THESIS

AUTOMATIC PARALLELIZATION OF "INHERENTLY" SEQUENTIAL NESTED LOOP PROGRAMS

Submitted by

Yun Zou

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2011

Master's Committee:

Advisor : Sanjay Rajopadhye

Michelle Strout
A.P.Willem Bohm
F.Jay Breidt

ABSTRACT

AUTOMATIC PARALLELIZATION OF "INHERENTLY" SEQUENTIAL NESTED LOOP PROGRAMS

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, which normally occur within loops. Existing work that deals with complicated reductions and scans normally focuses on the formalism, not the implementation. To help eliminate the gap between the formalism and implementation, we present a method for automatically parallelizing such “inherently” sequential programs. Our method is based on exact dependence analysis in the polyhedral model, and we formulate the problem as a detection that the loop body performs a computation that is equivalent to a matrix multiplication over a semiring. It handles both a single loop as well as arbitrarily nested loops. We also deal with mutually dependent variables in the loop. Our scan detection is implemented in a polyhedral program transformation and code generation system (AlphaZ) and used to generate OpenMP code. We also present optimization strategies to help improve the performance of the generated code. Experiments on examples demonstrate the scalability of programs parallelized by our implementation.

TABLE OF CONTENTS

Chapter 1

Introduction

Multi-core and many core processors are the current trends in microarchitecture. Till the early 2000s, increasing the clock frequency of micro-processors gave most software a performance boost without any additional effort on the part of the programmers, or much effort on the part of compiler or language designers. However, due to power dissipation issues, it is no longer possible to increase clock frequencies the same way it had been. Increasing the number of cores on the chip has currently become the accepted way to use the increasing number of transistors available, while keeping power dissipation in control. Parallel programming is necessary to make efficient use of these architectures. Writing sequential programs is quite intuitive and natural. However, parallel programming by hand is a challenge for most programmers. Among several approaches to address this problem, one that is very promising but simultaneously very challenging is automatic parallelization.

Automatic parallelization is the process of automatically converting a sequential program to a parallel program that can directly run on parallel platform. This process requires no effort on part of the programmer in parallelization and is therefore very attractive. Some tools, like Polaris, PIPS [1], PLUTO [2], Omega [3], PoCC [4], have already been developed for automatic parallelization. Most auto-

matic parallelizers focus on distributing independent operations among processors or threads. Despite their many advantages, such compilers fail when there is a true (i.e., value based) dependence between operations. To overcome this limitation, there are two broad approaches: (1) Optimistic parallelization [5]. For effective optimistic parallelization, exploiting a higher abstraction to compiler is crucial [6]. (2) Analyze special cases. One such special case 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. Reductions and Scans occurs frequently in scientific applications, e.g., sequence analysis, sorting kernels, construction of trees and summed-area tables, etc [7].

1.1 What is a Scan and a Reduction?

A scan is an operation that takes a binary associative operator $\odot$ and a list of n expressions

$$[e_0, e_1, \ldots, e_{n-1}]$$

and returns a list

$$[e_0, e_0 \odot e_1, \ldots, e_0 \odot \cdots \odot e_{n-1}].$$

For example, if $\odot$ is addition, given a list of values

$$[3, 1, 2, 6, 5, 3, 1]$$

a scan would return

$$[3, 4, 6, 12, 17, 20, 21].$$

A reduction is similar to a scan, but it only returns the single expression $e_0 \odot e_1 \odot \cdots \odot e_{n-1}$. The reduction result for the above example is 21. In some sense, we can say that a reduction is a special case of a scan.

As is well known, with P processors, a scan or reduction with size n can be parallelized with a time complexity of $O(\frac{n}{P} + log_2 P)$ [7,8]. Since $P \ll n$ in most practical applications, this is linearly scalable (i.e., iso-efficient) parallelism. To take advantage of this source of parallelization, some work [9–15] has been done to recognize scans and reductions in programs.

1.2 Method Overview

The standard reduction and scan is characterized by an expression of the form $x_i = x_{i-1} \odot e_i$ that is repeatedly evaluated for a range of i. A compiler could recognize a scan by identifying expressions written in this form. However, many scans and reductions are not written in the standard form. In a seminal paper, Kogge and Stone [9] showed that if we can transform a loop body 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. The following loop computes x_i using expression $x_i = a_i \cdot x_{i-1} + b_i$.

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

The loop computes x_i using expression $x_i = a_i \cdot x_{i-1} + b_i$, 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 $X_0 = \begin{pmatrix} x_0 \\ 1 \end{pmatrix} = \begin{pmatrix} b_0 \\ 1 \end{pmatrix}$ is the initial value of 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 [13–16].

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 [15, 16]. Redon and Feautrier [10] presented a method using the polyhedral model [17] 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 expressions into such a form. Furthermore, works done previously about scans and reductions in arbitrary nested loop programs do not have real implementations.

1.3 Our Contribution

In this thesis, 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. A code generator is implemented for automatic parallelization of the detected scans. We have developed optimization techniques for the code that is parallelized by our method. We also developed an analysis model to anticipate

the ideal speedup.

1.4 Thesis Overview

The rest of this thesis is organized as follows: Chapter 2 gives some background about the polyhedral model and the definitions that are used in the rest of the thesis. Chapter 3 gives the examples that are used in rest of the thesis. Chapter 4 describes how to identify the scans and deduce the SVU form. Chapter 5 describes how to do parallelization and optimization about scan and reduction. In Chapter 6, we discuss related work, and finally, the conclusion and future work is presented.

Chapter 2

Background & 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, such as signal and image processing, bioinformatics, etc. The polyhedral model provides a powerful abstraction to reason about a class of loop programs, which are called Affine Control Loop (ACL) programs. ACL programs consist of arbitrary nested loop programs for which the loop bounds and array accesses are affine functions of outer loop variables and program parameters. The polyhedral model views an iteration of each statement as an integer point in a well-defined space called the statement's domain, which is called polyhedron. With such a representation for each statement and a precise characterization of statement dependences, it is very helpful in analyzing the program and doing optimization. Many authors show that polyhedral model is very useful in code generation and automatic parallelization [2, 4, 17, 18].

Domain The domain of a statement describes the iteration space in which the statement is defined and is represented by a set of linear inequalities. An ACL program consists of a sequence of statements, each statement is surrounded by

loops in a given order. In Example 1 described in the introduction, statement S is executed by the surrounding loop. Hence, the domain for the statement is

$$\{i|0 \leq i < n\}$$

i is the index for the iteration space.

Dependence Two iterations S_i and S_j are said to be dependent if they access the same memory location and one of them is a write. A true dependence exists if the source iteration writes to a memory location and the target reads the memory location. True dependences are also called read-after-write (RAW) dependences, or flow dependences. Similarly, if a write happens after the read to the same location, the dependence is called WAR dependence. WAW dependences are called output dependences. Read-after-read or RAR dependences are not actually dependences, but they still could be important in characterizing reuse. We say S_i depends on S_j, denoted as $(S_i \rightarrow S_j)$, if S_i is consumer and S_j is producer.

Dependence is an important concept in this thesis. Our detection relies on a compiler pass called dependence analysis. We handle a class of programs for which a preprocessing analysis can precisely identify all the the true dependences.

Uniform Dependence and Non-uniform Dependence A uniform dependence is a dependence where distance between the source and target is a constant vector. Such a dependence is also called a constant dependence and represented as a distance vector. In contract, if the distance is an affine function but not a constant, the dependence is called non-uniform.

Polyhedral Reduced Dependence Graph In our technique we represent the program dependences using a *Polyhedral Reduced Dependence Graph* (PRDG) [19].

Before we define PRDG, we first define *Reduced Dependence Graph* (RDG). In the RDG, each vertex represents a variable in the program, 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 $\{f, \mathcal{D}\}$, where f is the dependence function, and $\mathcal{D}$ is the domain where the dependence occurs, i.e., the polyhedral domain of the associated statement. Figure 2.1 shows the PRDG for Example 1.

