\} \cdot wt) \Rightarrow d_t. \tag{16}$$

$$G:$$

$$\sum_{i}^{1,N} a\{i\} \Rightarrow tmp1 \tag{17}$$

$$\sum_{i}^{1,N} b\{i\} \Rightarrow tmp2 \tag{18}$$

$$\mathfrak{W}_t (tmp1 + tmp2 \cdot w\{t\}) \Rightarrow d_t. \tag{19}$$

$$\{h'\}:$$

$$\sum_{i}^{1,N} a[i] \Rightarrow tmp1 \tag{20}$$

$$\sum_{i}^{1,N} b[i] \Rightarrow tmp2 \tag{21}$$

$$\mathfrak{W}_t(tmp1 + tmp2 \cdot w_t) \Rightarrow d_t. \tag{22}$$

Following the optimized formulae, GLORE will generate the following optimized codes:

```
tmp1 = 0;
tmp2 = 0;
for(i = 1; i ≤ M; i++){
    tmp1 += a[i];
    tmp2 += b[i];
}
w = w_0;
while (d > 0.01){
    d = tmp1 + tmp2 * w;
    w = w - 0.001 * d;
}
```

9 EVALUATION

This section reports the speedups brought by GLORE. Our Code-to-LER translation is implemented in LLVM 3.7.0 for the easy extensibility of LLVM. We use GCC 4.8.4 for compiling both the original code and our GLORE optimized code to get executables. For the rich set of optimizations implemented in GCC, we use compilation flags "-O3 -msse3" in all cases to ensure that the standard optimizations are applied to all versions of code.

To test the robustness across machines, we employ two platforms: an Intel Core i5-4570 CPU (3.2 GHz) with 16 GB memory (denoted as *core-i5*), and an Intel Xeon E5-2620 v3 CPU (2.4 GHz) with 32 GB memory (denoted as *xeon-E5*). We compare with two prior techniques for extending the scope of redundancy removal for loops: ASE [Deitz et al. 2001] and ESR [Cooper et al. 2008]. They are the most recent studies that we have found that have either code published (ASE) or enough instructions for implementation (ESR). A prior work on reduction [Gupta and Rajopadhye 2006] makes some theoretical contributions, but giving no descriptions on how it can be implemented or any quantitative results.

9.1 Benchmarks

We collect a set of 21 benchmarks to cover the various levels of redundancy. In each of the benchmarks, there is a primary nested loop taking most of the time. What GLORE rewrote are those primary loops, which were also the targets of the previous studies compared in the paper. Other loops (e.g., those for data initializations) were not rewritten as no redundancies were found in them. These benchmarks come from four sources. In table 1, we show the main category of Redundancy existing in the primary loop of each benchmark. In the following, we will give details of the primary nested loop in each benchmark program.

Table 1. Category of Redundancy in the primary loop of each benchmark program.

Category of Redundancy	Benchmarks
2	*Dibligbilharm*, *Diso3X3*, *Imorph*, *Noise1*, *Iyoki*, *Inevatia*, *Dresid*, *LANL POP*, *mgrid:resid*, *mgrid:psinv*, *mgrid:interp*, *mgrid:rpj3*.
3	*ccsd*, *fuse*, *priv2*, *example1*, *ssymm*, *PDE*, *BGD*
4	*fmir*, *example2*

(1) The first five benchmarks are from the Pluto benchmark suite [Bondhugula et al. 2008a,b]: *ssymm, fuse, priv2, fmri, and ccsd*. For instance, *ssymm* has a three-level nested primary loop with two statements inside the innermost loop: one statement carries out reduction over the innermost loop, while the other conducts a regular assignment. Program *fuse* has a primary nested loop with one outer regular `for` loop and two separate `reduction` loops inside it. For all the other three benchmarks, there are only one innermost loop and one statement inside it, for instance, *priv2* has a three-level nested loop, where the two outer loops are regular `for` loops and the innermost loop is a *reduction* loop with constant loop bounds; the primary loop in *fmir* is a two-level nested loop, where an affine `reduction` loop is nested within a *regular for* loop; *ccsd* has a primary loop with eight nesting levels, in which the outer six loops are regular `for` loops and the inner two loops are `reduction` loops with constant loop bounds. They have redundancies at various granularities, including these large scope redundancies in category 2 and 3. Details are in section 9.2.

