Computer Science
Technical Report

Automatic Parallelization of "Inherently Sequential" Nested Loop Programs

Yun Zou and Sanjay Rajopadhye
Colorado State University
zou@cs.colostate.edu
Sanjay.Rajopadhye@colostate.edu

Technical Report CS-11-102

March 28, 2011

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

Automatic Parallelization of "Inherently Sequential" Nested Loop Programs

Yun Zou and Sanjay Rajopadhye
Colorado State University
zou@cs.colostate.edu
Sanjay.Rajopadhye@colostate.edu

March 28, 2011

Abstract

Most automatic parallelizers are based on detection of independent operations, and most of them cannot do anything if there is a true dependence between operations. However, there exists a class of programs, for which this can be surmounted based on the nature of the operations. The standard and obvious cases are reductions and scans (prefix computations), which normally occur within loops. We present a method for automatically parallelizing such "inherently" sequential programs. Our method is based on exact dependence analysis in the polyhedral model, and matrix multiplication over a semiring. It handles both single loop as well as arbitrarily nested loops. We also deal with mutually dependent variables in the loop. Finally, we present some optimizations for the code parallelization. Although the experimental results are preliminary, it shows that scan and reduction parallelizations are effective on practical applications.

Keywords: Automatic Parallelization, Polyhedral model, Recurrence Equations, Scan, Reduction, Matrix Multiplication, Semiring

1 Introduction

Multi-core and many core processors are becoming dominant in all areas of computing. Parallel programming is necessary to make efficient use of these architectures. However, parallel programming by hand is a challenge for most programmers, and automatic parallelization is therefore essential. Some tools [1, 2] have already been developed for automatic parallelization. Most automatic parallelizers focus on distributing independent operations among processors or threads, and they fail when there is a true (i.e., value based) dependence between operations. To overcome this limitation, new sources for parallelism need to be explored. One such source of parallelism allows the compiler to break a certain class of dependences, when the underlying computations come from a semantically-rich algebraic structure such as semiring. The most significant cases are scans and reductions [3].

A scan is an operation which takes a binary associative operator $\odot$ and an ordered set of expressions $[e_0, e_1, \ldots, e_{n-1}]$, and returns an ordered set $[e_0, e_0 \odot e_1, \ldots, e_0 \odot \cdots \odot e_{n-1}]$. A reduction is similar to a scan, but it only returns the single expression $e_0 \odot e_1 \odot \cdots \odot e_{n-1}$.

As is well known [3, 4], with P processors, a scan or reduction with size n can be parallelized with a time complexity of $O(\frac{n}{P} + log_2 P)$. Since $p \ll n$ in most proactical applications, this is linearly scalable (i.e., iso-efficient) parallelism.

The standard reduction and scan is characterized by an expression of the following form: $x_i = x_{i-1} \odot e_i$ that is repeatedly evaluated for a range of i. A compiler can recognize a scan by identifying expressions written in this form, but many scans and reductions are not written in the standard form. In a seminal paper, Kogge and Stone [5] showed that if we can transform a loop body expression into a so-called "state-vector update" (SVU) form, we can parallelize the loop as a scan or reduction. Look at the following example.

Example 1.

```
x[0] = b[0];
FOR i = 1 to n
    x[i] = a[i]*x[i - 1] + b[i];
END FOR
```

The loop computes x_i using expression $x_i = a_i \times x_{i-1} + b_i$, and this is not in the standard form, so no scan can be recognized. However, the loop body expression can be rewritten as an equivalent SVU expression:

$$\begin{pmatrix} x_i \\ 1 \end{pmatrix} = \begin{pmatrix} a_i & b_i \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_{i-1} \\ 1 \end{pmatrix}$$

Let $X_i = \begin{pmatrix} x_i \\ 1 \end{pmatrix}$, $A_i = \begin{pmatrix} a_i & b_i \\ 0 & 1 \end{pmatrix}$, and $X_0 = \begin{pmatrix} x_0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ the initial value for X_i. Hence, the above program is equivalent to $X_i = (\prod_{j=1}^{i} A_j)X_0$. Since matrix multiplication is associative, the product of the matrices can be parallelized as a scan. Many authors have generalized this elegant, well-known idea to automatic parallelization [6, 7, 8, 9].

We call an automatic parallelizer that first detects scans and reductions, and then parallelizes programs based on this, a *scan parallelizer*. Since scans and reductions normally occur within loops, most scan parallelizers analyze loop programs. Analysis of nested loops is much more difficult than that for a single loop, so most of the previous work only handles one dimensional loops [8, 9]. Redon and Feautrier [10] presented a method using the polyhedral model [11] to detect scans and reductions in arbitrary nested Affine Control Loop programs. However, they recognize scans based on pattern matching of the standard form, and a heuristic partial normalization algorithm that manipulates epressions into such a form.

In this paper, we present a more general and practical method for automatic parallelization based on reductions and scans. Our method, based on a formalism called the *polyhedral model*, is more general than previously proposed methods in two important ways: we handle arbitrarily nested affine loop programs, and we can detect a rich class of scans and reductions, based on extraction of expressions involving semiring operations expressed as matrix-vector products.

We integrated our technique into a polyhedral program transformation and code generation system. The core of the method works on a limited class of

programs and we therefore developed and implemented a normalization algorithm to bring a richer class of programs to this normal form. This allowed us to discover many hidden scans. We also propose some optimizations for the parallelization of reductions and scans.

The remainder of this paper is organized as follows: Section 2 gives some background about the polyhedral model and the definitions that are used in the rest of paper. Section 3 gives the examples that are used in the paper. Section 4 describes how to deduce the SVU form. Section 5 presents our normalization method based on the Polyhedral Reduced Dependence Graph. Section 6 describes how to do parallelization and optimization about scan and reduction. In Section 7, we discuss related work, and finally, we present our conclusions.

2 Preliminaries

2.1 Polyhedral model

The compute intensive parts of many applications often spend most of their execution time in nested loops. This is particularly common in scientific and engineering applications, signal and image processing, bioinformatics, etc. The Polyhedral model provides a powerful abstraction to reason about a class of loop programs, those that consist of arbitrary nested loop programs for which the loop bounds and array accesses are affine functions of outer loop variables and program parameters. Many authors [1, 11, 12] show that polyhedral model is very useful in code generation and automatic parallelization.