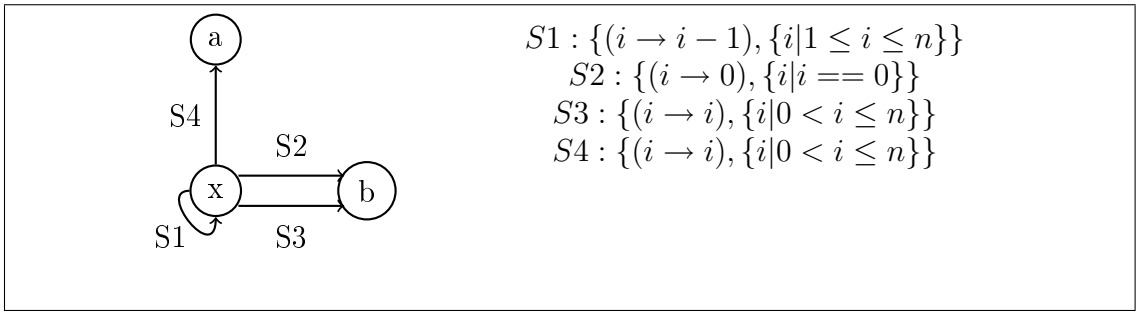

Figure 2.1: The PRDG for Example 1

2.2 Terminology

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

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. For example, in Example 1, the array variable x is a recurrence variable, since the computation of $x[i]$ is using the definition of $x[i-1]$.

Recurrence Equations A linear recurrence equation is defined as

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

Where z belongs to the domain of x, $\overrightarrow{d_i}$ is called a dependence vector. An ACL program can be transformed into a system of recurrence equations by exact data flow analysis [20].

In this thesis, we use the *Alphabets* notation to represent the extracted linear recurrence equations, since it is the input language of our system. Alphabets is an equational programming language, which evolved from the language ALPHA [21], a language originally used to describe and synthesize systolic systems. The mathematical representation of the computation in example 1 is

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

Written in Alphabets syntax:

$$\begin{aligned} x[i] = & case \\ & \{|i == 0\} : a[0]; \\ & \{|0 < i < n\} : a[i] \cdot x[i-1] + b[i]; \\ & esac; \end{aligned}$$

Semiring A two-operator algebraic structure $(R, \oplus, \otimes)$ is called a semiring [22], if R is the carrier set with two binary operators, $\oplus$ and $\otimes$ that satisfy the following properties:

- $(R, \oplus)$ is a commutative monoid with identity element $e_{\oplus}$:
  - Associativity: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
  - Commutativity: $a \oplus b = b \oplus a$
  - Identity element: $a \oplus e_{\oplus} = e_{\oplus} \oplus a = a$
- $(R, \otimes)$ is a monoid with identity element $e_{\otimes}$:
  - Associativity: $(a \otimes b) \otimes c = a \otimes (b \otimes c)$
  - Identity element: $a \otimes e_{\otimes} = e_{\otimes} \otimes a = a$
- Identity element $e_{\oplus}$ annihilates R with respect to $\otimes$:

– $l_{\oplus} \otimes a = a \otimes e_{\oplus} = e_{\oplus}$

For example, $(R, +, \cdot)$, $(R, \max, +)$ are all useful instances of semirings. Since matrix multiplication over a semiring is also associative, we can generalize Kogge and Stone's idea to handle scans and reductions over a semiring. The key idea of generalization of the detection technique is based on the algebraic properties of semiring.

State-Vector Update Form A system of recurrence equations is in SVU form iff it can be rewritten as follows:

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

Chapter 3

Examples

Before describing our detection technique, we present a number of examples that have various types of prefix computations and scans. These examples are repeatedly used in this thesis.

Example 2. The following example computes the *maximum segment sum (mss)*. It is known as a programming pearl [23]. 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 (there are thus, $\frac{n^2}{2}$. 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. For example, the *mss* of $[-1, 100, -2, 101, -3]$ is $100 - 2 + 101 = 199$.

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

The equivalent system of recurrence equations is:

$$
\begin{aligned}
x[i] &= case \\
&\quad \{|i == 0\} : a[0]; \\
&\quad \{|0 < i < n\} : \max(a[i], a[i] + x[i-1]); \\
&\quad esac; \\
t[i] &= case \\
&\quad \{|i == 0\} : x[0]; \\
&\quad \{|0 < i < n\} : \max(t[i-1], x[i]); \qquad\qquad esac; \\
m &= t[n];
\end{aligned}
$$

The operator max is a binary operator that 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 Fibonacci Sequence. The following program computes the first n Fibonacci numbers defined by, $fib[i] = fib[i-1] + fib[i-2]$.

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

The equivalent system of recurrence equations is:

$$
\begin{aligned}
f[i] = & \; case \\
& \{|i == 0\} : 0; \\
& \{|i == 1\} : 1; \\
& \{|1 < j < n\} : f[i-1] + f[i-2]; \\
& esac;
\end{aligned}
$$

This example has two recurrences on the left hand side of the computation. Current scan parallelizers do not support the detection of multi-recurrence on the left hand side of the computation.

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:

$$
\begin{aligned}
x[i,j] = & \; case \\
& \{|0 \leq i < n \&\& j == 0\} : a[i,0]; \\
& \{|0 \leq i < n \&\& 0 < j < m\} : x[i,j-1] + a[i,j]; \\
& esac;
\end{aligned}
$$

