Loop Optimization for Aggregate Array Computations[*]

Yanhong A. Liu and Scott D. Stoller

March 1997

Abstract

An aggregate array computation is a loop that computes accumulated quantities over array elements. Such computations are common in programs that use arrays, and the array elements involved in such computations often overlap, especially across iterations of loops, resulting in significant redundancy in the overall computation. This paper presents a method and algorithms that eliminate such overlapping aggregate array redundancies and shows both analytical and experimental performance improvements. The method is based on incrementalization, *i.e.*, updating the values of aggregate array computations from iteration to iteration rather than computing them from scratch in each iteration. This involves maintaining additional information not maintained in the original program and performing additionally enabled optimizations. We reduce various analysis problems to solving inequality constraints on loop variables and array subscripts, and we apply results from work on array data dependence analysis. Incrementalizing aggregate array computations produces drastic program speedup compared to previous optimizations. Previous methods for loop optimizations of arrays do not perform incrementalization, and previous techniques for loop incrementalization do not handle arrays.

1 Introduction

We start with an example—the local summation problem in image processing: given an n-by-n image, compute for each pixel $\langle i,j\rangle$ the sum $sum[i,j]$ of the m-by-m square with upper left corner $\langle i,j\rangle$. The straightforward program (1) takes $O(n^2m^2)$ time, while the optimized program (2) takes $O(n^2)$ time.[1]

```
for $i := 0$ to $n-m$ do
   for $j := 0$ to $n-m$ do
      $sum[i,j] := 0;$
      for $k := 0$ to $m-1$ do
         for $l := 0$ to $m-1$ do
            $sum[i,j] := sum[i,j] + a[i+k, j+l]$
```
(1)

```
for $i := 1$ to $n-m$ do
   for $j := 1$ to $n-m$ do
      $b[i-1+m, j] := b[i-1+m, j-1] - a[i-1+m, j-1] + a[i-1+m, j-1+m];$
      $sum[i,j] := sum[i-1,j] - b[i-1,j] + b[i-1+m,j]$
```
(2)

Inefficiency in the straightforward program (1) is caused by aggregate array computations (in the inner two loops) that overlap as array subscripts are updated (by the outer two loops). We call

[*]This work is supported by junior-faculty start-up grants from Indiana University. Authors' address: Computer Science Department, Indiana University, 215 Lindley Hall, Bloomington, IN 47405. Phone: (812)855-{4373,7979}. Email: {liu,stoller}@cs.indiana.edu. Corresponding author: Yanhong (Annie) Liu.

[1]For simplicity, initializations of *sum* and *b* for the array margins are omitted here. The full program is in Section 6.

this *overlapping aggregate array redundancy.* Figure 1 illustrates this: the horizontally filled square contributes to the aggregate computation $sum[i-1,j]$, and the vertically filled square contributes to the aggregate computation $sum[i,j]$. The overlap of these two squares reflects the redundancy between the two computations. The optimization for eliminating it requires explicitly capturing aggregate array computations in a loop body and, as the loop variable is updated, updating the results of the aggregate computations incrementally rather than computing them from scratch. In the optimized program (2), $sum[i,j]$ is computed efficiently by updating $sum[i-1,j]$. Finding such incrementality is the subject of this paper, and it is beyond the scope of previous compiler optimizations.

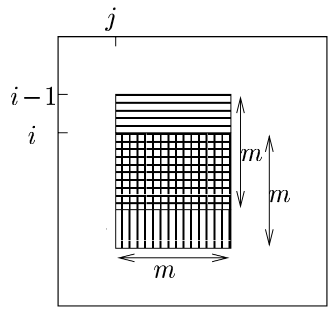

Figure 1: The overlap of the two squares shows the redundancy in the straightforward program (1) for the local summation problem.

There are many applications where programs can be written easily and clearly using arrays but with a great deal of overlapping aggregate array redundancy. These include problems in image processing, computational geometry, computer graphics, multimedia, matrix computation, list processing, graph algorithms, distributed property detection [27, 28], serializing parallel programs [10, 19], *etc.* For example, in image processing, computing information about local neighborhoods is common [22, 37, 69, 71, 73, 74]. The local summation problem above is a simple but typical example [71, 73].

Overlapping aggregate array redundancy can cause severe performance degradation, especially with the increasingly large data sets that many applications are facing, yet methods for eliminating overlapping aggregate array redundancy have been lacking. Optimizations similar to incrementalization have been studied for various language features, such as strength reduction of arithmetic operations [4, 14, 31, 32], finite differencing of set and bag operations [51, 52, 54, 53, 72], and promotion-and-accumulation, finite differencing, and incrementalization of recursive functions [9, 43, 45, 46, 47, 63], but no systematic technique handles aggregate computations on arrays. At the same time, many optimizations have been studied for arrays, such as various APL compiler optimizations [26, 36, 49], loop fusion [3, 5, 29, 66], pipelining [2], and loop reordering [7, 39, 50, 56, 62], but none of them achieves incrementalization.

This paper presents a method and algorithms for incrementalizing aggregate array computations. The method is composed of algorithms for four major problems: (1) recognizing an aggregate array computation and how its parameters are updated, (2) transforming an aggregate array computation into an incremental computation with respect to an update, by exploiting array data dependence analysis and algebraic properties of the primitive operators, (3) determining additional values not maintained in the original program that need to be maintained for the incrementalization, using a method called cache-and-prune, and (4) forming a new loop using incrementalized array computations, with any additional information needed appropriately initialized and additional optimizations enabled cleanly performed. Each of these components is relatively simple, and the overall optimization algorithm is modular. Both analytical and experimental results show drastic speedups that are not achievable by previous compiler optimizations.

Methods of explicit incrementalization [47], cache-and-prune [46], and use of auxiliary information [45] were first formulated for a functional language [43]. They have been adopted for loop incrementalization of imperative programs with no arrays, generalizing traditional strength reduction [44]. This paper extends that work to handle imperative programs that use arrays. It presents a broad generalization of strength reduction from arithmetics to aggregates in common high-level languages, such as FORTRAN, rather than to aggregates in special very-high-level languages, such as SETL [24, 25, 53, 54]. The speedup obtained from incrementalizing aggregate computations can be enormous compared to what is offered by previous compiler optimizations. Changes in hardware design have reduced the importance of strength reduction on arithmetic operations, but the ability to incrementalize aggregate computations remains essential, only more so when programmers are encouraged to write straightforward programs to facilitate reasoning about them.

The large body of work on high performance computing has dealt with computation-intensive applications, especially on arrays, via parallelizing compilers. Our method demonstrates a powerful alternative that is both orthogonal and correlated. It is orthogonal, since it speeds up computations running on a single processor, whether that processor is running alone or in parallel with others. It is correlated, since our optimization either allows subsequent parallelization to achieve greater speedup, or achieves the same or greater speedup than parallelization would while using fewer processors. In the latter case, resource requirements and communication costs are substantially reduced. Additionally, for this powerful optimization, we make use of techniques and tools for array dependence analysis [20, 21, 48, 50, 57, 58, 59, 60] and source-to-source transformation [7, 39, 50, 56, 62] that were developed for parallelizing compilers.

This paper is organized as follows. Section 2 gives the programming language. Sections 3 describes how to identify and incrementalize aggregate array computations and form incrementalized loops. Section 4 describes how to maintain additional information to facilitate incrementalization. Section 5 presents the overall algorithm and discusses relevant issues. Section 6 gives examples with performance figures. Section 7 discusses related work.

2 Language

This paper considers an imperative language whose data types include multi-dimensional arrays. The language has variables that can be array references ($a[j_1, ..., j_m]$). It has the usual primitive arithmetic and Boolean operations, assignment ($v := e$), sequencing ($s_1; s_2$), conditionals (**if** e **then** s_1 **else** s_2), and loops (**for** $v := e_1$ **to** e_2 **do** s). To reduce clutter, we use indentation to indicate syntactic scopes and omit **begin** and **end**. We use the following program as a running example.