Definition 1 (Domain). The domain of a statement describes the iteration space in which the statement is defined, it is represented by a set of linear inequalities. For example, the domain for x in Example 1 is represented as $\{i|0 \leq i < n\}$, i is the index for the iteration space.

Definition 2 (Dependence). Two iterations S_i and S_j are said to be dependent, written as $(S_i \rightarrow S_j)$, if they access the same memory location and one of them is a write. A true dependence exists if the source writes the memory location and the target reads the memory location. In this paper, we handle a class of programs for which a preprocessing analysis can precisely identify all the the true dependence.

Definition 3 (Uniform Dependence and Non-uniform Dependence). A uniform dependence is a dependence where distance between the source and target iteration is a constant vector. In contract, if the distance is an affine function but not a constant, the dependence is called non-uniform dependence.

2.2 Polyhedral Reduced Dependence Graph

An important representation that we will use later is the *Polyhedral Reduced Dependence Graph* (PRDG).

Before we give the definition of PRDG, let us see *Reduced Dependence Graph* (RDG) first. In the RDG, each vertex represents a statement in the program,

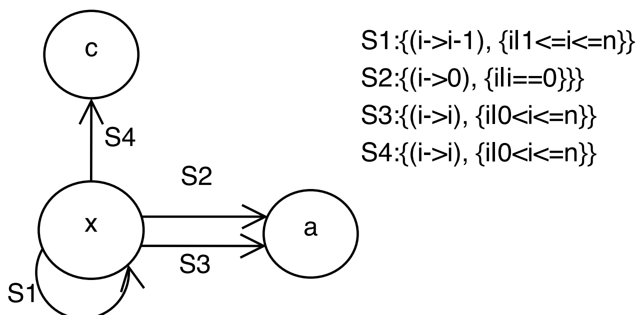


Figure 1: The PRDG for Example 1

there is an edge from vertex v_1 to v_2, if v_1 depends on v_2. A PRDG is an RDG, where every edge is additionally labeled with a dependence, represented as $\{f, D\}$, where f is the dependence function, and D is the domain where the dependence is defined. Figure 1 shows the PRDG for Example 1.

2.3 Terminology

Here we define some terminology that is used in this paper:

Definition 4 (Recurrence Variables). A variable (scalar variable or array variable) in a loop program is called a recurrence variable iff the variable is directly or indirectly used in its own definition.

Definition 5 (Linear Recurrence Equations). A linear recurrence equation is defined as

$$x_z = f(x_{z-d_1}, \ldots, x_{z-d_m})$$

Where z belongs to the domain of x, d_i is called a dependence vector. A loop program can be transformed into a system of recurrence equations by exact data flow analysis [13]. For example, Example 1 is transformed to:

$$x[i] = \begin{cases} a[0], & i = 0 \\ a[i] \times x[i-1] + b[i], & 0 < i < n \end{cases}$$

Definition 6 (Semiring). A two-operator algebraic structure $(R, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is called a semiring, if R is the carrier, $\oplus$ is an associative and commutative binary operator with identity element $\mathbf{0}$, $\otimes$ is an associative binary operator with identity $\mathbf{1}$, and $\otimes$ distributes over $\oplus$.

Definition 7 (Matrix multiplication form). A system of recurrence equations is in SVU form iff it can be rewritten as follows:

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ 1 \end{pmatrix} \leftarrow \begin{pmatrix} e_{1,0} & e_{1,1} & \cdots & e_{1,m} \\ e_{2,0} & e_{2,1} & \cdots & e_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \times_{\{\oplus,\otimes\}} \begin{pmatrix} l_1 \\ l_2 \\ \vdots \\ l_n \\ 1 \end{pmatrix}$$

2.4 Target system of recurrence equations

In our technique, we focus on a system of recurrence equations. The target system of recurrence equations that can be handled by our method needs to satisfy the following properties:

- The recurrence equations in the system are defined on the same domain and the PRDG for the system of recurrence equations is a strongly connected component.
- The dependence for the recurrence variables is uniform (except the dependence in the initial values), which means each dependence vector needs to be an integer vector.
- For each recurrence variable x, the dependence vectors on x are in the same direction.

3 Examples

We now present a number of examples that have various types of prefix computations and scans. These examples are repreatedly used in this paper.

Example 2. The following example computes the *maximum segment sum* (mss). Given an array, a, of n elements, the *segment* $\langle i, j \rangle$ is the subarray from the i-th to the j-th elements, inclusive. A segment sum, $S[i, j]$ is the sum of all the elements in the segment $\langle i, j \rangle$, and the mss of the array is the maximum of $S[i, j]$ over the $n^2/2$ segments.

```
x = a[0];
mss = x;
FOR i = 1 to n
    x = max(a[i], x + a[i]);
    mss = max(mss, x);
END FOR
```

The equivalent system of recurrence equations is:

$$x[i] = \begin{cases} a[0], & i = 0 \\ \max(a[i], a[i] + x[i-1]), & 0 < i < n \end{cases}$$

$$mss_tmp[i] = \begin{cases} x[0], & i = 0 \\ \max(mss_tmp[i-1], x[i]), & 0 < i < n \end{cases}$$

$$mss = mss_tmp[n];$$

max is a binary operator which takes two values as input and returns the bigger one. Most scan parallelizers fail since the x computed is immediately used for computing mss.

Example 3. This example is simply Fibonacci. The following program computes the first n fibonacci numbers, $fib[i] = fib[i-1] + fib[i-2]$.

```
fib[0] = 0;
fib[1] = 1;
FOR i = 2 to n
    fib[i] = fib[i-1] + fib[i-2];
END FOR
```

The equivalent system of recurrence equations is:

$$fib[i] = \begin{cases} 0, & i = 0 \\ 1, & i = 1 \\ fib[i-1] + fib[i-2], & 1 < j < n \end{cases}$$

Example 4. The following example computes an array of scans. The computation for the program is $x[i][j] = \sum_{k=0}^{j} a[i][k]$.

```
FOR i = 0 to n
    x[i][0] = a[i][0];
  FOR j = 1 to m
      x[i][j] = x[i][j-1] + a[i][j];
  END FOR
END FOR
```

The equivalent system of recurrence equations:

$$x[i][j] = \begin{cases} a[i][0], & 0 \leq i < n, j = 0 \\ x[i][j-1] + a[i][j], & 0 \leq i < n, 0 < j < m \end{cases}$$

Example 5. The following program does a lexicographical prefix sum computation on a triangular space with domain $\{i, j | 0 \leq i < n, 0 \leq j \leq i\}$. $x[i][j] = \sum_{k=0}^{i-1} \sum_{l=0}^{k} a[l][k] + \sum_{k=0}^{j} a[i][k]$.

```
x[0][0] = a[0][0];
FOR i = 1 to n
    x[i][0] = x[i-1][i-1] + a[i][0];
    FOR j = 1 to i
        x[i][j] = x[i][j-1] + a[i][j];
    END FOR
END FOR
```

The equivalent system of recurrence equations:

$$x[i][j] = \begin{cases} a[0][0], & i = 0 \\ x[i-1][i-1] + a[i][0], & 0 < i < n, j = 0 \\ x[i][j-1] + a[i][j], & 0 < i < n, 0 < j \leq i \end{cases}$$

Most scan parallelizers detect the trivial scans inside the innermost loop, but will not detect the whole program as a lexicographical scan.