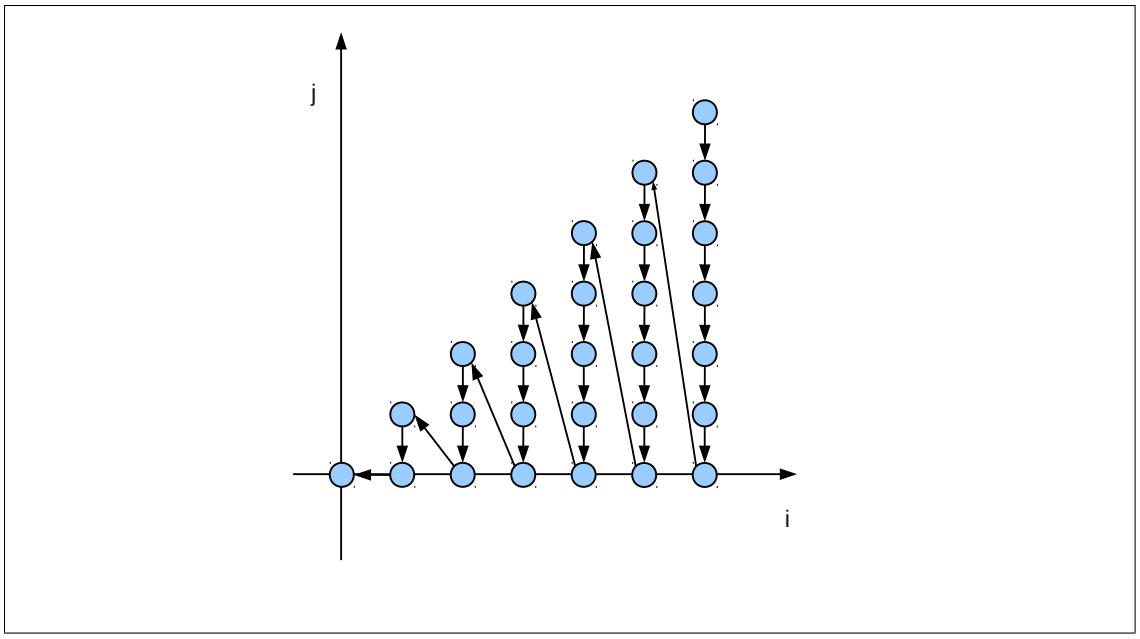

Figure 3.1: The computation domain and dependence for Example 5, where $n = 5$

Example 5. The following program does a lexicographical prefix sum computation on a triangular space with domain $\{i, j | 0 \leq i < n, 0 \leq j \leq i\}$. $x[i][j] = \sum_{k=0}^{i-1} \sum_{l=0}^{k} a[l][k] + \sum_{k=0}^{j} a[i][k]$. The computation domain and dependences of this example are shown in Figure 3.1.

```
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:

$$
\begin{aligned}
x[i,j] = {} & case \\
& \{|i == 0\} : a[0][0]; \\
& \{|0 < i < n, j == 0\} : x[i-1, i-1] + a[i, 0]; \\
& \{|0 < i < n, 0 < j \leq i\} : x[i, j-1] + a[i, j]; \\
& esac;
\end{aligned}
$$

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:

$$
\begin{aligned}
&x[i] = case \\
&\quad \{|i == 0\} : a[0]; \\
&\quad \{|0 < i < n\} : x[i-1] + y[i-1] + a[i]; \\
&\quad esac; \\
&y[i] = case \\
&\quad \{|i == 0\} : b[0]; \\
&\quad \{|0 < i < n\} : x[i-1] + y[i-1] + a[i]; \\
&\quad esac;
\end{aligned}
$$

Chapter 4

Detection of Scans

Our analysis, like Redon and Feautrier, relies on exact data-flow analysis [20] of an affine control loop program to extract a System of Recurrence Equations (SRE). Given such an SRE, the three major steps to detect scans are:

- Identify scan variables. Find the scan variables based on dependence analysis on recurrence variables.
- Extract coefficient matrix of the SRE.
- Transform the SRE into SVU form.

4.1 Identify Scan Variables

Scan variables are those recurrence variables that can be computed as a scan. Therefore, before identifying scan variables, we first obtain the recurrence variables. Given an SRE, we construct the PRDG for it, examine all the *self* dependences of a variable on itself, either directly or through a cycle in the PRDG. If such a cyclic dependence exists, we have identified a recurrence variable.

Theorem 1. Any variable that is involved in a *Strongly Connected Component* (SCC) of the PRDG is a recurrence variable.

Proof. For each vertex x in the SCC, let a be another vertex in the SCC. By definition of SCC, there exists a path from x to a and a path from a to x, so there is a path from x to x. In other words, there is a cyclic dependence on x. □

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.

Figure 4.1 shows the Strongly Connected Component for the PRDG of Example 1. The self dependence of x is $(i \rightarrow i-1)$, which is a uniform dependence, so x is a scan variable.

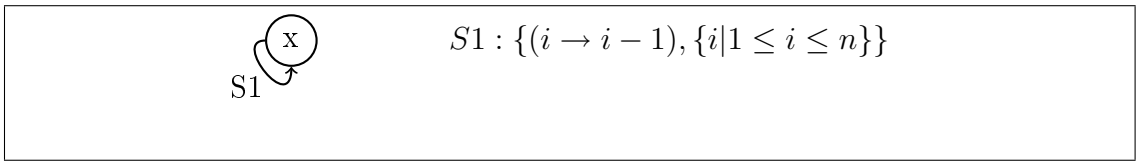

Figure 4.1: The Strongly Connected Component for Example 1

After identifying the scan variables, the final step is to extract the coefficient matrix for the scan variables. The detailed algorithm for extracting the linear terms for the matrix is shown in Figure 4.2, and the one for extracting the coefficient of the constant term is very similar, which is shown in Figure 4.3. Figure 4.4 gives an example about how the algorithm for extracting the linear terms works. The computation for the example is $x_i = a_i \cdot x_{i-1} + b_i \cdot y_{i-1} + c$, where x and y are both scan variables. It first constructs the expression tree of the right hand side expression of the equation, then visits the tree from bottom to top and applying the rules in the algorithm while visiting the tree.

The following section shows how to transform linear recurrence equations into SVU form.

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

Figure 4.2: 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 $\otimes$. True and False indicate whether the current term is involved with the scan variables.

4.2 State-Vector Update Form Transformation

The set of recurrence equations that we can solve need to satisfy the following constraints:

- The recurrence equation needs to be a linear recurrence equation.
- The dependence vectors on a recurrence variable need to be in the same direction. Two dependence vectors are in the same direction iff they have the same primitive vector. $\overrightarrow{d}$ is one of the dependence vectors of the recurrence variable, $gcd(\overrightarrow{d})$ returns the Greatest Common Divisor of the elements of $\overrightarrow{d}$, The primitive vector of $\overrightarrow{d}$ is computed as $\frac{\overrightarrow{d}}{gcd(\overrightarrow{d})}$. Take the examples in Figure 4.5. In (a), the dependence vectors on x are $(-1, -1)$ and $(-2, -2)$,

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

Figure 4.3: 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 $\otimes$. True and False indicate whether the current term is involved with constant coefficient.

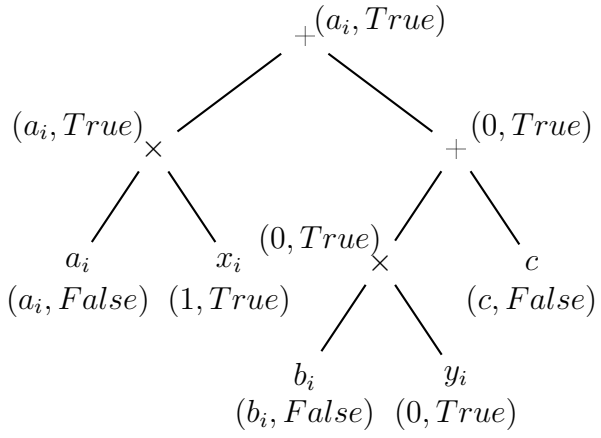

Figure 4.4: Extract coefficient of x

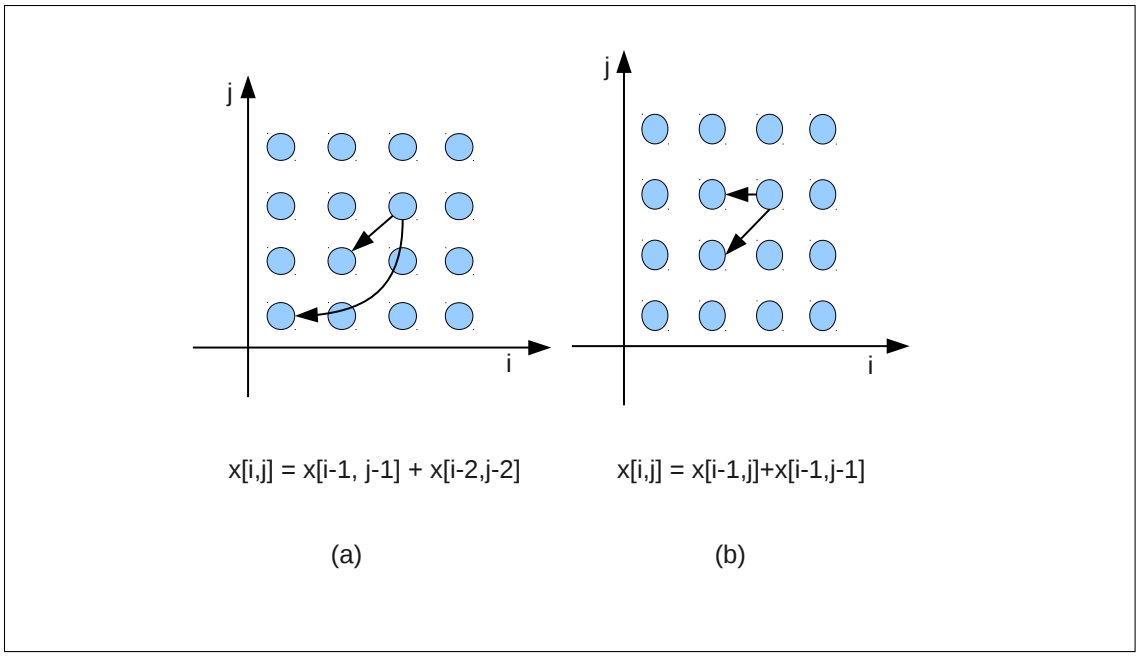