Example 2.1 Given an n_1-by-n_2 array a, the following code computes, for each i in the n_1-dimension, the sum of the m-by-n_2 rectangle starting at position i. It takes $O(n_1 n_2 m)$ time.

```
for $i := 0$ to $n_1 - m$ do
  $s[i] := 0;$
  for $k := 0$ to $m - 1$ do                (3)
    for $l := 0$ to $n_2 - 1$ do
      $s[i] := s[i] + a[i+k, l]$
```

Our primary goal is to reduce the running time. Of course, maintaining additional values takes extra space. Our secondary goal is to reduce the space consumption. We use $a[.i.]$ to denote a reference of array a that contains i in a subscript. We use $t[x := e]$ to denote t with each occurrence of x replaced with e.

3 Incrementalizing aggregate array computations

We first show how to identify aggregate array computations and determine how the parameters they depend on are updated. We then show how to incrementalize aggregate array computations with respect to given updates, by exploiting properties of the functions involved. Finally, we describe how to transform a loop with aggregate array computations in the loop body into a new loop with incrementalized aggregate array computations in the loop body.

3.1 Identifying candidates

Candidates for optimizations are in nested loops, where inner loops compute accumulated quantities over array elements and outer loops update the subscripts used by the array references.

Definition 3.1 *An* aggregate array computation (AAC) *is a loop that computes accumulated quantities over array elements. The canonical form of an AAC is*

$$\textbf{for } i := e_1 \textbf{ to } e_2 \textbf{ do } \; v := f(v, g(a[.i.], \ldots)) \tag{4}$$

where e_1 and e_2 are expressions, f is a function of two arguments, g is a function of one or more arguments, and v is a variable (which may be an array reference) that does not contain i or a. If v is not an array reference, then v must not occur in the arguments of g; if v is an array reference $s[j_1, \ldots, j_m]$, then s must not occur in the arguments of g or in $j_1, \ldots, j_m$. An initial assignment to v may be included in an AAC.

The existence of four items in the loop body identify such a computation. First, an *accumulating variable*—v—a variable that holds the accumulated quantity. This variable may itself be an array reference whose subscripts depend only on variables defined outside the loop. Second, a *contributing array reference*—$a[.i.]$—an array reference whose subscripts depend on the loop variable. Third, a *contributing function*—g—a function that computes using the contributing array references but not the accumulating variable. Fourth, an *accumulating function*—f—a function that updates the accumulating variable using the result of the contributing function.

More generally, an AAC may contain multiple **for** clauses, multiple assignments, and multiple array references. We use the above form only to avoid clutter in the exposition of the algorithms. Extending the algorithms to allow these additional possibilities is straightforward, and these extensions are assumed in the examples.

The essential feature of an AAC A is that a set of computations are performed by the contributing function, and their results are accumulated by the accumulating function. We characterize this set by the *contributing set* $S(A)$, which describes the ranges of the subscripts of the contributing array references. For an AAC of the form (4), the contributing set is $S(A) = \{\langle a[.i.] \rangle \mid e_1 \le i \le e_2\}$. More generally, for an AAC A with contributing array references $a_1, \ldots, a_k$,

$$S(A) = \{\langle a_1, \ldots, a_k \rangle \mid R\}, \tag{5}$$

where R is the conjunction of the constraints defined by the ranges of all the loop variables in A.

Example 3.1 For the program (3), the loop on k and the loop on l each forms an AAC; we denote them as A_k and A_l, respectively. For both of them, $s[i]$ is the accumulating variable, $a[i+k,l]$ is the contributing array reference, $g(u)=u$, and $f(v,u)=(v+u)$. The contributing sets are

$$S(A_k)=\{\langle a[i+k,l]\rangle \mid 0\le k<m \ \wedge\ 0\le l<n_2\} \qquad S(A_l)=\{\langle a[i+k,l]\rangle \mid 0\le l<n_2\}.$$

Definition 3.2 *A* parameter *of an AAC A is a variable used in A but defined outside A. A* subscript update operation (SUO) *for A is a redefinition of a parameter that, if it appears at all in the arguments of the contributing function, appears there only in the subscripts of the contributing array references of A. A SUO for a parameter w is denoted $\oplus_w$; in contexts where it is irrelevant to the discussion which parameter is being considered, we simply write $\oplus$.*

Parameters of AACs can be identified using def-use chains [1]. Redefinitions of parameters that satisfy the above are SUOs.

The heart of our approach is incrementalization of an AAC with respect to a SUO. For simplicity of exposition, we consider in this paper only updates to parameters that are themselves loop variables of loops enclosing the AAC. Since we omitted specification of step size from **for** loops, it is implied that the update operation for each parameter is the operation "increment by 1". We adopt these restrictions here only to simplify and condense the presentation. It is straightforward to consider updates in a more general way.

Example 3.2 For A_k in program (3), variables i,m,n_2 are its parameters, and the update $\oplus_i$ is a SUO. For A_l in program (3), i,k,n_2 are its parameters, and the updates $\oplus_i$ and $\oplus_k$ are SUOs.

An AAC A and a SUO $\oplus_w$ together form a problem of incrementalization. We use $A^{\oplus_w}$ to denote A with parameter w symbolically updated by $\oplus_w$. For example, if A is of the form (4), w is the loop variable of a loop enclosing A, and the update operation is "increment by 1", then $A^{\oplus_w}$ is

$$\textbf{for } i:=e_1^{\oplus_w} \textbf{ to } e_2^{\oplus_w} \textbf{ do } v^{\oplus_w}:=f(v,g(a[.i.],\ldots))^{\oplus_w} \tag{6}$$

where for any t, $t^{\oplus_w}$ abbreviates $t[w:=w+1]$.

The goal of incrementalization is to transform $A^{\oplus}$ so that it is computed efficiently by updating the result of A rather than by computing from scratch.

Algorithm 3.1 (Identifying candidates in a given loop)
For each loop A contained in the body of the given loop L, do the following.
1. Identify the accumulating variable, contributing array references, contributing function, and accumulating function (if the loop body of A contains multiple assignments, merge their functionalities to obtain the contributing and accumulating functions). If any of these are absent, then A is not an AAC, so do not consider it further.
2. Let w denote the loop variable of L. If w is a parameter of A that, if it appears at all in the arguments of the contributing function, appears there only in the subscripts of the contributing array references of A, then $\oplus_w$ is a SUO for A, so A and $\oplus_w$ form a problem of incrementalization.

3.2 Incrementalization

Incrementalization aims to perform an AAC A incrementally as its parameters are updated by a SUO $\oplus$. The basic idea is to replace with corresponding retrievals, whenever possible, subcomputations of $A^{\oplus}$ that are also performed in A and whose values can be retrieved from the saved results of A. To consider the effect of a SUO on an AAC we consider (i) the ranges of the subscripts of the array references on which the contributing function is computed and (ii) the algebraic properties of the accumulating function. These two aspects correspond to the following two steps.

The first step computes the differences between the contributing sets of A and $A^{\oplus}$. These differences are denoted

$$decS(A, \oplus) = S(A) - S(A^{\oplus}) \qquad incS(A, \oplus) = S(A^{\oplus}) - S(A).$$

Note that $S(A^{\oplus})$ can be computed from $A^{\oplus}$ by definition of S, or it can be obtained from $S(A)$ by symbolically updating the parameter being considered using $\oplus$.

Example 3.3 For the program (3), consider incrementalization of A_k with respect to update of i. $A_k^{\oplus_i}$ is

```
$s[i+1] := 0;$
for $k := 0$ to $m-1$ do
  for $l := 0$ to $n_2 - 1$ do
    $s[i+1] := s[i+1] + a[i+1+k, l]$                    (7)
```

and its contributing set is

$$S(A_k^{\oplus_i}) = \{\langle a[i+1+k, l]\rangle \mid 0 \le k < m \land 0 \le l < n_2\}.$$

To compute the difference of two sets represented in the form (5), we formulate the difference as a single set of constraints and then use well-studied methods developed for array data dependence analysis [20, 21, 48, 50, 57, 58, 59, 60] to simplify those constraints. In particular, the methods and tools developed by Pugh *et al.* in the Omega project [57, 58, 59, 60] have been used to produce the desired output. This approach is embodied in the following algorithm.

Algorithm 3.2 (Difference of two contributing sets)
Input: $S_1 = \{\langle a_{11}, \ldots, a_{1k}\rangle \mid R_1\}$ and $S_2 = \{\langle a_{21}, \ldots, a_{2k}\rangle \mid R_2\}$, where each a_{ij} is an array reference, and R_i is a conjunction of range constraints.
Output: The set difference $S_1 - S_2$.