Example 6. The following program exhibits a mutual dependence between variable x and y.

```
x[0] = a[0];
y[0] = b[0];
FOR i = 1 to n
   x[i] = x[i-1] + y[i-1] + a[i];
   y[i] = x[i-1] + y[i-1] + b[i];
END FOR
```

The equivalent system of recurrence equations:

$$x[i] = \begin{cases} a[0], & i = 0 \\ x[i-1] + y[i-1] + a[i], & 0 < i < n \end{cases}$$

$$y[i] = \begin{cases} b[0], & i = 0 \\ x[i-1] + y[i-1] + a[i], & 0 < i < n \end{cases}$$

4 Detection of scans

Our analysis, like Redon and Feautrier, first performs exact data-flow analysis [13] of an affine control loop program to extract a System of Affine Recurrence Equations (SARE, or SRE). Given such an SRE, the next step is to examine all the *self* dependences of a variable on itself, either direcly or through a cycle in the PRDG. If such a cyclic dependence exists, and is a *uniform* dependence, we have identified a *recurrence variable.* A recurrence variable is a scan variable if all the self dependences of this variable are *uniform* (i.e., translations) and in the same direction. After identifying the scan variable, we try to determine whether the computation that updates the scan variable at any iteration point can be written as a *linear semiring expression.* If we do this, the final step is

$\Phi :: (Exp, x, Y) \rightarrow (Exp, boolean)$
$\Phi[\![c]\!] = (c, False)$
$\Phi[\![x]\!] = (\mathbf{1}, True)$
$\Phi[\![y]\!] = (\mathbf{0}, True)$
$\Phi[\![v]\!] = (v, False)$
$\Phi[\![f\ e]\!] = let\ (e', b) = \Phi[\![e]\!]$
  in if b **then** *error* **else** $(f\ e', False)$
$\Phi[\![e_1 \odot e_2]\!] = let\ ((e_1^{'}, b_1), (e_2^{'}, b_2)) = (\Phi[\![e_1]\!], \Phi[\![e_2]\!])$
  in if $b_1 \vee b_2$ **then** *error* **else** $(e_1^{'} \odot e_2^{'}, False)$
$\Phi[\![e_1 \oplus e_2]\!] = let\ ((e_1^{'}, b_1), (e_2^{'}, b_2)) = (\Phi[\![e_1]\!], \Phi[\![e_2]\!])$
  in case (b_1, b_2) **of**
    $(True, True) \rightarrow (e_1^{'} \oplus e_2^{'}, True)$
    $(True, Flase) \rightarrow (e_1^{'}, True)$
    $(False, True) \rightarrow (e_2^{'}, True)$
    $(False, False) \rightarrow (e_1^{'} \oplus e_2^{'}, False)$
$\Phi[\![e_1 \otimes e_2]\!] = let\ ((e_1^{'}, b_1), (e_2^{'}, b_2)) = (\Phi[\![e_1]\!], \Phi[\![e_2]\!])$
  in if $b_1 \wedge b_2$ **then** *error* **else** $(e_1^{'} \otimes e_2^{'}, b_1 \vee b_2)$

Figure 2: Algorithm Φ extracts the coefficient matrix for scan variable x from expression Exp over semiring $(R, \oplus, \otimes)$. Y denotes a list of scan variables other than x, v denotes a non-scan variable, c denotes a constant, f denotes a unary operator, $\odot$ denotes a binary operator other than $\oplus$ or $\otimes$.

to extract the coefficient matrix for the scan variables. The algorithm to extracting the linear terms in the matrix is shown in Figure 2, and the one for extracting the coefficient of the constant term is similar to it. This algorithm, extends Kogge and Stones's original work [5], and enhances those developed by Xu et. al and Sato and Iwasaki [7, 9].

In this section, we first describe how to transform a single recurrence equation into matrix multiplication form, then show how to extend this to a system of recurrence equations, and finally, how to detect lexicographic scans.

4.1 First order recurrence equation

Given a semiring $(R, \oplus, \otimes)$, a *first* order recurrence equation is defined as follows:

$$z \in D : x_z = a \otimes x_{z-d} \oplus b \tag{1}$$

Where a and b are arbitrary expressions.These expressions may involve other variables that do *not* have a cyclic dependence, and therefore are considered as inputs to the computation of x. There might be additional subexpressions $a' x_{z-d}$ on the right hand side involving the *other* self dependences on x, but they all must have the same d, and these can be replaced, without loss of generality, by a single expression a. Since there is only one recurrence dependence, the dependence for the recurrence variable is always in the same direction.The

following discusses how a matrix form can be extracted under different situations.

If $gcd(d) = 1$, then a matrix form can be extracted

$$\begin{pmatrix} x_z \\ 1 \end{pmatrix} \leftarrow \begin{pmatrix} a & b \\ 0 & 1 \end{pmatrix} \times_{\{\oplus,\otimes\}} \begin{pmatrix} x_{z-d} \\ 1 \end{pmatrix}$$

.