Figure 4.5: (a) Dependence vectors of x are in the same directions, (b) Dependence vectors of x in different directions

the primitive vector for these two vectors is $(1, 1)$. In (b), the dependence vectors on x are $(-1, 0)$ and $(-1, -1)$, the primitive vector for $(-1, 0)$ is $(1, 0)$, an $(1, 1)$ for $(-1, -1)$, so we say the dependences on x are not in the same direction.

With regard to the above constraints, we define the set of recurrence equations that we can solve in the following subsections.

4.2.1 First order recurrence equation

In this section we define a general class of first-order recurrence equations that can be transformed into SVU form.

Given a semiring $(R, \oplus, \otimes)$, a *first* order recurrence equation that can be solved by our technique is defined as follows:

$$z \in D : x_z = a \otimes x_{z-\overrightarrow{d}} \oplus b \tag{4.1}$$

where a and b are arbitrary expressions that do not involve the recurrence variable x. They 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 $a' \otimes x_{z-\overrightarrow{d}}$ on the right hand side combined using $\oplus$, involving *other* self dependences on x, but they all must have the same $\overrightarrow{d}$, 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 $gcd(\overrightarrow{d}) = 1$, then a matrix form can be extracted

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

If $\gcd(\overrightarrow{d}) = t > 1$. 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 $\overrightarrow{d} = t \cdot \overrightarrow{\delta}$, where $\gcd(\overrightarrow{\delta}) = 1$. We can transform equation (4.1) into a matrix form by adding $t-1$ temporary "accumulation variables." The matrix form for equation (4.1) is shown below.

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

4.2.2 M-th order recurrence equation

In an m-th order recurrence equation, the recurrence variable x occurs m times in the right hand side of equations, each time with a different dependence. It can be rewritten in the form,

$$z \in D : x_z = (a_1 \otimes x_{z-\overrightarrow{d_1}}) \oplus \cdots \oplus (a_m \otimes x_{z-\overrightarrow{d_m}}) \oplus b \tag{4.2}$$

where a is a list of m expressions and b is a single expression. The technique we described in the previous section applies for the first-order recurrence equations. However, a little manipulation of the previous method can often convert an m-th order recurrence equation into a first-order recurrence equation.

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 that for every self-dependence $\overrightarrow{d_i}$ on x, whether $\frac{\overrightarrow{d_i}}{gcd(\overrightarrow{d_i})}$ are the same. If this does not hold, the computation is not a scan. Let us assume that this holds, and the set $\{\overrightarrow{d_1}, \overrightarrow{d_2}, \ldots, \overrightarrow{d_m}\}$ are distinct self dependences in ascending order of the value of their respective gcds.

If $\gcd(\overrightarrow{d_1}) = 1$ and $\overrightarrow{d_i} = i \cdot \overrightarrow{d_1}(i > 1)$, then equation (4.2) can be transformed to the matrix form

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

The Fibonacci example described in the chapter 3 is a second-order recurrence equation, it can be transformed into the following SUV form

$$\begin{pmatrix} f_i \\ f_{i-1} \\ 1 \end{pmatrix} \leftarrow \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times_{\{\oplus,\otimes\}} \begin{pmatrix} f_{i-1} \\ f_{i-2} \\ 1 \end{pmatrix}$$

If $\gcd(\overrightarrow{d_1}) \neq 1$ or $\overrightarrow{d_i} = k \cdot \overrightarrow{d_1}(k \neq i)$, 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 (4.2) 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 not

all the directions are the same, for example, $x_{i,j} = x_{i-1,j} + x_{i,j-1} + x_{i-1,j-1} + a_{i,j}$. For this kind of dependence, we could try to obtain wavefront schedules using the classic polyhedral scheduling algorithms [17]. Since this is not the focus of this work, we do not discuss it here.

4.2.3 System of recurrence equations

In a system of recurrence equations, there is more than one recurrence variable and these recurrence variables are the nodes in a SCC. Example 6 in chapter 3 is such a system of recurrence equations.

We now show how to transform such 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 necessary.

In Example 6, there are two dependences on x, $(i \rightarrow i-1)$ and $(i \rightarrow i-1)$, one from x itself and one from y, 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:

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

4.3 Detection of Lexicographical Scan

For Example 5 in chapter 3, most scan parallelizers only detect the scan inside the innermost 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 exploiting linear semiring operations.