(2) The second source is some real-world problems, including the Batch Gradient Descent (*BGD*) algorithm and the nonlinear Partial Differential Equation (*PDE*) solver. *BGD's* primary loop has four nesting levels, where at the outermost level there is a `for` loop with control flow statements (break), and at the innermost level there are two separate `reduction` loops with values flowing from the first to the second reduction loop. *PDE* also has a three-level nested primary loop, where twelve statements reside inside in innermost `for` loop. We also include the two *examples* in Figure 1 that we have used throughout our explanation in this paper.

(3) The third is the stencil benchmarks, which perform a sequence of sweeps to updates each array element using its neighboring array elements in a fixed pattern, generally in a two or three-dimensional regular grid. These benchmarks are used in prior work [Deitz et al. 2001], including seven real-world stencil kernels: *Dibligbilharm* (25 points two-dimensional) in Biharmonic operator, *Diso3X3* (9 points two-dimensional) for partial derivatives, *Imorph* (21 points two-dimensional) for mathematical morphology, *Noise1* (9 points two-dimensional) for noise cleaning, *Iyoki* (4 statements; each as a 3-point stencil in two dimensions) for connectivity number, *Inevatia* (6 statements; each as a 23-point stencil in two dimensions) for gradient edge detection, *Dresid* (21 points three-dimensional) stencil from NAS MG. The stencil benchmarks mainly feature category-2 redundancies.

(4) The last source is the programs used in another prior work [Cooper et al. 2008]. It includes five benchmarks: *LANL POP, mgrid:resid, mgrid:psinv, mgrid:interp*, and *mgrid:rpj3. LANL POP* is a kernel extracted from the Los Alamos National LabâĂŹs Parallel Ocean Model program, which has nested *regular for* loops with three levels. Inside its innermost loop, there are 16 statements with complex operations (e.g., *sin*, *cos*), featuring category-2 redundancies across these statements. The other four benchmarks are the top-4 most time-consuming routines in *mgrid* from SPEC CPU2000. Each of these four benchmarks has a primary three-level nested loop with one statement inside its innermost loop, conducting stencil-like computations. Similar to the previous stencil benchmarks, the main redundancy in these programs also resides in the innermost loop; the difference is that a

redundant expression in this set is more general, consisting of multiple array operands with various indexing expressions instead of the simple “sum-of-product" form.

In our survey of benchmarks, we found almost no category 1 redundancies. Such traditional redundancies are probably more common in the low-level translated code (e.g., array address calculations).

9.2 Overall Performance

Figure 10 reports the speedups over 21 benchmarks on two machines. As only slight differences exist in the results due to the machine differences, we focus our discussion on the first machine. For the large range of the speedups, we set the top of the Y-axis to 2X, and annotate the speedups larger than that.

What GLORE rewrote are those primary loops in the benchmarks, which were also the targets of the previous studies compared in the paper. Other loops (e.g., those for data initialization) were not rewritten as no redundancies were found in them. The rewriting time by either GLORE or other previous techniques (ASE and ESR) for each of the loops is less than a millisecond, negligible compared to the execution times of the loops, and our evaluation thus mainly focuses on the comparison of the performance of the original codes and generated optimized codes. Among all the rewritten loops, 12 have redundancy in category 2, two in category 3, and seven in category 4.

On the leftmost 9 benchmarks in Figure 10, none of the prior methods can find and remove any redundancies. GLORE, on the other hand, produces over 20X speedups on most of them. Among all the 21 benchmarks, GLORE outperforms ESR on 20 of them, and gives the same speedup on the other one. It outperforms ASE on 14 of them, and gives same speedups on the other seven. Besides the two prior methods, we also implement a combined version, which uses both prior methods to remove as much redundancy as possible. The result turns out to be the same as just getting the better result out of those from the two prior methods separately, shown as the “combined-prior” bars in Figure 10. We can see that the combined method is on par with GLORE only on 8 of the 21 benchmarks. On the others, GLORE shows significantly higher speedups, demonstrating the benefits of its broader applicability and more advanced algorithms for reordering explorations. In the following, we give a detailed analysis of the results.