If $\gcd(d) = t > 1$. For example, $x_i = x_{i-2} + a_i$, this computation can be computed with two scans, one scan on odd elements and another on even elements. Let $d = t\delta$, where $\gcd(\delta) = 1$. We can transform equation (1) into a matrix form by adding $t - 1$ temporary "accumulation variables." The matrix form for equation (1) is shown below.

$$\begin{pmatrix} x_z \\ x_{z-\delta} \\ x_{z-2\delta} \\ \vdots \\ x_{z-(t-1)\delta} \\ 1 \end{pmatrix} \leftarrow \begin{pmatrix} 0 & 0 & \cdots & a & b \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 1 & 0 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{pmatrix} \times_{\{\oplus,\otimes\}} \begin{pmatrix} x_{z-\delta} \\ x_{z-2\delta} \\ x_{z-3\delta} \\ \vdots \\ x_{z-d} \\ 1 \end{pmatrix}$$

4.2 M-th order recurrence equation

Unlike first order recurrences, in an m-th order recurrence equation, the recurrence variable x occurs m times in the right hand side of equations. It can be rewritten in the form,

$$z \in D : x_z = (a_1 \otimes x_{z-d_1}) \oplus \cdots \oplus (a_m \otimes x_{z-d_m}) \oplus b \qquad (2)$$

Where a is a set of m expressions and b is a single expression. As described at the beginning of this section, x is a scan variable if all the dependences on x are in the same direction, so first we check every dependence d_i, if $\frac{d_i}{\gcd(d_i)}$ are the same. If this does not hold, the computation is not a scan. Let us assume that this holds, and the set $\{d_1, d_2, \ldots, d_m\}$ is in ascending order of the value of $\gcd d_i$.

If $\gcd(d_1) = 1$ and $d_i = id_1\,(i > 1)$, then equation (2) can be transformed to the matrix form

$$\begin{pmatrix} x_z \\ x_{z-d_1} \\ x_{z-d_2} \\ \vdots \\ x_{z-d_{m-1}} \\ 1 \end{pmatrix} \leftarrow \begin{pmatrix} a_1 & \cdots & a_m & b \\ 1 & \cdots & 0 & 0 \\ 0 & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix} \times_{\{\oplus,\otimes\}} \begin{pmatrix} x_{z-d_1} \\ x_{z-d_2} \\ x_{z-d_3} \\ \vdots \\ x_{z-d_m} \\ 1 \end{pmatrix}$$

If $\gcd(d_1) \neq 1$ or $d_i = kd_1\,(k \neq i)$, we can apply the trick that we used in the first order recurrence, of adding some temporary accumulation variables, using which we can also transform equation (2) into a semiring matrix form.

As we see above, as long as all the self dependences of the scan variable in the recurrence equation are in the same direction, we can transform the recurrence equation into matrix form, and compute it as a scan or reduction. However, if all the directions are not the same, for example: $x_{i,j} = x_{i-1,j} + x_{i,j-1} + x_{i-1,j-1} + a_{i,j}$, we could try to obtain wavefront schedules using the classic polyhedral scheduling algorithms [11] for this kind of dependence. Since this is not the focus of this paper, we are not going to discuss it here.

4.3 System of recurrence equations

In a system of recurrence equations, there is more than one recurrence variables and those recurrence variables are the nodes in a SCC. Example 6 is one of such system of recurrence equations.

Now we are going to show how to transform a system of recurrence equations into matrix multiplication form. For each recurrence variable v in the system, we check if all the direct and indirect dependences on v are in the same direction. If all the dependences on each variable are in the same direction, then a matrix multiplication form can be extracted, we can also add some temporary "accumulator variables" if it is necessary.

In Example 6, there are two dependences on x, $(i \to i-1)$ and $(i \to i-1)$, they are in the same direction, the dependences on y are also in the same direction, so a matrix multiplication form can be extracted. The matrix multiplication form for Example 6 is shown below:

$$\begin{pmatrix} x_i \\ y_i \\ 1 \end{pmatrix} \leftarrow \begin{pmatrix} 1 & 1 & a[i] \\ 1 & 1 & a[i] \\ 0 & 0 & 1 \end{pmatrix} \times_{\{+,\times\}} \begin{pmatrix} x_{i-1} \\ y_{i-1} \\ 1 \end{pmatrix}$$

4.4 Lexicographical Scan

For Example 5, most scan parallelizers only detect the scan inside the inner most loop. Redon and Feautrier [10] showed that it is useful to detect the scan as a lexicographical scan instead of arrays of scans. The parallelization for lexicographical scan is more efficient than parallelization of sequence of scans.

Based on Redon and Feautrier's [10] algorithm for detecting multi-directional scan. We present a way for detecting the lexicographical scan. Whenever a matrix form is extracted, which means a scan is detected, we check the initial values of the detected scan. A branch of the initial values can be combined with the detected scan if and only if it satisfies the following:

- It is a recurrence equation of the same recurrence variables with the detected scan.
- The input value of the recurrence equation is the lexical maximum values computed by the detected scan.
- The branch can be transformed into a matrix multiplication regarding to the recurrence variables.

After we combine all the possible branches, if there is no more initial values or the rest of the initial values only computes the initial value for lexical minimum value for the scan, the detected scan is a lexicographical scan.

For example, in example 5, a scan about variable x is detected in the third branch, the branch $\{i, j | 0 < i < n, j = 0\}$ compute the initial values of the detected scan. The branch $\{i, j | 0 < i < n, j = 0\}$ is a recurrence equation about x, and the lexical maximum values computed buy the scan $x[i-1][i-1]$ is the input for this branch. and the a matrix form $\begin{pmatrix} 1 & a[i][0] \\ 0 & 1 \end{pmatrix}$ can be extracted for the branch, then the branch of the initial value into the scan. After that, no more branches can be merged into the scan and there is no more initial values to be merged.

4.5 Reduction

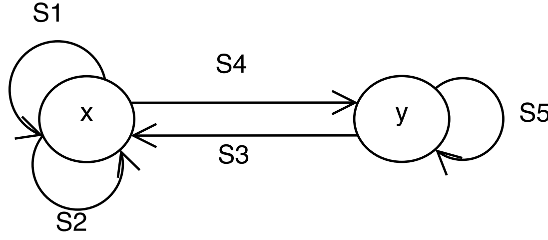
Until now, we have only focused on scan detection. Let us now address detection of reductions. Based on the recurrence equations, we can say that a reduction is a special case of scan, any value computed in the scan can be computed as a reduction. If a scan is used only on a finite domain, then it is not necessary to compute all the values, so we can recognize the values in this finite domain as reductions and remove the scan. For example, in the mss example, mss_*tmp* is detected as a scan first, however, there is only one value of mss_*tmp* is used mss_*tmp*$[n]$ in the definition of mss, so we say that the variable mss is a reduction.