As we go up to lexicographical scan, we won't be able to represent the scan with only one direction, therefore, we use a set of directions to represent a lexicographical scan. For example, we use two directions to describe the scan in Example 5, $(0,1)$ and $(1,1)$, the first direction shows the direction of the inner most scan, the second direction is used to switch from the initial value of the previous slice of scan to the initial value of the next slice of scan.

Before we go to the details of how to detect lexicographical scan, first we need to define a variant of the lexicographic minimum. Let $\mathcal{D} \subseteq \mathbb{Z}^p$ be a convex polyhedral set and let e_i be vectors of the same $\mathbb{Z}^p$ space.

$$\underset{e_0,...,e_k}{\overset{\mathcal{D}}{\min}}(z) = z - \sum_{i=0}^{k} v_i e_i$$

Where $(v_1, ..., v_k = lexmax(\mu_0, ..., \mu_k | (\mu_0, ..., \mu_k) \in \mathbb{N}^k, z - \sum_{i=0}^{k} \mu_i \cdot e_i) \in \mathcal{D})$ The lexmax function represents the classical lexicographical maximum.

It is obvious that a lexicographical scans can not be detected directly. They can be found using mutli-stage detection, the basic idea is detect the innermost scan first, then try to merge the initial branches into the detected scan as much as possible. Detection of the innermost scan is the same as described above, so the main question is how to merge the initial value branches into the detected scan?

Assume a scan S is detected, $\mathcal{D}^s$ is the domain for the scan, $\mathcal{D}^g$ represents the initial domain of scan S, a set of vector $\{e_0, ..., e_k\}$ are the extracted directions for scan S. The jth initial branch can be merged into the detected scan s if it satisfies the following:

- $\mathcal{D}^g_j \cap \mathcal{D}^g \neq \emptyset$, where $\mathcal{D}^g_j$ is the domain of the jth initial branch.

- It is a recurrence equation of the same recurrence variable with the detected scan. The computation of the branch is in the form $f(v(I_j^0(z)), ..., v(I_j^m(z)))$, v represents the variable computed by the scan, I_j^i is the ith dependence function of the jth initial branch.

- Either a new direction or an old direction can be extracted from the initial branch.

- An SVU form can be extracted from the initial branch.

Let $\mathcal{D} = \mathcal{D}^S \cup \mathcal{D}^g$ A direction e can be extracted by resolving the following integer problem:

$$\forall A \in \mathcal{D}_j^g, e = z - \min_{e_0,...,e_k}^{D} (I_j^i(z))$$

If a constant direction e can be found for all the dependence functions in the jth branch, and an SVU form can be extracted, then this branch can be merged into the detected branch.

In example 5, a scan on variable x is detected first in the third branch with direction $(0, 1)$. $\mathcal{D}^S = \{0, i < n, 0 < j \leq i\}$, $\mathcal{D}^g = \{0 \leq i < n, j = 0\}$. The second branch with domain $\{i, j | 0 < i < n, j = 0\}$ computes the initial value of the detected scan, and it is a recurrence equation on x. There is one dependence function in the branch $(i, j \rightarrow i - 1, j - 1)$. The vector e is the solution of the following problem:

$$\forall z \in \{i, j | 0 \leq i < n, j = 0\} : e = z - \min_{e_0,...,e_k}^{D} (z + (-1, -1))$$

It is possible to transform this problem to a mere maximization problem under the following constraints:

$$\begin{cases} \mu > 0 \\ z - e = \begin{pmatrix} z_0 - 1 \\ z_1 - 1 \end{pmatrix} - \mu \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ z - e \in \{i, j | 0 \leq i < n, j = 0\} \end{cases}$$

Problems like this can be solved by some tools, like Parametric Integer Program (PIP) [24], PIP finds the lexicographic minimum of the set of integer points which lie inside a convex polyhedron which depends linearly on one or more parameters. In this example, the result for e is $(1, 1)$, it is a constant vector, so a new direction is extracted. Applying the coefficient matrix extraction algorithm to this branch, 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 is merged. After that, no more branches can be merged into the scan.

4.4 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, the computation of t is detected as a scan first, however, only the last value of t is used in the definition of m, so we say that the computation of m is a reduction.

Chapter 5

Code Generation

We integrated the scan detection technique into our polyhedral program transformation and code generation system AlphaZ. The framework of our scan parallelizer is shown in Figure 5.1. A C program is can be transformed into recurrence equations through GeCoS [25]. Our scan parallelizer takes recurrence equations as input, detects scans and generates parallelized scan code.