9.2.1 Redundant Loops. The first nine benchmarks in Figure 10 have loop redundancies of categories 3 and 4. None of the prior methods can handle them.

For these large-scope optimizations, GLORE moves one or more entire loops, and hence often reduces the computation complexity by some orders of magnitude. For instance, *fmir* and *example2* have redundancy in category 4, where the affine loop related reduction can be computed efficiently in the incremental manner. Program *ccsd* has category 3 redundancies, on which, GLORE separates the computations in two nested reduction loops into two individual loops. In the code of *example1*, *fuse* and *priv2*, there are reduction loops that are independent of some outer regular (or while) loops and are moved out by GLORE to be computed alone. Program *fuse* furthermore has two identical reduction computations; GLORE eliminates one of them.

When computation complexity changes, the speedups usually increase as the problem instance becomes larger. The speedups shown in Figure 10 for the leftmost nine benchmarks are attained when the size of each loop (# of iterations) is set to 100. Figure 11 shows the speedups of the leftmost six benchmarks as the loop size increases. The rapid increase of the speedups echoes the effects of the reduction of the computational complexities on these benchmarks.

The other three of the nine programs show smaller speedups. Unlike the first six benchmarks, on these three programs, the optimized part is only a fraction of the core computations. On *pde* and *ssymm*, for instance, multiple LER-formulae are produced, and only some of them have redundancies

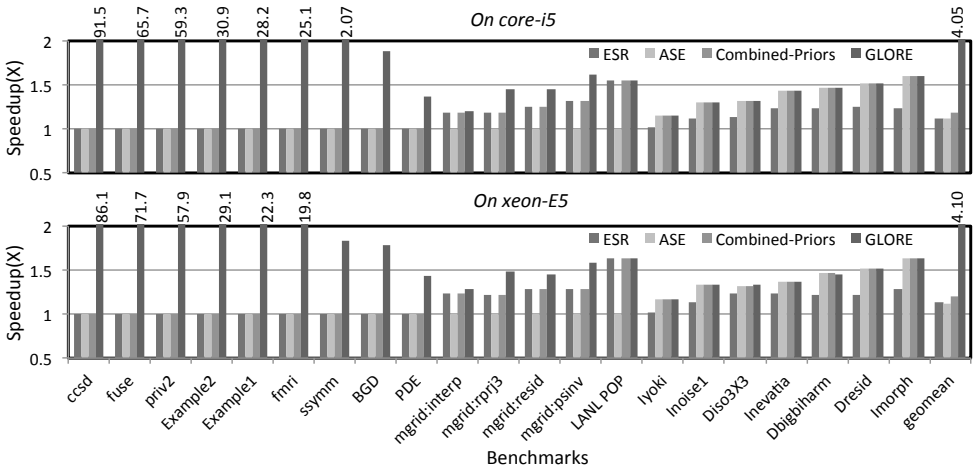


Fig. 10. Speedups on benchmarks optimized by GLORE, compared to the performance by two previous techniques ASE and ESR, and their combination ("combined-prior") on two machines. The baseline is the performance of the original benchmarks.

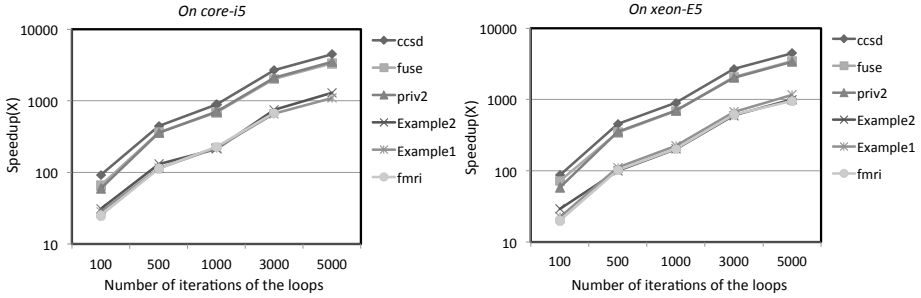


Fig. 11. Speedups by GLORE on six benchmarks with (partially) loop-invariant loops with different loop iterations.

(the parts with redundancies take 46% and 53% of the overall execution time, respectively). The overall speedup is constrained by the ratio of the optimized part over the whole program.