5 Normalization

Most of the systems of recurrence equations is not written in the standard way as we defined. Some strategies needs to be applied to rewrite the system without changing the meaning of the code, so that a matrix form can be extracted, this strategy is called normalization. Our normalization is based on the Strongly Connected Components (SSCs) of the PRDG.

5.1 Preprocessing

A simple preprocessing can be used to filter out the parts that do not belong to scans. As we described in the target recurrence system, all the recurrence equations are defined on the same domain, which means all the dependences on the recurrence variables are defined on the same domain and those dependences need to be uniform. In other words, if there exists a non-uniform dependence d on variable v with a domain D, it is impossible for our method to extract v as a scan variable on domain D, so we can ignore all dependence on D. Moreover, we do not want the non-uniform dependences in initial values occur in the analysis. We can ignore those dependence by simply removing the edges from the PRDG.



S1:{(i,j->i,j-1), {i,j|0<=i<n,1<j<n}}
S2:{(i,j->i,j-1), {i,j|0<=i<n,n<j<2n}}
S3:{(i,j->i,j-1), {i,j|0<=i<n,1<j<n}}
S4:{(i,j->i,j-1), {i,j|0<=i<n,1<j<n}}

Figure 3: The PRDG for the splitting example

5.2 Splitting

Consider the following system of recurrence equations:

$$x[i][j] = \begin{cases} b, & 0 \le i < n, j = 0 \\ x[i][j-1] + y[i][j-1], & 0 \le i < n, 1 < j < n \\ a[j-n], & 0 \le i < n, j = n \\ x[i][j-1] + a[j-n], & 0 \le i < n, n < j < 2n \end{cases}$$

$$y[i,j] = \begin{cases} c, & 0 \le i < n, j = 0 \\ y[i][j-1] + x[i][j-1], & 0 \le i < n, 1 < j < n \end{cases}$$

The SCC of the PRDG of this system is shown in Figure 3. It is easy to see from the equations that the system contains two scans: on domain $\{i, j | 0 \le i < n, 1 < j < n\}$, variable x and y together construct a scan, the direction for x is $(0,1)$ and $(0,1)$ for y; another scan is x on domain $\{i, j | 0 \le i < n, n < j < 2n\}$ with direction $(0,1)$, it is a scan on array a.

To detect the scans in this kind of system, we need to do some proper splitting on the system. Since the target system of recurrence equations we solve requires all the uniform dependences of a recurrence variable are defined on the same domain, in other words, one scan is defined on one domain, we are going to do splitting according to the domain of the uniform dependence. The algorithm is shown as below:

Algorithm 1. Splitting.

1. Construct the PRDG for the system.

2. In a SCC, for each vertex x, x is defined on D_x.

- For a uniform dependence d on x, d is defined on domain D_0, the initialization domain for this dependence is D_{g_0}. Find all the dependence $\{d_1, d_2, \ldots, d_m\}$ on v, where $D_{d_i} \cap D_0 \neq \emptyset$, D_{d_i} is the domain for d_i, compute the initialization domains for the set of dependence, assume they are $\{D_{g_1}, D_{g_2}, \ldots, D_{g_m}\}$. Define domain $D = D_0 \cup D_{g_i}, (0 \leq i \leq m)$.
- If $D_x - D \neq \emptyset$. Split the definition for x according to D.
- Go back to step 1. This procedure is performed until there is no more splitting can be performed.

In the above example, for vertex x, one of the dependences $\{i, j \to i, j-1\}$ on x is defined on domain $\{i, j | 0 \leq i < n, n < j < 2n\}$, it is different from the domains for other uniform dependence. The initialize domain for the dependence is $\{i, j | 0 \leq i < n, j = n\}$, then we get $D = \{i, j | 0 \leq i < n, n \leq j < 2n\}$. Do splitting on x according to D, we get the new system:

$$x1[i][j] = \begin{cases} b, & 0 \leq i < n, j = 0 \\ x[i][j-1] + y[i][j-1], & 0 \leq i < n, 0 < j < n \end{cases}$$

$$x2[i][j] = \begin{cases} a[j-n], & 0 \leq i < n, j = n \\ x[i][j-1] + a[j], & 0 \leq i < n, n < j < 2n \end{cases}$$

$$y[i][j] = \begin{cases} c, & 0 \leq i < n, j = 0 \\ y[i][j-1] + x[i][j-1], & 0 \leq i < n, 0 < j < n \end{cases}$$

Now there is two SCCs in the PRDG of the system, and for each SCC, no more splitting can be performed. Two scans can be detected based on the two SCCs in the PRDG.

5.3 Substitution

Consider the following example:

$$x[i] = \begin{cases} a[0], & i = 0 \\ x[i-1] + y[i-1] + a[i], & 0 < i < n \end{cases}$$

$$y[i] = \begin{cases} b[0], & i = 0 \\ y[i-1] + x[i] + b[i], & 0 < i < n \end{cases}$$

The PRDG for this system is shown in Figure 4. In this system, all the uniform dependences of recurrence variable x and y are defined on the same domain. However, the dependences on x are $\{(i \to i-1), (i \to i)\}$, they are not on the same direction. Our method will fail to detect this system of recurrence equations as a scan.

We present a normalization method based on simple substitution. The substitution rule for a dependence between two variables is shown in Figure 5.

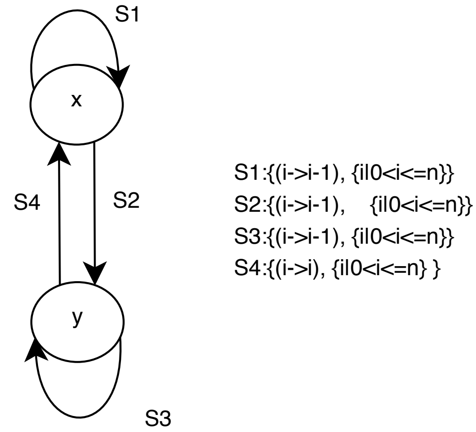


Figure 4: The PRDG for the system with mutual dependence

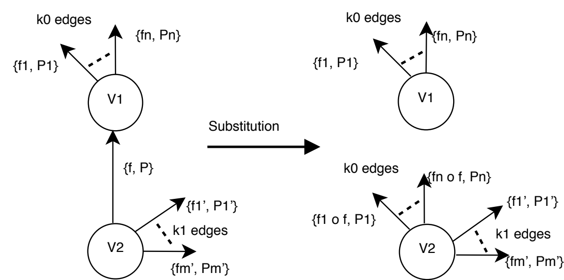


Figure 5: Substitution rule for a dependence

As is shown in Figure 5, there are $(k_1 + 1)$ dependences for v_2, one is on x, k_1 are on other variables. For v_1, there are k_0 dependences. Now we want to remove the dependence from v_2 to v_1. We achieve this by substituting the definition of v_1 into v_2. Assume the polyhedron for the dependence we want to remove is $\{f, P\}$, where f represents the dependence function, P represents the domain for the dependence, the new PRDG can be construct with the following rules:

- For each variable v that v_1 depends on, we add a new edge from v_2 to v, if the polyhedron for the dependence from v_1 to v is $\{f', P'\}$, the polyhedron for the new edge from v_2 to v is $\{f' \circ f, P\}$represents composition operator.
- Remove the edge from v_2 to v_1.