In this chapter, we described how to parallelize a scan or reduction. We also proposed some optimization strategies to improve the performance.

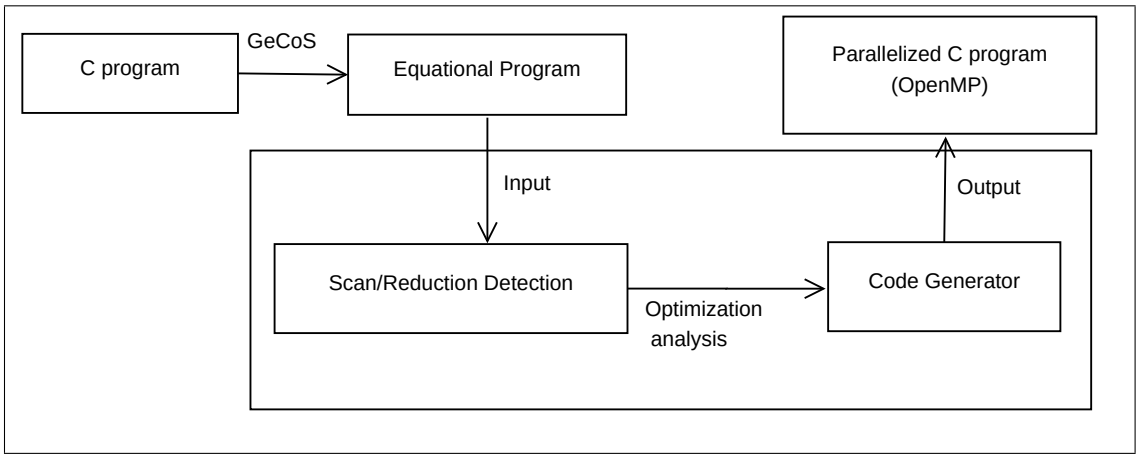

Figure 5.1: Framework for the scan parallelizer

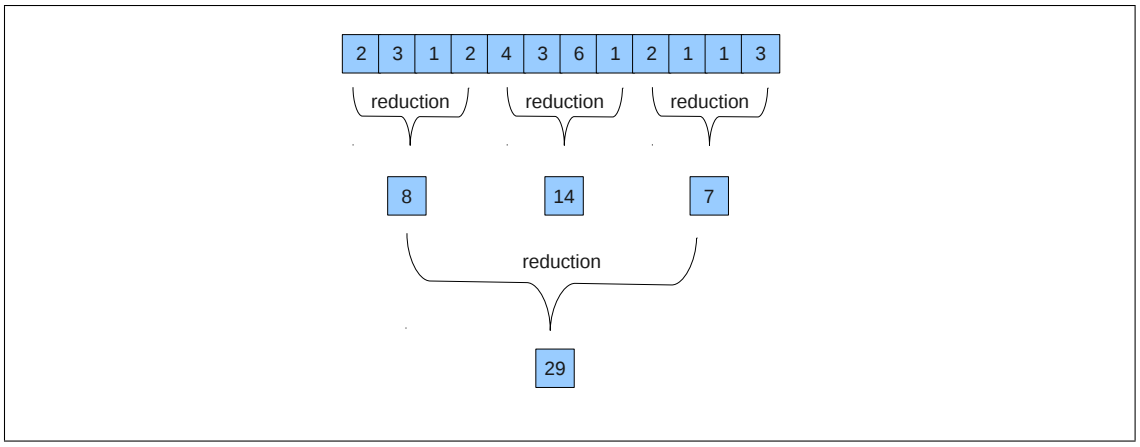

Figure 5.2: Example for reduction paralllelization, $p = 3$

5.1 Parallelization of Scan and Reduction

Much work has been done for the parallelization of scans and reductions [7, 16, 26, 27].

The parallelization of reduction is straightforward, and consists of two phases: local reduction and global reduction. Given an associative operator $\odot$ an a list of expressions $[e_0, e_1, \ldots, e_{n-1}]$, we want to do a reduction using p threads. 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 reduction on the p local reduction values. Figure 5.2 shows an example for how the reduction parallelization works. Furthermore, in OpenMP, there is even a direct pragma to do the reduction whenever the operator is one of the predefined ones.

The classic scan parallelization algorithm is shown in Figure 5.3. It contains three phases: local scan, global scan and final add. In the first phase, the n elements are distributed to the p threads, every thread performs a sequential scan on $\frac{n}{p}$ elements. Then a single thread performs a global scan on the last scan value of produced by each thread. Finally, each thread performs a local add to each

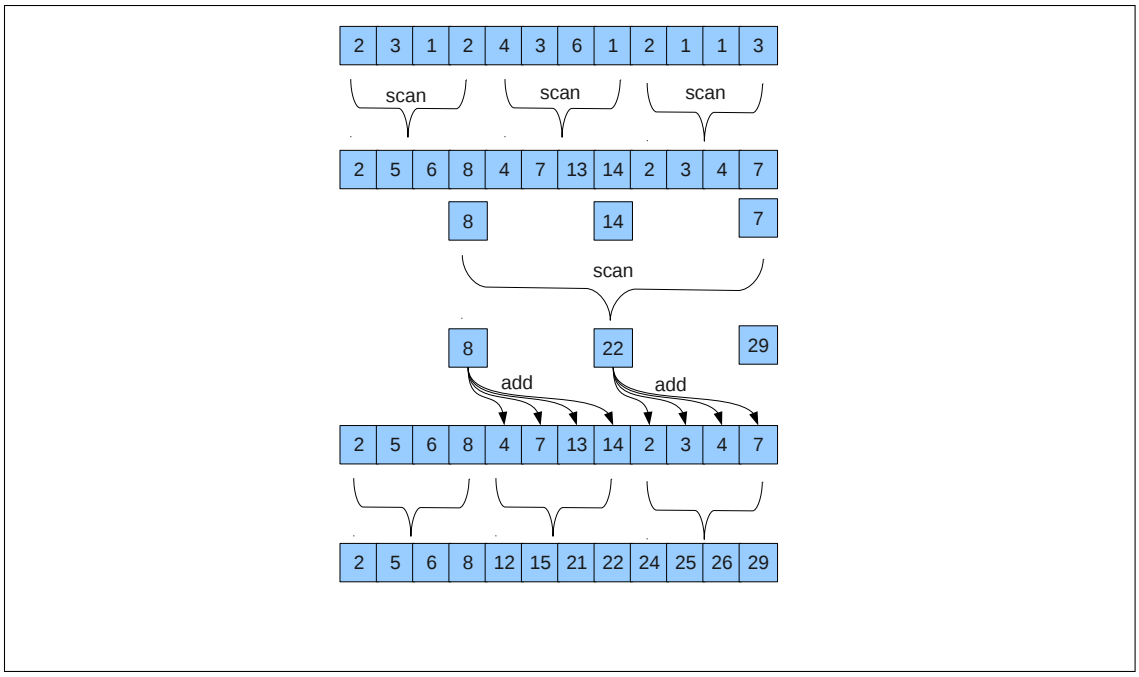

Figure 5.3: Example for classic scan parallelization, $p = 3$

value of its block.

Recently, Merrill and Grimshaw's work [26] points out that the parallelization of scan is actually I/O bound. The classic parallelization described above has $4n$ memory accesses. To reduce the memory access, they change the first phase to local reduction. They first let each thread do 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 initialization value form the global scan. This way, they save n memory access, since reduction requires only n memory accesses. Figure 5.4 gives an example of how Merrill and Grimshaw's scan parallelization works.

However, as we observed in the above algorithm, in phase one, the reduction value of the last thread is actually useless, it is not used in any computation, so instead of distributing the n elements to p threads, we divide the elements into $(p + 1)$ chunks. In the first phase, thread 0 does a scan on chunk 0 instead of a

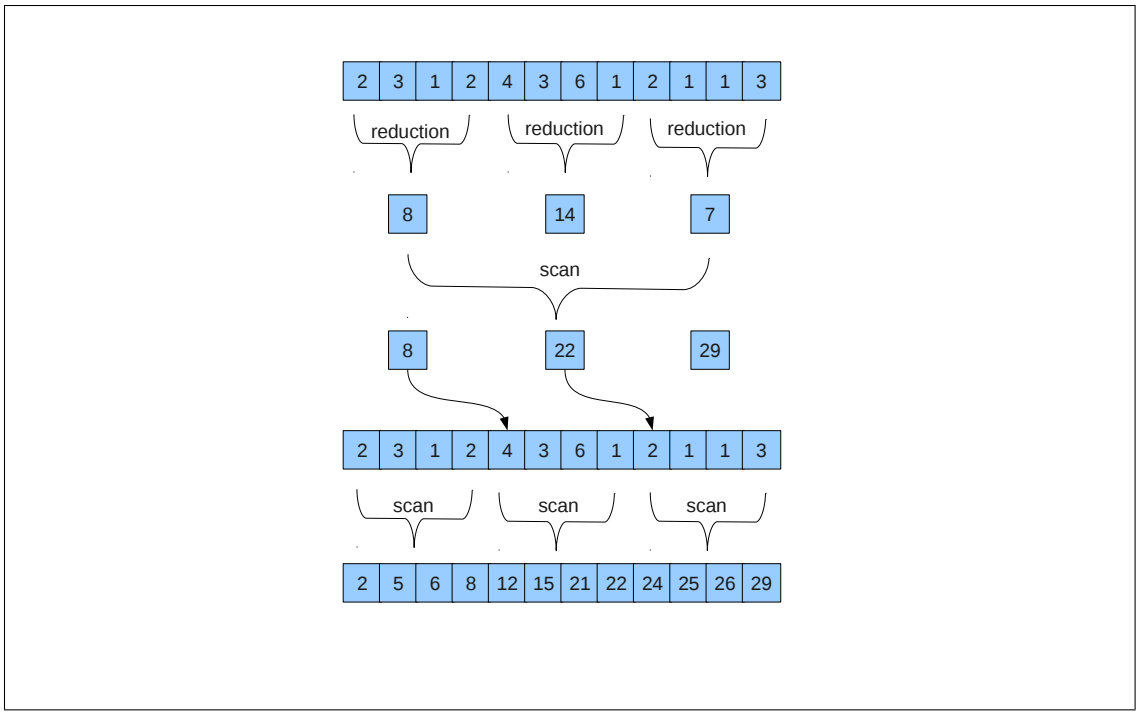

Figure 5.4: Example for Merrill and Grimshaw's scan parallelization, $p = 3$

reduction, every other thread performs a reduction on its own chunk—nothing has to be done for the last chunk. After the last value of scan and $(p-1)$ local reduction values are produced, a single thread performs a global scan of the p values. Finally, each thread does a scan on the last p chunks with the proper initialization value form the global scan. Figure 5.5 gives an example of how our algorithm works.

The example we used to illustrate the parallelization algorithm is a list of value, but the parallelized program does a parallelization for $X_i = (\prod_{j=1}^{i} A_j)X_0$, which mainly does a matrix-matrix multiplication. The original sequential program iterates over $X_i = A_i X_{i-1}$, which iterates over a matrix-vector multiplication. Compare the parallelized version with the original program, the parallelized version has $O(m)$ overhead, m is the size of the coefficient matrix. Hence, optimization is necessary to improve the performance of the parallelized code.

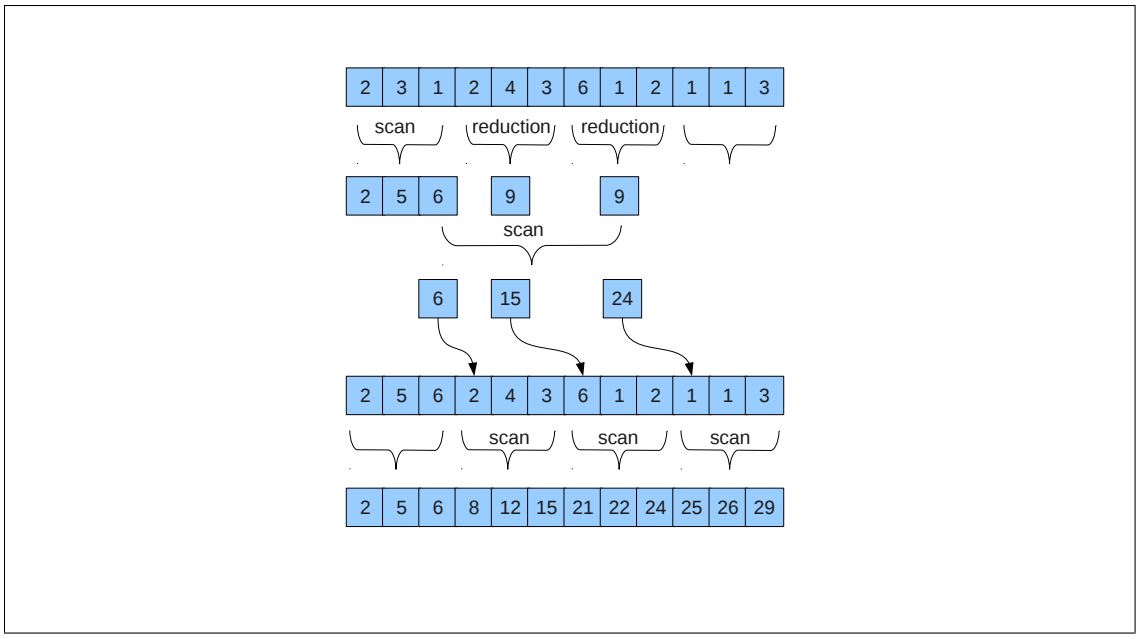

Figure 5.5: Example for our scan parallelization, $p = 3$

5.2 Optimization of Matrix Multiplication

Since most overhead comes from the matrix-matrix multiplication, the first optimization we considered is the optimization of matrix-matrix multiplication. Matsuzaki [13] 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 propagation. Based on this idea, we developed some optimization of matrix multiplication.

We use Z to denote $e_{\oplus}$, I to denote $e_{\otimes}$, C denotes the constant value and V denotes non-constant values. The semantics for the abstract operators $\oplus'$ and $\otimes'$ are shown in Figure 5.6. Let $M_i = \prod_{j=0}^{i} A_i$, $M_i = A_i \times_{\{\oplus',\otimes'\}} M_{i-1}$. In the optimization phase, we will iterate through the matrix multiplication about the abstract matrix until the same abstract matrix pattern appears.

Assume the following matrix is the initial matrix.

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

$\oplus'$	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 5.6: Semantics for $\oplus'$ and $\otimes'$ with the abstract values