1. Let $\bar{u}$ be a tuple of all the constrained variables in S_2. Let $\bar{u}'$ be an equal-length tuple of fresh variables. Note that $S_2 = \{\langle a_{21}[\bar{u} := \bar{u}'], \ldots, a_{2k}[\bar{u} := \bar{u}']\rangle \mid R_2[\bar{u} := \bar{u}']\}$.
2. Let $S = \{\langle a_{11}, \ldots, a_{1k}\rangle \mid R_1 \land \neg(\exists \bar{u}' : (a_{11} = a_{21}[\bar{u} := \bar{u}'] \land \cdots \land a_{1k} = a_{2k}[\bar{u} := \bar{u}'] \land R_2[\bar{u} := \bar{u}'])\}$.
3. Simplify the constraints in S using the techniques studied by Pugh *et al.* [57, 58, 59, 60].

Example 3.4 For incrementalization of A_k with respect to $\oplus_i$ in the running example, the set differences are computed as follows:

$$\begin{aligned}
&incS(A_k, \oplus_i) = S(A_k^{\oplus_i}) - S(A_k)\\
&= \{\langle a[i+1+k, l]\rangle \mid (0 \le k < m \land 0 \le l < n_2) \land \neg(\exists \langle k', l'\rangle : i+1+k = i+k' \land l = l' \land 0 \le k' < m \land 0 \le l' < n_2)\}\\
&= \{\langle a[i+1+k, l]\rangle \mid k = m-1 \land 0 \le l < n_2\}\\
&= \{\langle a[i+m, l]\rangle \mid 0 \le l < n_2\}
\end{aligned}$$

$$\begin{aligned}
&decS(A_k, \oplus_i) = S(A_k) - S(A_k^{\oplus_i})\\
&= \{\langle a[i+k, l]\rangle \mid (0 \le k < m \land 0 \le l < n_2) \land \neg(\exists \langle k', l'\rangle : i+k = i+1+k' \land l = l' \land 0 \le k' < m \land 0 \le l' < n_2)\}\\
&= \{\langle a[i+k, l]\rangle \mid k = 0 \land 0 \le l < n_2\}\\
&= \{\langle a[i, l]\rangle \mid 0 \le l < n_2\}
\end{aligned}$$

The second step in incrementalization uses the properties of the accumulating function to determine how a new AAC can be performed efficiently by updating the result of the old AAC. The goal is to update the result of A by removing the contributions from $decS(A, \oplus)$ and inserting the contributions from $incS(A, \oplus)$. To remove contributions, the accumulating function f must have an inverse f^{-1} with respect to its second argument, *i.e.*, f^{-1} and f must satisfy $f^{-1}(f(v, c), c) = v$.

We say that a set of array references is *at the end* of A if it is empty or it contains array references corresponding to a suffix of the iterations by A; for example, $incS(A_k, \oplus_i)$ is at the end of $A_k^{\oplus_i}$, but $decS(A_k, \oplus_i)$ is not at the end of A_k. If $decS(A, \oplus)$ is not at the end of A or $incS(A, \oplus)$ is not at the end of $A^{\oplus}$, then we must also require that f be associative and commutative.

If these requirements are satisfied, $A^{\oplus}$ of the form (6) can be transformed into an incrementalized version of the form

```
$v^{\oplus} := v$;
for $i := last(decS(A, \oplus))$ downto $first(decS(A, \oplus))$ do $v^{\oplus} := f^{-1}(v^{\oplus}, g(a[.i.], \ldots))$;        (8)
for $i := first(incS(A, \oplus))$ to $last(incS(A, \oplus))$ do $v^{\oplus} := f(v^{\oplus}, g(a[.i.], \ldots))$
```

where v contains the result of the previous execution of the AAC, and i is a re-use of the loop variable of the outermost loop in A. If f is not associative or not commutative, in which case $decS(A, \oplus)$ must be at the end of A, then the contributions from the elements of $decS(A, \oplus)$ must be removed from v in the opposite order from which they were added; this is why **downto** is used in (8).

The structure of the code in (8) is schematic; the actual loop structure needed to iterate over $decS(A, \oplus)$ and $incS(A, \oplus)$ depends on the form of the simplified constraints in them, which depends on the ranges of the loops in A and on subscripts in the contributing references. In particular, if all of these are linear expressions over the loop variables of the loops in A, then the constraints in $decS(A, \oplus)$ and $incS(A, \oplus)$ can be simplified into a set of inequalities giving upper and lower bounds on constrained variables; these inequalities are easily converted into loops that iterate over $decS(A, \oplus)$ and $incS(A, \oplus)$, using Omega's code generation facility. When the size of the set $decS(A, \oplus)$ or $incS(A, \oplus)$ is zero, the corresponding **for** loop can be omitted; when the size is a small constant, the corresponding **for** loop can be unrolled.

Example 3.5 For the running example, to incrementalize $A_k^{\oplus_i}$ in (7). Since $+$ has an inverse $-$, and since $+$ is associative and commutative, we obtain the following incrementalized AAC:

```
$s[i+1] := s[i]$;
for $l := 0$ to $n_2 - 1$ do
   $s[i+1] := s[i+1] - a[i, l]$;                (9)
for $l := 0$ to $n_2 - 1$ do
   $s[i+1] := s[i+1] + a[i+m, l]$
```

The transformation from (6) to (8) is worthwhile only if the total cost of (8) is not larger than the total cost of (6). The costs of f and f^{-1} and the sizes of the contributing sets together provide good estimates of the total costs.

First, consider the asymptotic time complexity. If f^{-1} is asymptotically at least as fast as f, and $|decS(A, \oplus)| + |incS(A, \oplus)|$ is asymptotically less than $|S(A^{\oplus})|$ (these quantities are all functions of the size of the input), then the transformed program is asymptotically faster than the original program. For the running example, this condition holds, since f and f^{-1} are both constant-time, $|decS(A_k, \oplus_i)| + |incS(A_k, \oplus_i)|$ is $O(n_2)$, and $|S(A_k^{\oplus_i})|$ is $O(n_2 m)$.

Asymptotic time complexity is an important but coarse metric; statements about absolute running time are also possible. If f^{-1} is at least as fast as f in an absolute sense, and if $|decS(A, \oplus)| + |incS(A, \oplus)|$ is less than $|S(A^{\oplus})|$, then the transformed program (8) is faster than the original program (6) in an absolute sense. For the running example, this condition holds when $m > 2$. This speedup is supported by our experimental results.

Theorem 3.1 *The transformation from (6) to (8) is correct, and if the above conditions on (asymptotic) time complexity hold, then the transformed program is (asymptotically) faster than the original program.*

Algorithm 3.3 (Incrementalization of A with respect to $\oplus_w$)
1. Obtain $A^{\oplus_w}$ from A by symbolically updating w.
2. Compute the contributing sets $S(A)$ and $S(A^{\oplus_w})$.
3. Compute $decS(A, \oplus_w)$ and $incS(A, \oplus_w)$, using Algorithm 3.2.
4. If (i) $decS(A, \oplus_w) = \emptyset$, or f has an inverse, (ii) $decS(A, \oplus_w)$ is at the end of A and $incS(A, \oplus_w)$ is at the end of $A^{\oplus}$, or f is associative and commutative, and (iii) the condition on complexity holds, then construct an incremental version of $A^{\oplus_w}$, of the form (8).

3.3 Forming incrementalized loops

To use incrementalized AACs, we transform the original loop. The basic idea is to unroll the first iteration of the original loop to form the initialization and, for the remaining iterations, replace AACs with their corresponding incremental versions. While incrementalized AACs are formulated to compute values of the next iteration based on values of the current iteration, we use them to compute values of the current iteration based on values of the previous iteration. This is straightforward for any **for** loop. For the particular SUO $\oplus_w$ that is "increment by 1", we just symbolically decrement w in the incremental version by 1. Thus, we replace w by $w-1$ and replace $w+1$ by w.