In this normalization, we only check the dependences between different variables, since a the self dependence can never be removed. A dependence can be normalized only when it is identity or its direction is opposite of the given direction. Since the latter situation will lead to a cyclic dependence, which will not occur in a legal program, only normalization for the dependence with identity function will succeed. The following algorithm describes the algorithm for substitution.

Algorithm 2. Substitution

- For each vertex v in the SCC.
  1. Initialize the direction d for v with the direction of one of the self dependences.
  2. For each dependence d_i on v (d_i is from w to v), if d_i is not on the same direction with d.
     - If d_i is identity. Remove the dependence edge for d_i by substitution.
     - Else normalization fails, which means we are not able to normalize the dependence on v to the same direction, so v is not a scan variable, then we ignore the vertex v by removing the vertex and all the dependence on v from the PRDG.

5.4 Partial normalization

Splitting and substitution are the main normalizations we need to do. To make sure that the algorithm for scan detection works well, there are still some trivial normalizations we need to do. Consider the following example:

$$X[i] = a[i] * X[i-1] - b[i]$$

In the above equation, the recurrence variable is X, the binary operators involved are $\times$ and $-$, which can not construct a legal semiring, our scan detection will fail. However, since $-$ is the inverse of $+$ we can rewrite the equation to:

$$X[i] = a[i] * X[i-1] + (-b[i])$$

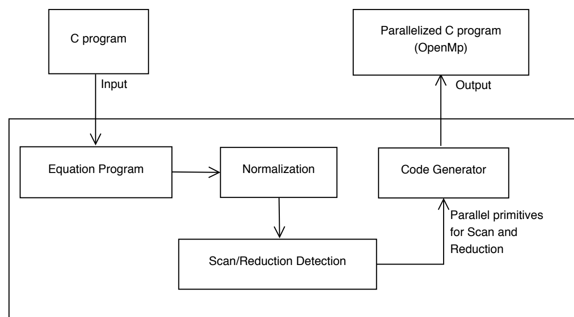


Figure 6: Framework for the scan parallelizer

Now the binary operators involved become $\times$ and $+$, now a legal semiring can be extracted. So a normalization according to the inverse of the semiring operator is necessary to help detecting scans as much as possible.

5.5 Algorithm summary

For a dependence $(z \rightarrow I(z))$, define the dependence level p as the greatest integer that:

$$I(z)[1...p] = z[1...p]$$

Our method does scan detection level by level. At each level p we only consider about the dependences at level p. The following algorithm gives a summery about the detection of scans.

Algorithm 3. Scan Detections.

1. Let S be the system of recurrence equations we are going to solve.
2. Do splitting on S. The system after splitting is S'.
3. Do scan detection at level p from the maximum nesting level until 0.
   - Construct the PRDG based on the dependences at level p.
   - Do preprocessing.
   - Compute the SCCs of the graph.
   - For each SCC g, first check if all the dependences are defined on the same domain. If they are defined on the same domain, do substitution and apply the scan detection if substitution succeeded.

6 Automatic parallelization

The framework for our scan parallelizer is shown in Figure 6. We integrated our

$\oplus'$	Z	I	C	V
Z	Z	I	C	V
I	I	V	V	V
C	C	V	V	V
V	V	V	V	V

$\otimes'$	Z	I	C	V
Z	Z	Z	Z	Z
I	Z	I	C	V
C	Z	C	V	V
V	Z	V	V	V

Figure 7: Semantics for $\oplus'$ and $\otimes'$ with the abstract values

scan detection into a polyhedral program transformation and code generation system. The automatic parallelization part is current not done yet.

In this section, we are going to talk about how we are going to parallelize a scan or reduction. We also proposed some optimization strategies to improve the performance.

6.1 Parallelization for scan and reduction

Many works have been done for the parallelization of scan and reduction [3, 8, 14, 15]. In our work, we will implement parallelized scans and reductions as primitives using the strategy described in Merrill's work [14].

The parallel algorithm for reduction consists of two phases: local reduction and global reduction. Given an associative operator $\odot$ an a set of expressions $[e_0, e_1, \ldots, e_{n-1}]$, we want to do a reduction with p threads. Fist, we distribute the n elements to the p threads, every thread performs a sequential reduction on $\frac{n}{p}$ elements. After p local reduction values are produced, a single thread performs a global reduction on the p local reduction values.

We parallelize scan with three phases: local reduction, global scan and final local scan. Similar to reduction, first, we distribute the n elements to the p threads, every thread performs a sequential reduction on $\frac{n}{p}$ elements. After p local reduction values are produced, a single thread performs a global scan on the p local reduction values. Finally, each thread performs a local scan with the proper seeding value form the global scan.

6.2 Optimizations of Matrix Multiplication

Matsuzaki [6] presented an optimization based on abstract matrix multiplication in the work for automatic parallelization of tree reductions. Their method removes the redundant variables and computations by detecting the constant propagations. Based on this idea, we proposed some optimizations of matrix multiplication.

We use Z to denote **0**, I to denote **1**, C denotes the constant value and V denotes non-constant values. The semantics for $\oplus'$ and $\otimes'$ is shown in Figure 7. Let $M_i = \prod_{j=0}^{i} A_i$, $M_i = A_i \times \{\oplus', \otimes'\} M_{i-1}$. In the optimization phase, we are going to iterate through the matrix until the same matrix pattern appears.