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. However, in our case, this optimization is not enough to reduce the overhead of matrix multiplication. Here is a three order recurrence equation, $x_i = a_i \cdot x_{i-1} + b_i \cdot x_{i-2} + c_i \cdot x_{i-3} + d_i$, the coefficient matrix A_i is the following

$$\begin{pmatrix} a_i & b_i & c_i & d_i \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The abstract representation for this coefficient matrix is

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

Applying Matsuzaki's method, we iterate through the matrix multiplication until the same matrix pattern occurs. We get the following result

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

There will be twelve elements need to be computed during the matrix multiplication. Further optimization is needed to achieve comparable performance regarding to the original computation. The following result shows the first iteration over matrix multiplication with the original coefficient matrix.

$$M_1 = \begin{pmatrix} a_1 & b_1 & c_1 & d_1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_0 & b_0 & c_0 & d_0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Let $m_{1,0}$ be the first element of the second row of M_1

$$\begin{aligned} m_{1,0} &= 1 \times a_0 + 0 \times b_0 + 0 \times c_0 + 0 \times d_0 \\ &= a_0 \end{aligned}$$

The value of $m_{1,0}$ is a copy of value a_0, no other computation is needed. We achieve this optimization by unrolling the computation of matrix multiplication and remove unnecessary computations for each element.

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.

5.3 Load balance

Most scan parallelizers handle one dimensional scan, and it is relatively easy to distribute data evenly to each thread. However, in our case, since we deal with

lexicographical scan, it is possible that the domain of the scan is some shape that can not be distributed evenly as simple as one dimensional scan.

To address this problem, one important thing to know is the number of integer points in the polytope, or the volume of the polytope.

Clauss and Loechner [28] presented an automatic method for computing the number of integer points contained in a convex ploytope or in a union of convex polytopes. It computes the *Ehrhart* polynomial, which is a parametric expression of the number of integer points. Here we use their Ehrhart polynomial result.

Assume the domain of a scan is a parameterized polytope $\mathcal{P}$, the Ehrhart polynomial of the polytope is represented as $V(\mathcal{P})$. Given p threads, ideally, each thread should get $\frac{V(\mathcal{P})}{p+1}$ work. To do this, we try to divide the polytope along the outer dimension, we add one more constraint to the outer dimension, so that the outer dimension is parameterized by x, the new ploytope's Ehrhart polynomial is $V(\mathcal{P}(x))$. The index value of thread i can be computed by solving the following problem

$$\frac{V(\mathcal{P})}{(p+1)} \cdot (i+1) = V(\mathcal{P}(x))$$

We use the Newton-Raphson method [29] to compute the value of x, then the ith chunk has the outermost index in the range of $f(i)$ to $f(i+1)$.

However, there is still some imbalance in the computation. Assume that we have a scan S, the volume of the scan domain is N, the overhead of the matrix-matrix multiplication after optimization is β. Now we have p threads, what is the ideal speedup? As we described before, in phase one, thread 0 is doing a scan, which is a matrix-vector multiplication, and the other threads are doing reduction over matrix-matrix multiplication. If we divide the work evenly into $(p+1)$ chunks, the other threads which is doing a reduction over matrix-matrix multiplication will do β times more work than thread 0, and there is an imbalance. To get the best

performance, we want thread 0 to do the same work as the the other threads, assume thread 0 gets n work, each of the other thread gets $\frac{N-n}{p}$ work. We want

$$n = \beta \cdot \frac{N-n}{p}$$

Then we get

$$\begin{aligned} n \cdot p &= N \cdot \beta - n \cdot \beta \\ n \cdot (p+\beta) &= N \cdot \beta \\ n &= N \cdot \frac{\beta}{p+\beta} \end{aligned}$$

When $n = N \cdot \frac{\beta}{p+\beta}$, the parallel execution time P_T is

$$\begin{aligned} P_T &= n + \frac{N-n}{p} \\ &= n + \frac{N-n}{p} \\ &= n \cdot \frac{1+\beta}{\beta} \\ &= N \cdot \frac{\beta}{p+\beta} \cdot \frac{1+\beta}{\beta} \\ &= \frac{(1+\beta) \cdot N}{p+\beta} \end{aligned}$$

The best sequential time is N, the ideal speedup will be

$$\begin{aligned} speedup &= \frac{N}{\frac{(1+\beta)\cdot N}{p+\beta}} \\ &= \frac{p+\beta}{1+\beta} \end{aligned}$$

5.4 Experimental Validation

The technique presented here has been implemented and used to generate OpenMP code. The structure of the code generated by the code generator is shown in Figure 5.7.