Example 3.6 For the running example, using the incrementalized AAC in (9), we obtain the following program, which takes $O(n_1 n_2)$ time and no additional space.

```
   init using  [  $s[0] := 0;$
  $A_k$ in (3) |  for $k := 0$ to $m-1$ do
   with $i=0$  |     for $l := 0$ to $n_2 - 1$ do
               [        $s[0] := s[0] + a[k,l]$
    for clause     for $i := 1$ to $n_1 - m$ do
               [     $s[i] := s[i-1]$;                      (10)
     inc using |     for $l := 0$ to $n_2 - 1$ do
      code (9) |        $s[i] := s[i] - a[i-1, l]$;
  with $i$ dec 1 |     for $l := 0$ to $n_2 - 1$ do
               [        $s[i] := s[i] + a[i-1+m, l]$
```

4 Maintaining additional information

The above incrementalization is possible only when the results of AACs are stored in the program, so that they are available from one iteration to the next. Such results, if not stored already, and, often, additional information need to be maintained for efficient incremental computation [45, 46]. Such information often comes from *intermediate results* computed in the middle of the original computation [46]. It may also come from *auxiliary information* that is not computed at all in the original computation [45]. The central issues are how to find, use, and maintain appropriate information.

General methods have been proposed and formulated for a functional language [43, 45, 46]. Here we apply them to AACs, using a variant of the *cache-and-prune* method [46]. We proceed in three stages: (I) transform the code for AACs to store all intermediate results and related auxiliary information not stored already, (II) incrementalize the resulting AACs from one iteration to the next based on the results stored in Stage I, and (III) prune out stored values that were not useful in the incrementalization in Stage II.

4.1 Stage I: Caching results of all AACs

We consider saving and using results of all AACs. This allows much greater speedup than saving and using results of primitive operations.

After every AAC, we save in fresh variables the intermediate results that are not saved already. Since we consider AACs that are themselves performed inside loops, we must distinguish intermediate results obtained after different iterations. To this end, for each AAC A, we introduce a fresh array variable subscripted with the loop variables of all loops enclosing A; immediately after A, we add an assignment that stores the value computed by A in the corresponding element of the fresh array.

Example 4.1 In the program (3), to save intermediate results computed by A_l, we introduce an array s_1 subscripted by i and k, and immediately after the loop on l, we copy the value of $s[i]$ into $s_1[i,k]$.

```
$s[i] := 0;$
for $k := 0$ to $m-1$ do
    for $l := 0$ to $n_2 - 1$ do
        $s[i] := s[i] + a[i+k, l];$
    $s_1[i,k] := s[i]$
```

A related class of auxiliary information can be obtained to facilitate incrementalization if the accumulating function f is associative and has a zero element 0 (*i.e.*, $f(v,0) = f(0,v) = v$). In this case, we save in a fresh array values of the AAC starting from 0, rather than from any other previous result, and we accumulate these values into the original accumulating variable immediately after the AAC.

Example 4.2 For the program (3), storing such auxiliary information yields a program that, for each value of k, accumulates separately in $s_1[i,k]$ the sum computed by A_l starting from 0, and then accumulates that sum into the accumulating variable $s[i]$:

```
$s[i] := 0;$
for $k := 0$ to $m-1$ do
    $s_1[i,k] := 0;$
    for $l := 1$ to $n_2$ do                              (11)
        $s_1[i,k] := s_1[i,k] + a[i+k, l];$
    $s[i] := s[i] + s_1[i,k]$
```

Essentially, this class of auxiliary information is obtained by chopping intermediate results into independent pieces based on the associativity of g. These values are not computed at all in the original program and thus are called *auxiliary information.* This auxiliary information helps reduce the analysis effort in later stages, since the value of an aggregate computation is directly maintained rather than being computed as the difference of two subsequent intermediate results of the larger computation.

Another optimization at this stage that helps simplify analyses in the later stages and reduce the space consumed by the additional information is to avoid generation of redundant subscripts for the fresh arrays. Redundancies arise when the same value is computed in multiple iterations and therefore stored in multiple entries in that array. To detect such redundancies, we proceed as follows. Let $\bar{w}$ be the subscript vector of the fresh array for an AAC A, *i.e.*, $\bar{w}$ is the tuple of the loop variables of all loops enclosing A. We define two tuples $\bar{w}_1$ and $\bar{w}_2$ to be *equivalent for A* (denoted $\equiv_A$) if they lead to the same contributing set, hence to the same auxiliary information, *i.e.*, $\bar{w}_1 \equiv_A \bar{w}_2$ iff $S(A)[\bar{w} := w_1] = S(A)[\bar{w} := \bar{w}_2]$.

Example 4.3 For A_l in (11), we have

$$\begin{aligned} \langle i_1, k_1 \rangle \equiv_{A_l} \langle i_2, k_2 \rangle \quad & \text{iff} \quad (\{\langle a[i_1 + k_1, l] \rangle \mid 0 \le l < n_2\} = \{\langle a[i_2 + k_2, l] \rangle \mid 0 \le l < n_2\}) \\ & \text{iff} \quad (i_1 + k_1 = i_2 + k_2). \end{aligned} \tag{12}$$

To avoid saving the same value in multiple entries in the array, we choose a canonical element of each equivalence class, and replace accesses to equivalent subscripts with accesses to that canonical element. This makes the array sparser, and, if the canonical elements are chosen so that some block of the array is unused, then we can save space by eliminating that part of the array. Specifically, we exploit this equivalence by observing that the simplified expression for $\equiv_A$ is always of the form

$$\bar{w}_1 \equiv_A \bar{w}_2 \quad \text{iff} \quad (e_1[\bar{w} := \bar{w}_1] = e_1[\bar{w} := \bar{w}_2]) \wedge \cdots \wedge (e_h[\bar{w} := \bar{w}_1] = e_h[\bar{w} := \bar{w}_2]) \tag{13}$$

for some expressions $e_1, \ldots, e_h$. This implies that the values of $e_1, \ldots, e_h$ together distinguish the equivalence classes of $\equiv_A$, so we can take the fresh array to be h-dimensional and use $e_1, \ldots, e_h$ as its subscripts.

Example 4.4 For A_l in (11), the equivalence $\equiv_{A_l}$ in (12) is of the form (13) with $h = 1$ and $e_1 = i + k$, so we take s_1 to be a 1-dimensional array with subscript $i + k$, obtaining the extended AAC

```
$s[i] := 0;$
for $k := 0$ to $m - 1$ do
    $s_1[i+k] := 0;$
    for $l := 0$ to $n_2 - 1$ do                                  (14)
        $s_1[i+k] := s_1[i+k] + a[i+k, l];$
    $s[i] := s[i] + s_1[i+k]$
```

The auxiliary information now occupies $O(n_1 + m)$ space, compared to $O(n_1 m)$ space in (11).

Algorithm 4.1 (Caching additional information for an AAC A)

1. If the result of A is available in some variable, then don't maintain any additional information for A. Otherwise, do Step 2 or 3.
2. If the accumulating function is not associative, then introduce a fresh array variable u, subscript u with the loop variables of all the loops enclosing A, and add an assignment after A that copies the value of the accumulating variable into u.
3. If the accumulating function is associative, then introduce a fresh variable u, avoid redundant subscripts using the method described above, add an assignment before A to initialize u (with appropriate subscripts) to zero, accumulate values computed in A into u instead of the original accumulating variable, and add an assignment after A that accumulates the value of u into the original accumulating variable.

4.2 Stage II: Incrementalization

Stage II incrementalizes the AACs from one iteration to the next based on the results stored by Stage I. Section 3.2 describes how to incrementalize an AAC with respect to a SUO. In general, we want to perform all AACs in an iteration efficiently using all stored results of the previous iteration. As a basic case, we avoid performing AACs whose values have been computed completely in the previous iteration. This can be done by keeping track of all the AACs and the variables that store their values. We incrementalize other AACs using the algorithms in Section 3.2.