Among all these benchmarks, there are loop dependences (except regular reduction) in *example 1, ssymm and bgd*. GLORE treats them effectively and yields the large speedups. The machine differences entail slightly different performance results on the two machines, but the trends are the same and the speedups are consistently significant.

9.2.2 Redundant Expressions. On the rightmost 12 benchmarks in Figure 10, the redundancies are (partially) loop-invariant expressions. The previous methods, ESR and ASE, were designed to handle such redundancies, but with much limited applicability.

The first five benchmarks were from the ESR paper [Cooper et al. 2008]. On them, GLORE gives up to 1.61X speedup with an average of 1.45X, which is greater than the 1.30X average speedup from ESR. The better performance comes from our sophisticated search strategy. For example, in the benchmark *mgrid:resid*, the redundant expressions are comprised of operands from the same array. When building up an affinity graph to decide a good association of calculations, ESR considers only

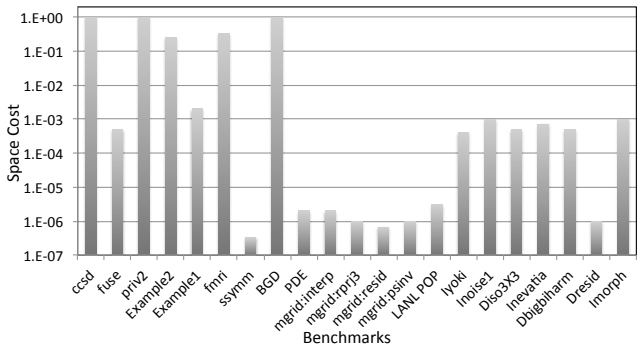


Fig. 12. Space cost of the optimized code by GLORE.

operands from different arrays and hence misses the opportunities. ASE, on the other hand, can not work on these benchmarks due to its rigid requirement on the code format. As mentioned in Section 2, ASE is designed for perfectly nested loops with sum-of-products computation—stencils, where the operands must be single-rank array references and the index expressions must be of a stringent form.

For the seven stencil benchmarks from the ASE paper [Deitz et al. 2001], GLORE works similarly well as the specialized technique ASE does, achieving up to 1.60X speedup with a 1.40X average. ASE outperforms GLORE slightly on two out of the 12 programs (*dbigbiharm* and *imporph*) for a better computation order it happens to use on them. ESR does not work as well on these programs for its coarse-grained grouping and limitations in handling operand associations.

9.3 Space Cost

GLORE optimizations introduce some assistant arrays to hold intermediate results for reuse, and hence incur some space cost and possibly also some negative impact on memory bandwidth or cache performance. The performance impact is marginal compared to the benefits since all our reported speedups already include all those effects.

The space cost is shown in Figure 12. Among the 21 benchmarks, 16 of them have space cost less than 1% of the space used by the original program. Two of the other ones have a cost around 30%. The largest space cost happens on *ccsd*, about two times. For categories 2 and 1, the space cost is usually small because of the lower dimensions of the assistant arrays than the original data sets. Consider category 2, as the reuse is in one specific loop direction, the assistant arrays generated by GLORE only need to be stored along that dimension, even if the original data is multi-dimensional. The space cost for categories 3 and 4 can be larger, but at most linear to the space of the original loops.

In all the tested cases, we have seen favorable results from the reduction of computations. Nevertheless, using detailed performance models to find the best tradeoff between storing overhead and recomputation reduction could be worth of future exploration.

10 DISCUSSIONS

In this section, we discuss the correctness of GLORE, its limitations, some usage complexities related with it, and possible ways to extend it.

The correctness comes from the design of GLORE, which ensures that the transformations violate no dependences (through the dependence analysis and subscripts), and that the moved code observes the property of value invariance in the corresponding code scope. In our experiments, we

checked the correctness by running some tests of the original and the transformed code, as well as manually checking the code semantic consistency.

There are two main limitations of GLORE. First, as Section 6.3 mentions, GLORE detects category-4 redundancies (partially loop-invariant loops) through matching against some common patterns (e.g., affine loop bounds) with the formula, whereas, incremental computations that get revealed only through some index space transformations are not yet handled directly by GLORE. Combinations of space transformations with GLORE could potentially extend the coverage. Second, as Section 6.3 mentions, for category-2 redundancies, GLORE focuses on stencil-like index expressions, which are among the most common and important types of redundancy in this category. To make GLORE cover cases beyond stencil in category two, some extensions would be necessary.