```
#pragma omp parallel for
{
    Do index computation using Newton-Rapson method
    #pragma omp barrier
    Initialize the begin and end variables of each thread block
    if(thread_num == 0)
    {
        Scan computation parameterized with begin and end
    }
    else
    {
        Reduction computation parameterized with begin and end
    }
    #pragma omp barrier
    #pragma omp single
    {
        Scan computation on the reduction result
    }
    #pragma omp barrier
    Initialize the begin and end variable of the each thread block

    Compute the new initialize value for each thread

    Scan computation parameterized with begin and end
}
```

Figure 5.7: The structure of the generated code

To confirm the efficiency and scalability of the parallelization algorithm, we ran our code generator on the following examples.

Polynomial Scan The polynomial scan computes $x[i] = c \cdot x[i-1] + a[i]$, It computes polynomial evaluation through the Horner scheme. The parallelized computation computes a scan on a $(2,2)$ matrix over semiring $(\mathbb{R}, \times, +)$. The test size for this problem is 2^{29}.

Convolution The convolution computes $x[i] = a0 + a1 \cdot x[i-1] + a2 \cdot x[i-2] + a3 \cdot x[i-3] + a4 \cdot x[i-4]$. It is a forth-order recurrence equation. It is parallelized with a $(5,5)$ matrix over semiring $(\mathbb{R}, \times, +)$. The test size for this problem is 2^{29}.

Lexicographical Scan on a Rectangle space Here we does a lexicographical scan on rectangle space. The main computation is $x[i,j] = A[i,j] \cdot x[i,j-1] + B[i,j]$. It is parallelized with a $(2,2)$ matrix over semiring $(\mathbb{R}, \times, +)$.

Lexicographical Scan on a Triangle space This is example has the same domain with example 5 in chapter 3. The main computation is the same with the above example $x[i,j] = A[i,j] * x[i,j-1] + B[i,j]$.

All the testing is done on a machine equipped with one XeonE5520 (8 cores; 2.26 GHz) and 12 GB memory. The running environment is Fedora 15, each program is compiled using icc 12.0.3 with the O3 optimization. For speedup, we compare with the sequential code without matrix multiplication. The results are shown in Figure 5.8 and Figure 5.9.

Figure 5.8 shows the speedup for horner scheme, lexicographical scan on a rectangle space and triangular space. Since the computation for these four programs are very similar, the overhead of the matrix computation are the same and they have the same ideal speedup. Horner scheme and the lexicographical scan on rectangle space are very close to the ideal speedup. However, the performance for the triangle lexicographical scan without balancing the load are very poor. After we fix

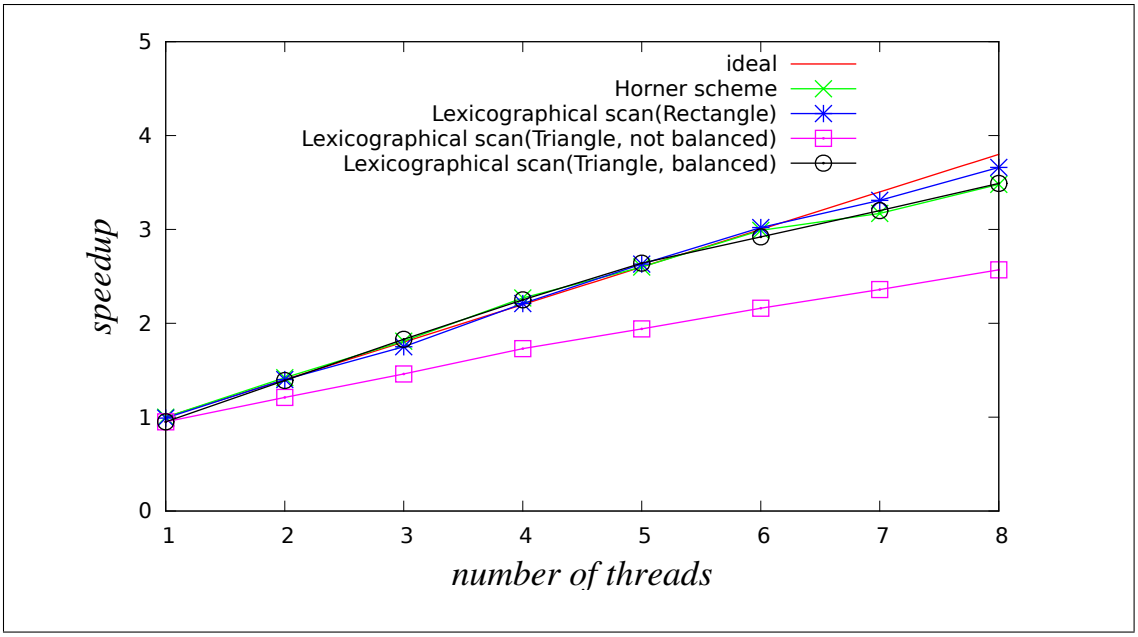

Figure 5.8: Speedup for the testing program

the load balancing problem, it matches the ideal speedup. So the load balancing problem is important to get good speedup.

Figure 5.9 shows the result for convolution. The speedup for the original convolution is far below the ideal. In convolution, the parallelized program does a computation over a 5 by 5 matrix, so the overhead of the reduction will be relatively larger than the above programs, then the load balance problem for the first chunk is obvious. We can see form the result that after we fixing the load balance problem for the first chunk, the speedup gets very close to ideal.

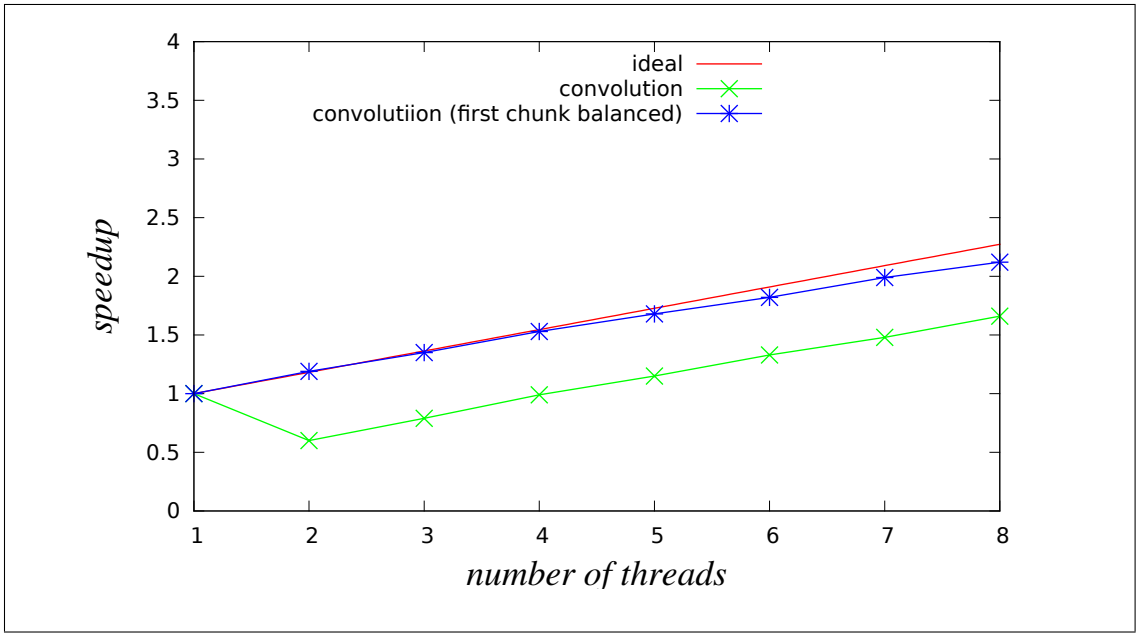

Figure 5.9: Speedup for convolution

Chapter 6

Related Work

The parallel implementation of recurrence equations was first discussed by Karp, Miller and Winograd [30]. They treated program dependences as inviolate constraints that any parallelization had to respect. Later, in a seminal paper, Kogge and Stone [9] 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 [31] descibed an efficient, general-purpose circuit for scan operations. Blelloch [7] also his text 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 their main detection 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 the 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 pattern matching fails, their algorithm gives up, which prevents it from detecting many scans. For example, in the system of equations

$$\begin{aligned} x_i &= x_{i-1} + y_{i-1} \\ y_i &= x_i + a_i \end{aligned}$$

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. Furthermore, they recognize the scans based on pattern matching, and one of the common limitations for pattern matching method is that it will fail once the target becomes too complicated.

Redon and Feautrier [8] 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 [13] 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 [14] demonstrated an automatic type-based system that detects parallelizability of sequential functional programs.

Han and Liu [12] 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 [15] 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 [14] and Matsuzaki et. al [13] 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 [13–15] 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.

Chapter 7

Conclusion

We presented a method for automatically parallelizing a class of "inherently" sequential programs. It is based on the classic recurrence parallelization technique of Kogge and Stone [9] but extended to nested loops, where the problems are more difficult. Our method extends the state of art, it handles a wider range of programs than the previous works. We can automatically detect reductions, arrays of scans, lexicographic scan, and scans with mutually dependent variables.

We implemented our scan detection method in a polyhedral program transformation and code generation system. We also developed a code generator to generate parallel code for the detected scan.

However, there is some future works remains to be done:

- Scans and reductions in other forms. There are still some kind of hidden scans and reductions that we do not handle at that point, for example, reductions written in a tree way.

- Detect scans in the outer loop. Such as the matrix power example $B = A^K$, the inner loop does a matrix matrix multiplication. A simple parallelization is parallelizing the matrix matrix multiplication. However, sometimes the matrix size is too small to gain benefit from this kind of parallelization, and k is fairly large. Then we can try to parallelize the outer loop as a scan.

- Optimization of Matrix Multiplication. Currently, we assume that the optimization analysis given, we take the final stable matrix and generate the corresponding code.

- More experiments have to be done. Our current examples for the lexicographical graphical scans are constructed examples, some experiments about the real world applications need to be done to prove the value of our work.

REFERENCES

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

[2] 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.

[3] T. Ohl, "O'mega: An optimizing matrix element generator," 2000.

[4] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan, "Combined iterative and model-driven optimization in an automatic parallelization framework," in *Conference on Supercomputing (SC'10)*. New Orleans, LA: IEEE Computer Society Press, Nov. 2010.

[5] D. R. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 404–425, July 1985.

[6] M. Kulkarni, K. Pingali, and Walter, "Optimistic parallelism requires abstractions," *SIGPLAN Not.*, vol. 42, pp. 211–222, June 2007.

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

[8] 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.

[9] 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.

[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] T. Suganuma and Komatsu, "Detection and global optimization of reduction operations for distributed parallel machines," pp. 18–25, 1996.

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

[13] 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.

[14] S.-C. K. D. N. Xu and Z. Hu, "Ptype system: A featherweight parallelizability detector," in *in Procedings of 2nd asian symposium on programming languages and systems (APLAS 2004), LNCS 3302*. Springer, LNCS, 2004, pp. 197–212.

[15] 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.

[16] 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.

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

[18] 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.

[19] A. Darte, Y. Robert, and F. Vivien, *Scheduling and Automatic Parallelization*, 1st ed. Birkhauser Boston, 2000.

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

[21] H. Le Verge, C. Mauras, and P. Quinton, "The alpha language and its use for the design of systolic arrays," *J. VLSI Signal Process. Syst.*, vol. 3, pp. 173–182, September 1991.

[22] S. Aji and R. McEliece, "The generalized distributive law," *Information Theory, IEEE Transactions on*, vol. 46, no. 2, pp. 325 –343, mar 2000.

[23] J. Bentley, *Programming pearls*, ser. ACM Press Series. Addison-Wesley, 2000.

[24] P. Feautrier, J. Collard, and C. Bastoul, "Solving systems of affine (in)equalities," PRiSM, Versailles University, Tech. Rep., 2002, related to the PIP/PipLib tool.

[25] INRIA, "Gecos: Generic compiler suite," https://gforge.inria.fr/projects/gecos/.

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

[27] 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.

[28] P. Clauss and V. Loechner, "Parametric analysis of polyhedral iteration spaces," *J. VLSI Signal Process. Syst.*, vol. 19, pp. 179–194, July 1998.

[29] T. J. Ypma, "Historical development of the newton-raphson method," *SIAM Rev.*, vol. 37, pp. 531–551, December 1995.

[30] 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.

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

[32] X. Redon and P. Feautrier, "Detection of scans in sequential programs," 1997.

[33] T. T. A. Nisto, W. Chin and N. Tapus, "Optimizing the parallel computation of linear recurrences using compact matrix representations," *J. Parallel Distrib. Comput.*, vol. 69(4), pp. 373–381, April 2009.