Example 4.5 Incrementalize the AACs A_k and A_l in (14) with respect to $\oplus_i$. First, we avoid performing AACs that have been performed in the previous iteration. Thus, we only need to compute A_l for elements in the set difference $\{s_1[i+1+k] \mid 0 \leq k \leq m-1\} - \{s_1[i+k] \mid 0 \leq k \leq m\text{–}1\} = \{s_1[i\text{+}1\text{+}k] \mid k = m\text{–}1] = \{s_1[i\text{+}m]\}$. Then, we incrementalize A_k with respect to $\oplus_i$. We have $decS(A_k, \oplus_i) = \{s_1[i\text{+}k] \mid k = 0\} = \{s_1[i]\}$ and $incS(A_k, \oplus_i) = \{s_1[i\text{+}1\text{+}k] \mid k = m\text{–}1\} = \{s_1[i\text{+}m]\}$. These sets both have size 1, so we unroll the loops over them and obtain the incrementalized AAC

```
$s_1[i+m] := 0$;
**for** $l := 0$ **to** $n_2 - 1$ **do**
  $s_1[i+m] := s_1[i+m] + a[i+m, l]$;                    (15)
$s[i+1] := s[i] - s_1[i] + s_1[i+m]$
```

Algorithm 4.2 (Incrementalization)
1. Compute the set of AACs in the new iteration that are not computed in the previous iteration.
2. Incrementalize them with respect to the SUO using Algorithm 3.3.

4.3 Stage III: Pruning

Some of the additional information saved in Stage I might not be useful for the incrementalization in Stage II. Stage III analyzes dependencies in the incrementalized computation and prunes useless information in both the AACs that store all the additional information and the incrementalized AACs that maintain all the additional information.

The analysis starts with the uses of such information in computing the original accumulating variables and follows dependencies back to the definitions of such information. The dependencies are transitive [46] and can be used to compute all the information that is useful. Pruning then eliminates useless information, saving both space and time. Pruning of unused intermediate results presents no special difficulties. If the accumulating function is associative, and if auxiliary information was added and turns out to be useless, then associativity of the accumulating function will be needed to justify elimination of that information, since the caching transformation integrates it into the original computation.

After pruning, we obtain AACs that store only useful additional information and incrementalized AACs that use and maintain only the useful additional information.

4.4 Forming incrementalized loops

The incrementalized loop is formed as in Section 3.3, except that the useful additional information needs to be initialized and maintained. In particular, in the unrolled first iteration, we replace the original AACs with AACs that store useful additional information, and in the rest of the iterations, we replace the original AACs with incrementalized AACs that maintain the useful additional information.

Example 4.6 From the code in (14) and its incremental version in (15) that together compute and maintain useful additional information, we obtain the optimized program below that takes $O(n_1 n_2)$ time and $O(n_1)$ additional space. Compared with the program in (10), this program eliminates a constant factor of 2 in the execution time, and thus is twice as fast. Our experimental results also support this speedup.

```
init           ┌  $s[0] := 0;$
using          │  **for** $k := 0$ **to** $m-1$ **do**
(14)           │    $s_1[k] := 0;$
with           │    **for** $l := 0$ **to** $n_2-1$ **do**
$i = 0$        │      $s_1[k] := s_1[k] + a[k,l];$
               └    $s[0] := s[0] + s_1[k];$                              (16)
**for** clause    **for** $i := 1$ **to** $n_1 - m$ **do**
               ┌    $s_1[i-1+m] := 0;$
inc using      │    **for** $l := 0$ **to** $n_2-1$ **do**
code (15)      │      $s_1[i-1+m] := s_1[i-1+m] + a[i-1+m,l];$
with $i$ dec 1 └    $s[i] := s[i-1] - s_1[i-1] + s_1[i-1+m]$
```

Further optimizations to the resulting loop may be enabled by the incrementalization, as described in [44]; these optimizations include folding initialization, replacing termination test, and minimizing information maintained in the loop. While it is the explicit incrementalization, which often relies on exploiting additional information, that can give drastic speedup, further optimizations may reduce space consumption and code size.

5 The optimization algorithm

The overall optimization algorithm aims to incrementalize aggregate array computations in every loop of a program. For nested loops, it considers them from inner to outer.

Algorithm 5.1 (Optimization to eliminate overlapping aggregate array redundancies) Consider nested loops from inner to outer and, for each loop L encountered, perform Steps 1-5.

1. Let w denote the loop variable of L. Identify all loops in the loop body of L that are AACs and for which $\oplus_w$ is a SUO, using Algorithm 3.1.
2. Extend these AACs to save all appropriate additional information in variables, if not saved already, using Algorithm 4.1.
3. Incrementalize these AACs with respect to $\oplus_w$, using Algorithm 4.2. If any of these AACs are nested, consider them from inner to outer.
4. Prune additional information that is not useful for the incrementalization.
5. If incrementalization is performed, then form incrementalized loops using incrementalized AACs, as described in Section 4.4.

The optimization algorithm we just described is expensive but automatic. A number of optimizations can be made to this algorithm. For example, Step 1 needs to consider only AACs whose contributing array references depend on the current loop variable. For another example, since we consider nested loops from inner to outer, Step 2 only needs to consider saving results of AACs outside of loops considered already.

Our optimization can achieve drastic program speedup, often by exploiting appropriate additional information. We have described conditions that guarantee the running time improvement by the resulting programs. The additional space consumption may look worrisome, since it may affect cache performance for large data sets. While our optimization eliminates redundant computation, it also eliminates redundant data access, so it generally preserves or increases cache locality. In general, however, more rigorous study is needed for analysis of space as well as various trade-offs.

We have also established the correctness of our optimizations. It is based on the usage of an exact inverse function f^{-1} when $decS(A, \oplus) \neq \emptyset$. Inverse function f^{-1} is always exact for integer computations. For floating-point computations, even if the inverse is exact in principle, it might be computed only approximately. In such cases, the optimized program may produce less accurate results than the original program. Further study is needed to quantify and mitigate this effect.

6 Examples and performance results

The following examples and performance results show that our method achieves more substantial optimizations than otherwise possible. The figures are obtained from running the original and the corresponding optimized programs, coded in FORTRAN, on a dedicated SPARCstation 4. The programs were compiled using Sun Microsystems' f77 compiler, with optimization flags -O4 and -fast.

6.1 Partial sum

Partial sum is a simple but interesting and illustrative example. Given an array $a[1..n]$ of numbers, for each index i (line [1]), compute the sum of elements 1 to i (lines [2] to [4]). The straightforward program (17) takes $O(n^2)$ time.

```
[1]  for $i := 1$ to $n$ do
[2]      $s[i] := 0$;
[3]      for $j := 1$ to $i$ do                    (17)
[4]          $s[i] := s[i] + a[j]$
```

It can be optimized using our algorithm. First, consider the inner loop. Its loop body does not contain any AACs. Now, consider the outer loop. Step 1. Its loop body contains an AAC A_j, where $s[i]$ is the accumulating variable, and its loop increment is a SUO $\oplus_i$. Step 2. No additional values need to be saved. Step 3. $decS(A, \oplus_i) = \emptyset$ and $incS(A, \oplus_i) = \{\langle a[i+1] \rangle\}$. Thus, the computation of $s[i+1]$ is incrementalized by accumulating to the value of $s[i]$ the only contribution $a[i+1]$. We obtain $s[i+1] := s[i] + a[i+1]$. Step 4. Pruning leaves the code unchanged. Step 5. Initializing $s[1]$ to $a[1]$ and forming the rest of the loop for $i = 2..n$, we obtain the program (18).

```
$s[1] := a[1]$;
for $i := 2$ to $n$ do                    (18)
    $s[i] := s[i-1] + a[i]$
```

This program takes only $O(n)$ time. Running times for programs (17) and (18) are plotted in Figure 2; the rate of increase of the running time of the optimized program is extremely small.

When code is written as in (17), previous techniques can recognize that each inner loop is independent. On a single processor machine, these techniques result in better pipelining but still leave the code running in quadratic time. On a parallel machine with n processors, iterations of the inner loop can be computed completely in parallel, and the program uses only linear time, but due to the additional communication cost, the result would still come out slower than with our optimized sequential code. An additional advantage of our optimized code is that parallelizing compilers can more easily recognize that it is the prefix sums problem and, on a parallel machine with n processors, compute the result in only $O(\log n)$ time.