GLORE uses a heuristic search algorithm for scalability as the problem is NP hard in general. Naturally, there could be examples on which the solution is not optimal. The hybrid strategy mentioned in Section 6.2.2 could help find the best results for loops with a small number of operands and loops. Our experiments did not enable the exhaustive search module; the results indicate that GLORE still outperforms prior methods substantially.

The heuristics used by GLORE for finding good reuse directions are simple but covers the most common cases we have observed in our survey. The simplicity is key for it to work as an automatic tool with a good balance between coverage and complexity, which, we believe, is crucial for this automatic tool to cover significantly more cases and give much larger speedups than prior automatic tools do. That said, GLORE is extensible; more heuristics for reuse directions could be added to enlarge the coverage even further.

There are many traditional loop transformation techniques. They are largely complementary to GLORE. GLORE is a source-to-source transformer; our experiments used GCC to compile both the original and the transformed code. Therefore, the reported large speedups already reflect the combined effects of GLORE and the many loop transformations already implemented in GCC. Our observation is that the other loop transformations (including those in Pluto) are mostly complementary to GLORE transformations. They can be applied after GLORE for further optimizing the loops. Some could be applied early to make the code more amenable for GLORE to work with. For example, *loop skewing* [Allen and Kennedy 2001] replaces some loop index with an expression of the loop indices (e.g., $k = i - j$), which sometimes may expose even more redundant computations. Maximizing the synergy between them is left to future studies.

Just like many traditional compiler optimizations, order changes by GLORE could, in rare case, cause concerns on the output precision. Compilation flags for maintaining precision by avoiding aggressive optimizations have long become a part of standard compilers. They could be extended to GLORE optimizations.

Equipped with the insights we have provided on the four categories of redundancies, a programmer could manually refract a piece of code to remove some of the redundancies that the code contains. As there are many possible ways for redundancies to exist and many of the redundancies get exposed only after appropriate reordering of loops and operations, our experience shows that such a process requires some careful code examinations as well as the trials of many possible reordering of loops and expressions in the code. It is hence often both tedious and prone to missing some of the redundancies. In comparison, GLORE offers an approach to examining the possible orders automatically and efficiently, and provides a way to check and remove the four categories of redundancies in a systematic manner.

11 CONCLUSIONS

In this paper, we have presented GLORE and shown that it can remove large-scope redundancies from a much broader range of loops than existing methods do, and yield significantly larger speedups

overall. The key benefits of GLORE come from two aspects. First, the introduced LER-notation offers a way to concisely represent the key operations in loops with various complexities. Its explicit representations of dependences also offer conveniences in the reordering process. Second, its *operand folding* and *alternating form generation* techniques, along with the set of new algorithm designs, help GLORE treat complex math operations with ease, and more importantly, allow GLORE to explore loop reordering and expression reordering and reassociations in a unified process. These novel techniques help GLORE substantially expand the scope and effectiveness of loop redundancy elimination.

ACKNOWLEDGMENTS

We thank the reviewers for the helpful comments. This material is based upon work supported by DOE Early Career Award (DE-SC0013700), and the National Science Foundation (NSF) Grants No. 1455404, 1455733 (CAREER), and 1525609. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE or NSF.

REFERENCES

R. Allen and K. Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann Publishers.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008a. http://pluto-compiler.sourceforge.net.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008b. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Lam Chi-Chung, P Sadayappan, and Rephael Wenger. 1997. On optimizing a class of multi-dimensional loops with reduction for parallel execution. *Parallel Processing Letters* 7, 02 (1997), 157–168.

Keith Cooper, Jason Eckhardt, and Ken Kennedy. 2008. Redundancy elimination revisited. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 12–21.

Steven J Deitz, Bradford L Chamberlain, and Lawrence Snyder. 2001. Eliminating redundancies in sum-of-product array computations. In *Proceedings of the 15th international conference on Supercomputing*. ACM, 65–77.

Yufei Ding, Lin Ning, Hui Guan, and Xipeng Shen. 2017. Generalizations of the theory and deployment of triangular inequality for compiler-based strength reduction. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 33–48.