Assume the following matrix is the initial matrix.

$$\begin{pmatrix} V & V \\ Z & I \end{pmatrix}$$

The iteration for the matrix yields the following results.

$$\begin{pmatrix} V & V \\ Z & I \end{pmatrix} \rightarrow \begin{pmatrix} V & V \\ Z & I \end{pmatrix} \rightarrow \begin{pmatrix} V & V \\ Z & I \end{pmatrix}$$

The stable matrix has two V elements, which indicates that we need those two V elements for the computation. Similarly, if the initial matrix is similar to the above matrix, but with the first element as I, we can find that it yields the following computation.

$$\begin{pmatrix} I & V \\ Z & I \end{pmatrix} \rightarrow \begin{pmatrix} I & V \\ Z & I \end{pmatrix} \rightarrow \begin{pmatrix} I & V \\ Z & I \end{pmatrix}$$

So we only need one V element for the computation. Beside this kind of optimization, we also try to look at the computations between the matrix multiplication. Since we handle higher order recurrence equations, we will have the matrix A_i with the following pattern.

$$\begin{pmatrix} V & V & V \\ I & Z & Z \\ Z & Z & I \end{pmatrix}$$

There exists a unique permutation matrix P, by which $A_i^{'} = PA_i$ can be permuted to the following pattern.

$$\begin{pmatrix} I & Z & Z \\ V & V & V \\ Z & Z & I \end{pmatrix}$$

We have $M_i = A_i \times \{\oplus', \otimes\} M_{i-1} = P^{-1} P A_i \times \{\oplus', \otimes'\} M_{i-1} = P^{-1}(A_i^{'} \times \{\oplus', \otimes'\} M_{i-1})$, now let's see $A_i^{'} \times \{\oplus', \otimes'\} M_{i-1}$, assume M_{i-1} is an arbitrary matrix, we can use V to represent every element for arbitrary elements.

$$\begin{pmatrix} I & Z & Z \\ V & V & V \\ Z & Z & I \end{pmatrix} \times \{\oplus', \otimes'\} \begin{pmatrix} V & V & V \\ V & V & V \\ V & V & V \end{pmatrix} \rightarrow \begin{pmatrix} V & V & V \\ V' & V' & V' \\ V & V & V \end{pmatrix}$$

Here we use the V' to represent the value for the corresponding element changes. We can see that in this computation, the elements for the first row and last row in the result matrix is the same as the first row and last row of M_{i-1}, so we only need to do computation for the second row and permute the matrix back after the computation.

There are some other trivial optimizations we can do. For example, if A_i is a constant matrix, which never changes, then we can do the reduction for A_i

in log_n step using the power of A_i. We can also do parallelization for matrix multiplication when there is no optimization discovered. Moreover, in the local reduction phase for scan, the last thread does not need to do reduction, since its reduction value is never used in the following computation.

6.3 Experiment Results

To confirm the efficiency and scalability of the parallelization algorithm, we did some hand parallelization experiments on the following examples.

Array Scan It performs a scan on a given array a, the computation is $x_i = \sum_{j=0}^{i} a[j]$. The parallelized code computes a scan on a $(2, 2)$ matrix on over the semiring $(float, \times, +)$. The test size for the array is 2^{28}.

Polynomial Scan The polynomial scan computes $x[i] = cx[i-1] + a[i]$. The parallelized computation computes a scan on a $(2, 2)$ matrix over semiring $(float, \times, +)$. The test size for this problem is 2^{28}.

Maximum Segment Sum The computation for maximum segment sum is shown in Example 2. MSS is used in some important applications in filter design and bio-informatics. It is known as a programming pearl and have been studied by many researchers. It is parallelized as a reduction on a 3 by 3 matrix with a semiring $(float, \max, +)$. The test length of the input array is 2^{29}.

Fibonacci The fibonacci program is shown in Example 3. The Fibonacci sequence is a well known math problem. It is used in the analysis of financial markets, in strategies such as Fibonacci retracement, and also used in computer algorithms such as the Fibonacci search technique and the Fibonacci heap data structure. It is parallelized with a 3 by 3 matrix with a semiring $(int64, \times, +)$. int64 represents the type for 64 bits integer. We generated a fibonacci sequence with length 2^{28}.

Maximum subarray problem The maximum subarray problem is also called two dimensional maximum segment sum. Given a two dimensional array (m, n), it computes the subarray with the maximum sum among all the subarrays and returns the maximum sum. This computation is widely used in image processing applications. Since this computation includes computation of arrays of scans and arrays of maximum segment sum, we do parallelization among independent scans. In the test, the input array size is (2000, 1000).

All the testing is done on a machine equipped with one Xeon3450 running on double threads (4 cores; 2.66 GHz) and 8 GB memory. The running environment is Fedora 14, each program is compiled gcc 4.5.1 with the O3 optimization. For speedup, we compare with the sequential code without matrix multiplication.

The results are shown in Figure 8. Most of them show good scalability and all of the implementations are no worse than the sequential code. For the array scan and poly, we got more than 3 times speed up.

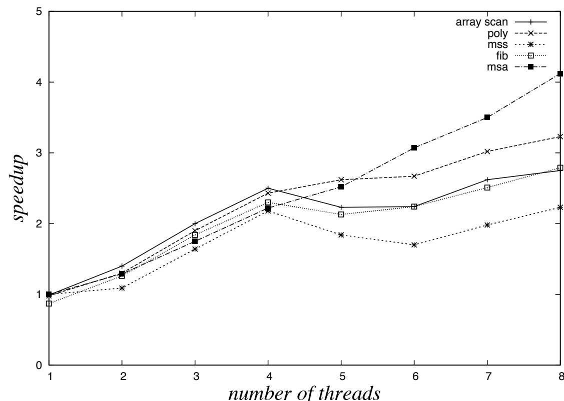


Figure 8: Speedup for the testing program

7 Related Work

The parallel implementation of recurrence equations was first discussed by Karp, Miller and Winograd [16]. They treated program dependences as inviolate constraints that any parallelization had to respect. Later, in a seminal paper, Kogge and Stone [5] described the first successful "dependence breaking" technique, that proposed a parallelzation of a general class of recurrence equations. They also introduced the "matrix notation" where the computation is described as a small matrix-vector product, and the associativity of this operation leads to efficient and scalable parallelization.

Lander and Fischer [17] descibed an efficient, general-purpose circuit for scan operations. Blelloch [3] describes the implementation of prefix-sum computation on parallel machines, and gives a strong motivation for using scan computations as a "primitive" or a library. He presents a set of practical examples, such as quicksort line-of-sight and watershed computations in topographical/geographical data, and spanning tree computations.