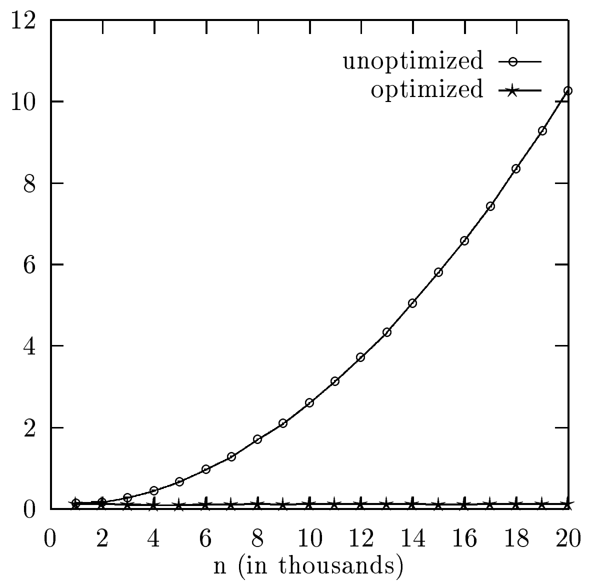

Figure 2: Running time (in seconds) for partial sums problem.

6.2 Local neighborhood problems

This problem was introduced in Section 1. We show that applying our optimization algorithm to the straightforward program (1) yields the efficient program (2) with appropriate initializations of the array margins.

First, consider the innermost loop L_l on l. There is no AAC in its body.

Next, consider the loop L_k on k. Its loop body L_l is an AAC A_l, and its loop increment is a SUO $\oplus_k$. Array analysis yields $decS(A_l, \oplus_k) = S(A_l)$ and $incS(A_l, \oplus_k) = S(A_l^{\oplus_k})$, so incrementalization is not worthwhile. The algorithm leaves the code unchanged.

Next, consider the loop L_j on j. Step 1. Its loop body contains two AACs, A_l and A_k, and its loop increment is a SUO $\oplus_j$. Step 2. Since the accumulating function $+$ is associative, saving the values of A_l in an array b yields a new loop body

```
$sum[i,j] := 0;$
for $k := 0$ to $m-1$ do
    $b[i+k,j] := 0;$
    for $l := 0$ to $m-1$ do
        $b[i+k,j] := b[i+k,j] + a[i+k,j+l];$
    $sum[i,j] := sum[i,j] + b[i+k,j]$                                (19)
```

Step 3. Incrementalizing A_l in the body of the loop on k with respect to $\oplus_j$, we have $decS(A_l, \oplus_j) = \{\langle a[i+k, j+l]\rangle \mid l = 0\} = \{\langle a[i+k, j]\rangle\}$ and $incS(A_l, \oplus_j) = \{\langle a[i+k, j+1+l]\rangle \mid l = m-1\} = \{\langle a[i+k, j+m]\rangle\}$. Incrementalizing A_k with respect to $\oplus_j$, we have $decS(A_k, \oplus_j) = S(A_k)$ and $incS(A_k, \oplus_j) = S(A_k^{\oplus_j})$, so incrementalization is not worthwhile. We obtain

```
$sum[i,j+1] := 0;$
for $k := 0$ to $m-1$ do
    $b[i+k,j+1] := b[i+k,j] - a[i+k,j] + a[i+k,j+m];$          (20)
    $sum[i,j+1] := sum[i,j+1] + b[i+k,j+1]$
```

Step 4. Pruning (20) leaves the code unchanged. Step 5. Initialize using (19) with $j = 0$ and form loop for $j = 1..n{-}m$ using (20) as loop body. We obtain

```
         init   $sum[i,0] := 0$;
        using   for $k := 0$ to $m{-}1$ do
         (19)       $b[i{+}k,0] := 0$;
         with       for $l := 0$ to $m{-}1$ do
        $j=0$           $b[i{+}k,0] := b[i{+}k,0] + a[i{+}k,l]$;
                    $sum[i,0] := sum[i,0] + b[i{+}k,0]$;                                (21)
   for clause   for $j := 1$ to $n{-}m$ do
                    $sum[i,j] := 0$;
    inc using       for $k := 0$ to $m{-}1$ do
    code (20)           $b[i{+}k,j] := b[i{+}k,j{-}1] - a[i{+}k,j{-}1] + a[i{+}k,j{-}1{+}m]$;
 with $j$ dec 1         $sum[i,j] := sum[i,j] + b[i{+}k,j]$
```

Finally, consider the outermost loop L_i. Step 1. Its loop body is now (21); the first half contains AACs A_k and A_l, and the second half contains, in the body of the loop on j, incrementalized AAC of $b[i+k,j]$ and AAC $A_{k'}$ of $sum[i,j]$ by the loop over k. Its loop increment is a SUO $\oplus_i$. Step 2. No additional values need to be saved. Step 3. Incrementalize AACs in (21) with respect to $\oplus_i$. In the first half, only A_l in $\{b[i{+}1{+}k,0] \mid k = m{-}1\} = \{b[i{+}m,0]\}$ needs to be computed; also, $decS(A_k,\oplus_i) = \{\langle b[i{+}k,0]\rangle \mid k = 0\} = \{\langle b[i,0]\rangle\}$ and $incS(A_k,\oplus_i) = \{\langle b[i{+}1{+}k,0]\rangle \mid k = m{-}1\} = \{\langle b[i{+}m,0]\rangle\}$. In the second half, in the body of the loop on j, only $A_{k'}$ in $\{b[i{+}1{+}k,j] \mid k = m{-}1\} = \{b[i{+}m,j]\}$ needs to be computed; also, $decS(A_j,\oplus_i) = \{\langle b[i{+}k,j]\rangle \mid k = 0\} = \{\langle b[i,j]\rangle\}$ and $incS(A_j,\oplus_i) = \{\langle b[i{+}1{+}k,j]\rangle \mid k = m{-}1\} = \{\langle b[i{+}m,j]\rangle\}$. We obtain

```
$b[i{+}m,0] := 0$;
for $l := 0$ to $m{-}1$ do
    $b[i{+}m,0] := b[i{+}m,0] + a[i{+}m,l]$;
$sum[i{+}1,0] := sum[i,0] - b[i,0] + b[i{+}m,0]$;                                (22)
for $j := 1$ to $n{-}m$ do
    $b[i{+}m,j] := b[i{+}m,j{-}1] - a[i{+}m,j{-}1] + a[i{+}m,j{-}1{+}m]$;
    $sum[i{+}1,j] := sum[i,j] - b[i,j] + b[i{+}m,j]$
```

Step 4. Pruning (22) leaves the code unchanged. Step 5. Initialize using (21) with $i = 0$ and form loop for $i = 1..n{-}m$ using (22) as loop body. We obtain the optimized code in (23). If we assume that the array margins are appropriately initialized, we can obtain the code in (2).

```
                $sum[0,0] := 0$;
                for $k := 0$ to $m{-}1$ do
                    $b[k,0] := 0$;
         init       for $l := 0$ to $m{-}1$ do
        using           $b[k,0] := b[k,0] + a[k,l]$;
         (21)       $sum[0,0] := sum[0,0] + b[k,0]$;
         with   for $j := 1$ to $n{-}m$ do
        $i=0$       $sum[0,j] := 0$;
                    for $k := 0$ to $m{-}1$ do
                        $b[k,j] := b[k,j{-}1] - a[k,j{-}1] + a[k,j{-}1{+}m]$;
                        $sum[0,j] := sum[0,j] + b[k,j]$;
   for clause   for $i := 1$ to $n{-}m$ do
                    $b[i{-}1{+}m,0] := 0$;
                    for $l := 0$ to $m{-}1$ do
    inc using           $b[i{-}1{+}m,0] := b[i{-}1{+}m,0] + a[i{-}1{+}m,l]$;
    code (22)       $sum[i,0] := sum[i{-}1,0] - b[i{-}1,0] + b[i{-}1{+}m,0]$;
 with $i$ dec 1     for $j := 1$ to $n{-}m$ do
                        $b[i{-}1{+}m,j] := b[i{-}1{+}m,j{-}1] - a[i{-}1{+}m,j{-}1] + a[i{-}1{+}m,j{-}1{+}m]$;
                        $sum[i,j] := sum[i{-}1,j] - b[i{-}1,j] + b[i{-}1{+}m,j]$
                                                                                 (23)
```

The cost analysis in both incrementalization steps (20) and (22) ensures that the transformations are worthwhile when $m > 2$, which is usually the case. Essentially, in the resulting code (23), only four $\pm$ operations are performed for each pixel, independent of m. Thus, the optimized code takes $O(n^2)$ time. Running times for programs (1) and (23) are shown in Figure 3. As expected, the running time for the optimized program is approximately independent of m.