Y. Ding, X. Shen, M. Musuvathi, and T. Mytkowicz. 2015. TOP: A Framework for Enabling Algorithmic Optimizations for Distance-Related Problems. In *Proceedings of the 41st International Conference on Very Large Data Bases*.

Jonathan Drake and Greg Hamerly. 2012. Accelerated k-means with adaptive distance bounds. In *5th NIPS Workshop on Optimization for Machine Learning*.

Charles Elkan. 2003. Using the triangle inequality to accelerate k-means. In *ICML*, Vol. 3. 147–153.

AM Fahim, AM Salem, FA Torkey, and MA Ramadan. 2006. An efficient enhanced k-means clustering algorithm. *Journal of Zhejiang University SCIENCE A, Springer* 7, 10 (2006), 1626–1633.

Andrew V Goldberg and Chris Harrelson. 2005. Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM*. 156–165.

Michael Greenspan, Guy Godin, and Jimmy Talbot. 2000. Acceleration of binning nearest neighbor methods. In *Vision Interface, IEEE*. 337–344.

Gautam Gupta and Sanjay V Rajopadhye. 2006. Simplifying reductions.. In *POPL*, Vol. 6. 30–41.

Ronald J Gutman. 2004. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks.. In *ALENEX/ANALC*. 100–111.

Greg Hamerly. 2010. Making k-means Even Faster.. In *SDM, SIAM*. 130–140.

Matthew A Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S Foster, Michael Hicks, and David Van Horn. 2015. Incremental Computation with Names. *arXiv preprint arXiv:1503.07792* (2015).

Matthew A Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S Foster. 2014. Adapton: Composable, demand-driven incremental computation. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 156–166.

Albert Hartono, Qingda Lu, Xiaoyang Gao, Sriram Krishnamoorthy, Marcel Nooijen, Gerald Baumgartner, David E Bernholdt, Venkatesh Choppella, Russell M Pitzer, J Ramanujam, et al. 2006. Identifying cost-effective common subexpressions to reduce operation count in tensor contraction evaluations. In *International Conference on Computational Science*. Springer, 267–275.

Albert Hartono, Alexander Sibiryakov, Marcel Nooijen, Gerald Baumgartner, David E Bernholdt, So Hirata, Chi-Chung Lam, Russell M Pitzer, J Ramanujam, and P Sadayappan. 2005. Automated operation minimization of tensor contraction expressions in electronic structure calculations. In *International Conference on Computational Science*. Springer, 155–164.

David Joyner, Ondřej Čertík, Aaron Meurer, and Brian E Granger. 2012. Open source computer algebra systems: SymPy. *ACM Communications in Computer Algebra* 45, 3/4 (2012), 225–234.

Chris Lattner and Vikram Adve. 2008. http://clang.llvm.org.

Fabio Luporini, David A Ham, and Paul HJ Kelly. 2017. An algorithm for the optimization of finite element integration loops. *ACM Transactions on Mathematical Software (TOMS)* 44, 1 (2017), 3.

Wang Kay Ngai, Ben Kao, Chun Kit Chui, Reynold Cheng, Michael Chau, and Kevin Y Yip. 2006. Efficient clustering of uncertain data. In *Data Mining, 2006. ICDM'06, IEEE*. 436–445.

Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static detection of asymptotic performance bugs in collection traversals. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 369–378.

Robert Paige and Shaye Koenig. 1982. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4, 3, 402–454.

Yonghong Song and Zhiyuan Li. 1999. New tiling techniques to improve cache temporal locality. *ACM SIGPLAN Notices* 34, 5 (1999), 215–228.

TR. Omitted to Avoid Conflicts with Blind Review, 2017. Generalized Loop Redundancy Elimination upon Formula-Based Redundancy Removal. In *http://goo.gl/j4UKAp*.

Jing Wang, Jingdong Wang, Qifa Ke, Gang Zeng, and Shipeng Li. 2012. Fast approximate k-means via cluster closures. In *Computer Vision and Pattern Recognition (CVPR), IEEE*. 3037–3044.

Xueyi Wang. 2011. A fast exact k-nearest neighbors algorithm for high dimensional search using k-means clustering and triangle inequality. In *Neural Networks (IJCNN), IEEE*. 1293–1299.