In the context of automatic parallelization, especially in the polyhedral model, the earliest work on parallelization of reductions and scans is due to Redon and Feautrier [10]. They present a scan detector which is based on analyzing systems of recurrence equations extracted from an imperfectly neted affine control loop program. They deal with scalar reductions, array reductions/scans and arrays of reductions/scans. They also described a scan algebra for the combination of scans, and some semantics preserving transformations on recurrence equations that embody scans.

They propose and use a normal form on which the main algorithm is applicable, and a normalization technique to bring other more general programs

into such a form. They separate the system graph into strongly connected components (SCCs), and use the core algorithm separately on each SCC. The core algorithm effectively identifies a dependence cycle involving a node, performs repeated substitution through a process called called *total elimination* seeking to reduce he entire SCC into a single node. They then inspect the composition of the computation along a dependence cycle to see if it matches the pattern of a scan. If either total elimination or the patern matching fails, their algorithm gives up, which prevents it from detecting many scans. For example, in the system of equations $x_i = x_{i-1} + y_{i-1}$, $y_i = x_i + a_i$, we can do a simple substitution of y in the definition of x and remove the definition for y, and this yields $x_i = 2x_{i-1} + a_i$. However, the total elimination will fail if there is no common vertex for all the circuits in the SCC. In general, this situation occurs when there is a mutual dependence, as in the example shown in section 5.3. Furthermore, they recognize the scans base on pattern matching, one of the common limitations for pattern matching method is that it will fail once the target becomes too complicated.

Redon and Feautrier [4] also present a method to schedule programs with reductions based on the recurrence equations. Although they assume an ideal (PRAM) machine model, they show that the generated schedules can be adapted to work on real parallel machines.

Matsuzaki et. al [6] proposed an algebraic approach for deriving reductions from recursive tree programs. They extended the matrix multiplication model to arbitrary semirings, which makes the systematic parallelization of reductions become more practical. Xu [7] demonstrated an automatic type-based system that detects parallelizability of sequential functional programs.

Han and Liu [18] describe a speculative parallelization method based on detecting *partial* reduction variables, i.e., those that either cannot be proven to be reductions, or that violate the requirements of a reduction variable in some way.

More recently, Sato and Iwasaki [9] developed a sophisticated and pragmatic system incorporating most of these algebraic approaches. Their system proposed many enhancements to existing analysis techniques to optimize the generated code, to detect hidden max operators from existing imperative codes, and an extension of the algorithms of Xu et. al [7] and Matsuzaki et. al [6] to detect semiring matrix operations from expressions.

All previous methods suffer from one limitation or the other. In particular, the techniques of Redon and Feautrier does not use any of the work on matrix operations on semirings, and the recent work on algebraic techniques [6, 7, 9] are limited to only detect reductions or scans in single loops, and do not detect multi-dimensional or lexicographic scans.

Our method based on the exact dependence analysis on systems of recurrence equations, detects both scans and reductions in the nested loops. It also deals with variables that have mutual dependence. The technique based on extracting matrix multiplication form makes our method more general and powerful. Overall, our method can handle a wider range of programs than the previous work.

8 Conclusion

We presented a method for automatically parallelizing a class of "inherently" sequential program. It is based on the classic recurrence parallelization technique of Kogge and Stone [5] but extended to nested loops, where the problems are more difficult. Our method extends the previous works, it handles a wider range of programs than them previous works. We can automatically detect reductions, arrays of scans, lexicographic scan, and scans with mutually dependent variables.

We implemented our method in a polyhedral program transformation and code generation system. One of our future work that remains to be done is the automatic parallelization part, although we present some optimizations in this paper, there should be more room for improvement. One limitation of our method is that every time the normalization step performs a splitting, we get a new graph. Since the convergence process is in general undecidable, we use a heuristic to stop the splitting. A better splitting method needs to be discovered.

References

[1] U. Bondhugula and J. Ramanujam, "Pluto: A practical and fully automatic polyhedral program optimization system," in *In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08)*, 2008.

[2] M. ParisTech, "Pips: Automatic parallelizer and code transformation framework," http://www.cri.ensmp.fr/pips/.

[3] G. E. Blelloch, "Scans as primitive parallel operations," *IEEE Trans. Comput.*, vol. 38(11), pp. 1526–1538, November 1989.

[4] X. Redon and P. Feautrier, "Scheduling reductions," in *Proceedings of the 8th international conference on Supercomputing*, ser. ICS '94. New York, NY, USA: ACM, 1994, pp. 117–125.

[5] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Comput.*, vol. 22(8), pp. 786–793, August 1973.

[6] Z. H. M. Morita and M. Takeichi, "Towards automatic parallelization of tree reductions in dynamic programming," in *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '06. New York, NY, USA: ACM, 2006, pp. 39–48.

[7] S.-C. K. D. N. Xu and Z. Hu, "Ptype system: A featherweight parallelizability detector," in *IN PROCEEDINGS OF 2ND ASIAN SYMPOSIUM ON PROGRAMMING LANGUAGES AND SYSTEMS (APLAS 2004), LNCS 3302.* Springer, LNCS, 2004, pp. 197–212.

[8] A. L. Fisher and A. M. Ghuloum, "Parallelizing complex scans and reductions," in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, ser. PLDI '94. New York, NY, USA: ACM, 1994, pp. 135–146.

[9] S. Sato and H. Iwasaki, "Automatic parallelization via matrix multiplication," in *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI 2011), to appear*, 2011.

[10] X. Redon and P. Feautrier, "Detection of recurrences in sequential programs with loops," in *Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, ser. PARLE '93. London, UK: Springer-Verlag, 1993, pp. 132–145.

[11] P. Feautrier, "Automatic parallelization in the polytope model," *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, pp. 79–103, 1996.

[12] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 7–16.

[13] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, 1991.

[14] D. Merrill and A. Grimshaw, "Parallel scan for stream architectures," *Technical Report CS2009-14*, pp. 373–381, 2009.

[15] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for gpu computing," *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pp. 97–106, 2007.

[16] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *J. ACM*, vol. 14(3), pp. 563–590, July 1967.

[17] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27(4), pp. 831–838, October 1980.

[18] W. L. L. Han and J. M. Tuck, "Speculative parallelization of partial reduction variables," pp. 141–150, 2010.