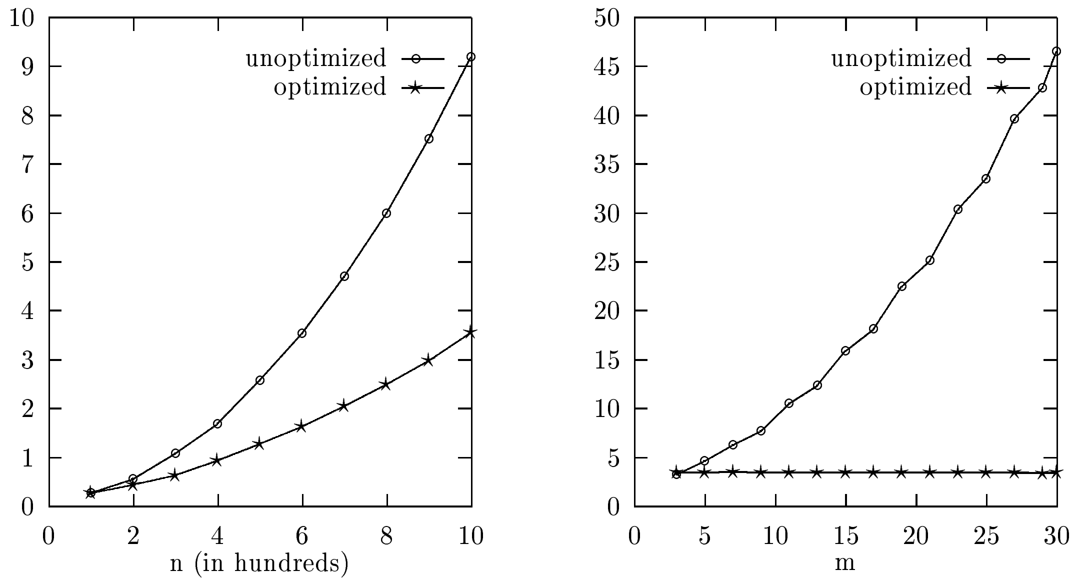

Figure 3: Running time (in seconds) for the local summation problem. For the graph on the left, $m = 10$. For the graph on the right, $n = 1000$.

7 Related work and conclusion

The basic idea of incrementalization is at least as old as Babbage's difference machine in the nineteenth century [30]. *Strength reduction* is the first realization of this idea in optimizing compilers [4, 14, 31, 32, 64]. The idea is to compute certain multiplications in loops incrementally using additions. Our work extends traditional strength reduction from arithmetic operations to aggregate array computations.

Finite differencing generalizes strength reduction to handle set operations in very-high-level languages like SETL [17, 24, 25, 51, 52, 54]. The idea is to replace aggregate operations on sets with incremental operations. Similar ideas are also used in the language INC [72], which allows programs to be written using operations on bags, rather than sets. Our work exploits the semantics underlying finite differencing to handle aggregate computations on arrays, which are more common in high-level languages and are more convenient for expressing many application problems.

APL compilers optimize aggregate array operations by performing computations in a piece-wise and on-demand fashion, avoiding unnecessary storage of large intermediate results in sequences of operations [26, 36, 49, 70]. The same basic idea underlies techniques such as *fusion* [3, 5, 13, 29, 66], *deforestation* [65], and transformation of *series expressions* [67, 68]. These optimizations do not aim to compute each piece of the aggregate operations incrementally using previous pieces and thus cannot produce as much speedup as our method can.

Specialization techniques, such as *data specialization* [35, 40], *run-time specialization and code generation* [15, 41, 42], and *dynamic compilation and code generation* [6, 18], have been used in program optimizations and achieved certain large speedups. These optimizations allow subcomputations repeated on fixed dynamic values to be computed once and reused in loops or recursions. Our optimization exploits subcomputations whose values can be efficiently updated, in addition to directly reused, from one iteration to the next. Thus, it allows far more speedup.

General *program transformations* [12, 38] can be used for optimization, as demonstrated in projects like CIP [8, 11, 55]. In contrast to such manual or semi-automatic approaches, our optimization of aggregate array computations can be automated and requires no user intervention or annotations. Our method for maintaining additional information is an automatic method for *strengthening loop invariants* [16, 33, 34, 61].

Our optimizations are based on the idea of explicit *incremental computation*, for which a general systematic transformational approach has been studied and formulated for a functional language [43, 45, 46, 47]. The idea has been used in optimizing imperative programs that do not use arrays [44]. The optimizations for arrays described here greatly extend the scope of that work, since arrays are widely used in so many application domains.

Directionals are unary operations, such as LEFT and UP, invented by Fisher and Highnam [22, 23, 37], to describe computations involving small numbers of neighboring nodes on grid structures. Such computations are optimized by directional rule-based transformations and common subexpression elimination, which essentially eliminate overlapping subcomputations. Their experiments show that the Cray Fortran compiler cannot perform these optimizations. Since local computations are written using directionals but no loops, their optimizations can potentially exploit more associativities than ours. However, they optimize only computations involving a statically determined number of neighbors, not other overlapping computations, such as those in the partial sum example. Also, programs must be written using directionals to take advantage of their optimizations; furthermore, it is inconvenient to write when more than a few neighbors are involved. Finally, they do not give general methods for handling grid margins.

Loop reordering [7, 39, 50, 56, 62], *pipelining* [2], and *array data dependence analysis* [20, 21, 48, 50, 57, 58, 59, 60] have been studied extensively for optimizing—in particular, parallelizing—array computations. While they aim to determine dependencies among uses of array elements, we further seek to determine exactly how subcomputations differ from one another. We reduce our analysis problem to symbolic simplification of constraints on loop variables and array subscripts, so methods and techniques developed for such simplifications for parallelizing compilers can be exploited. In particular, we have used tools developed by Pugh's group [57, 58, 59, 60]. Interestingly, ideas of incrementalization are used for optimizations in serializing parallel programs [10, 19].

In conclusion, this work describes a method and algorithms that allow more drastic optimizations of aggregate array computations than previous methods. Besides achieving optimizations not previously possible, our techniques fall out of one general approach, rather than simply being yet another new but *ad hoc* method. Future work includes implementation, faster optimization algorithms, and more general classes of aggregate computations.

Applying incrementalization to loop optimization on arrays will enable us to study important issues of cost, performance, and trade-offs of time, space, and locality more explicitly, precisely, and empirically than before. This is due to the large body of previously studied and implemented techniques and the availability of benchmarks for optimizing and parallelizing compilers.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[2] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):366–432, September 1995.

[3] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice Hall, 1971.

[4] F. E. Allen, J. Cocke, and K. Kennedy. Reduction of operator strength. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*, chapter 3, pages 79–101. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[5] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Rice University, 1983.

[6] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on PLDI*, pages 149–159, Philadelphia, Pennsylvania, May 1996.

[7] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Workshop on Advances in Languages and Compilers for Parallel Processing*, pages 192–219, August 1990.

[8] F. L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations—Computer-aided, intuition-guided programming. *IEEE Transactions on Software Engineering*, 15(2):165–180, February 1989.

[9] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, October 1984.

[10] M. Bromley, S. Heller, T. McNerney, and G. L. Steele Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the ACM SIGPLAN '91 Conference on PLDI*, pages 145–156, June 1991.

[11] M. Broy. Algebraic methods for program construction: The project CIP. In P. Pepper, editor, *Program Transformation and Programming Environments*, pages 199–222. Springer-Verlag, Berlin, 1984.

[12] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[13] W.-N. Chin. Safe fusion of functional expressions. In *Proceedings of the 1992 ACM Conference on LFP*, pages 11–20, June 1992.

[14] J. Cocke and K. Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11):850–856, November 1977.

[15] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the 23rd Annual ACM Symposium on POPL*, St. Petersburg Beach, Florida, January 1996.

[16] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[17] J. Earley. High level iterators and a method for automatically designing data structure representation. *Journal of Computer Languages*, 1:321–342, 1976.

[18] D. R. Engler. VCODE: A retragetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN '96 Conference on PLDI*, pages 160–170, Philadelphia, Pennsylvania, May 1996.

[19] M. D. Ernst. Serializing parallel programs by removing redundant computation. Master's thesis, MIT, August 1992, Revised August 1994.

[20] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, September 1988.

[21] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1), February 1991.

[22] A. L. Fisher and P. T. Highnam. Communication and code optimization in simd programs. In *International Conference on Parallel Processing*, August 1988.

[23] A. L. Fisher, J. Leon, and P. T. Highnam. Design and performance of an optimizing simd compiler. In *Frontiers of Massively Parallel Computation*, 1990.

[24] A. C. Fong. Inductively computable constructs in very high level languages. In *Conference Record of the 6th Annual ACM Symposium on POPL*, pages 21–28, San Antonio, Texas, January 1979.

[25] A. C. Fong and J. D. Ullman. Inductive variables in very high level languages. In *Conference Record of the 3rd Annual ACM Symposium on POPL*, pages 104–112, Atlanta, Georgia, January 1976.

[26] O. I. Franksen. *Mr. Babbage's Secret : The Tale of a Cypher and APL.* Prentice Hall, Englewood Cliffs, New Jersey, 1985.

[27] V. K. Garg. *Principles of Distributed Systems.* Kluwer, 1996.

[28] V. K. Garg and J. R. Mitchell. An efficient algorithm for detecting conjunctions of general global predicates. Technical Report TR-PDS-1996-005, University of Texas at Austin, 1996.

[29] A. Goldberg and R. Paige. Stream processing. In *Conference Record of the 1984 ACM Symposium on LFP*, pages 53–62, August 1984.

[30] H. H. Goldstine. Charles Babbage and his analytical engine. In *The Computer from Pascal to von Neumann*, chapter 2, pages 10–26. Princeton University Press, Princeton, New Jersey, 1972.

[31] A. A. Grau, U. Hill, and H. Langmaac. *Translation of ALGOL 60*, volume 1 of *Handbook for automatic computation*. Springer, Berlin, 1967.

[32] D. Gries. *Compiler Construction for Digital Computers*. John Wiley & Sons, New York, 1971.

[33] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[34] D. Gries. A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming*, 2:207–214, 1984.

[35] B. Guenter, T. B. Knoblock, and E. Ruf. Specializing shaders. In *Proceedings of ACM SIGGRAPH '95 (Computer Graphics Proceedings, Annual Conference Series)*, pages 343–349, 1996.

[36] L. Guibas and K. Wyatt. Compilation and delayed evaluation in APL. In *Conference Record of the 5th Annual ACM Symposium on POPL*, pages 1–8, January 1978.

[37] P. T. Highnam. *Systems and Programming Issues in the Design and Use of a SIMD Linear Array for Image Processing*. PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, April 1991. Available as technical report CMU-CS-91-136.

[38] S. Katz. Program optimization using invariants. *IEEE Transactions on Software Engineering*, SE-4(5):378–389, November 1978.

[39] W. Kelly and W. Pugh. Finding legal reordering transformations using mappings. In *Proceedings of the 7th Annual Workshop on Programming Languages and Compilers for Parallel Computing*, volume 892 of *Lecture Notes in Computer Science*, Ithaca, New York, August 1994.

[40] T. B. Knoblock and E. Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN '96 Conference on PLDI*, Philadelphia, Pennsylvania, June 1996.

[41] M. Leone and P. Lee. A declarative approach to run-time code generation. In *Workshop on Compiler Support for System Software (WCSSS)*, February 1996.

[42] M. Leone and P. Lee. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN '96 Conference on PLDI*, pages 137–148, Philadelphia, Pennsylvania, May 1996.

[43] Y. A. Liu. *Incremental Computation: A Semantics-Based Systematic Transformational Approach*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, January 1996.

[44] Y. A. Liu. Principled strength reduction. In *Proceedings of the IFIP TC2 Working Conference on Algorithmic Languages and Calculi*, Le Bischenberg, France, February 1997. Chapman & Hall.

[45] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Discovering auxiliary information for incremental computation. In *Conference Record of the 23rd Annual ACM Symposium on POPL*, pages 157–170, St. Petersburg Beach, Florida, January 1996.

[46] Y. A. Liu and T. Teitelbaum. Caching intermediate results for program improvement. In *Proceedings of the ACM SIGPLAN Symposium on PEPM*, pages 190–201, La Jolla, California, June 1995.

[47] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–39, February 1995.

[48] V. Maslov. Lazy array data-flow dependence analysis. In *Conference Record of the 21th Annual ACM Symposium on POPL*, January 1994.

[49] J. A. Mason. *Learning APL : an array processing language*. Harper & Row, New York, 1986.

[50] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array data-flow analysis and its use in array privatization. In *Conference Record of the 20th Annual ACM Symposium on POPL*, January 1993.

[51] B. Paige and J. T. Schwartz. Expression continuity and the formal differentiation of algorithms. In *Conference Record of the 4th Annual ACM Symposium on POPL*, pages 58–71, January 1977.

[52] R. Paige. Transformational programming—Applications to algorithms and systems. In *Conference Record of the 10th Annual ACM Symposium on POPL*, pages 73–87, January 1983.

[53] R. Paige. Symbolic finite differencing—Part I. In *Proceedings of the 3rd ESOP*, volume 432 of *Lecture Notes in Computer Science*, pages 36–56, Copenhagen, Denmark, May 1990. Springer-Verlag, Berlin.

[54] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[55] H. A. Partsch. *Specification and Transformation of Programs—A Formal Approach to Software Development*. Springer-Verlag, Berlin, 1990.

[56] W. Pugh. Uniform techniques for loop optimization. In *International Conference on Supercomputing*, pages 341–352, Cologne, Germany, June 1991.

[57] W. Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 31(8), August 1992.

[58] W. Pugh and D. Wonnacott. Going beyond integer proramming with the omega test to eliminate false data dependences. Technical Report CS-TR-3191, Department of Computer Science, University of Maryland, College Park, Maryland, December 1992. An earlier version of this paper appeared at the ACM SIGPLAN '92 Conference on PLDI.

[59] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the 6th Annual Workshop on Programming Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, Portland, Oregon, August 1993.

[60] W. Pugh and D. Wonnacott. Nonlinear array dependence analysis. Technical Report CS-TR-3372, Department of Computer Science, University of Maryland, College Park, Maryland, November 1994.

[61] J. C. Reynolds. *The Craft of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[62] V. Sarkar and R. Thekkath. A general framework for iteration-reordering loop transformations. In *Proceedings of the ACM SIGPLAN '92 Conference on PLDI*, pages 175–187, San Francisco, California, June 1992.

[63] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.

[64] B. Steffen, J. Knoop, and O. Rüthing. Efficient code motion and an adaption to strength reduction. In *Proceedings of the 4th International Joint Conference on TAPSOFT*, volume 494 of *Lecture Notes in Computer Science*, pages 394–415. Springer-Verlag, Berlin, 1991.

[65] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the 2nd ESOP*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358, Nancy, France, March 1988. Springer-Verlag, Berlin.

[66] J. Warren. A hierarchical basis for reordering transformations. In *Conference Record of the 11th Annual ACM Symposium on POPL*, pages 272–282, January 1984.

[67] R. Waters. Efficient interpretation of synchronizable series expressions. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 74–85, June 1987.

[68] R. C. Waters. Automatic transformation of series expressions into loops. *ACM Transactions on Programming Languages and Systems*, 13(1):52–98, January 1991.

[69] J. Webb. Steps towards architecture-independent image processing. *IEEE Computer*, February 1992.

[70] B. Wegbreit. Goal-directed program transformation. *IEEE Transactions on Software Engineering*, SE-2(2):69–80, June 1976.

[71] W. M. Wells, III. Efficient synthesis of Gaussian filters by cascaded uniform filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(2):234–239, March 1986.

[72] D. M. Yellin and R. E. Strom. INC: A language for incremental computations. *ACM Transactions on Programming Languages and Systems*, 13(2):211–236, April 1991.

[73] R. Zabih. *Individuating Unknown Objects by Combining Motion and Stereo*. PhD thesis, Department of Computer Science, Stanford University, Stanford, California, 1994.

[74] R. Zabih and J. Woodfill. Non-parametric local transforms for computing visual correspondence. In J.-O. Eklundh, editor, *3rd European Conference on Computer Vision*, volume 801 of *Lecture Notes in Computer Science*, pages 151–158. Springer-Verlag, 1994.