THESIS

CODE GENERATION IN ALPHAZ

Submitted by

Pradeep Srinivasa

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2011

ABSTRACT

CODE GENERATION IN ALPHAZ

Computer architecture technology is evolving rapidly. Many of the programs written for a specific architecture are not very useful when a new architecture evolves. They have to be either modified or rewritten to suit the new architectures. Instead one can write a high level program and feed this to a tool which can produce code for different architectures. AlphaZ is such a tool which takes a high level program and helps us to analyze, transform and generate code for different architectures.

In this thesis, we develop a code generation framework in AlphaZ, which takes equations as programs called alphabets program. Alphabets is a high level abstraction language which allows us to write equational programs. Equational programs consists of a set of equations along with their associated domains. We describe how code is generated in our code generation framework by taking an Alphabets program and the necessary target mapping specification. We illustrate how different code generators can be developed by extending the existing modules in our code generation framework.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Currently, most of the Multi-core architectures are based on the shared memory model. To exploit parallelism from these architectures one needs to write an efficient parallel program, and this is not an easy task. Even if someone writes code for a given architecture, the same code cannot be used for other architectures and it has to be rewritten to suit the new architecture. Instead of this, if we have a tool which generates code for any architecture by taking a high level program (as abstract as equations) and some additional specification of the architecture, one will opt for writing those high level programs. These high level programs need not be changed to produce different code to suit different architectures, only the specification has to be modified to suit the architectures. Therefore, we need code generators which can generate code for different architectures, without much effort from the user. In this thesis, we have developed a code generator framework and a number of different code generators as a part of our research tool called AlphaZ.

AlphaZ is a system, which is based on a mathematical framework called polyhedral model. By working within this formal model, we can benefit from its mathematical properties. In this model, we represent computations as a set of equations along with their associated domains. These equations along with their domain forms the input called the Alphabets program for our system. AlphaZ transforms the alphabets program to a intermediate representation (IR). This representation allows us to analyze, transform and generate code as per the specification. The transformations can be specified as affine functions and these transformations can represent either a schedule, a processor allocation and/or a memory allocation specification. These can be given by the user and there is a distinction between all these specifications, this allows us to keep one specification constant and experiment with the other specifications. This provides us a platform where we can experiment and generate code that is specific to the architecture and will allow us to eventually develop a tool which automatically tunes or parallelizes an application for the given architecture.

In order to experiment with the various architectures we need to have various code generators that produces efficient code, so we provide in this thesis, a code generator framework and a number of different code generators. In this framework, one can easily implement various code generators by extending the existing modules. Using this code generator framework, we have developed two code generators: a scheduled code generator and a shared memory parallel code generator. The scheduled code generator takes a sequential schedule and produces C code according to the given schedule. The shared memory parallel code generator is an extension of the scheduled code generator and it takes a parallel schedule that may be interleaved (parallel and time loops can be in any order) and produces OpenMP parallel C code.

The remaining chapters of this thesis are organized in the following way. Chapter 2 describes the necessary background on polyhedral model, the architecture mapping specification and the related work. Chapter 3 describes the AlphaZ system, the transformation engine, the representation of schedule, processor allocation and memory allocation in AlphaZ, and some of the transformations that are used in this work. Chapter 4 describes with an example the generated code, the code generator framework and the code generators that are developed in this framework. Chapter 5 describes the handling of reductions in the code generators. Chapter 6 concludes this work and explains the future work.

Chapter 2

Background

In this chapter, we describe the necessary background about polyhedral model, scheduling, and the various tools that we use. We also give a brief description of the language directives used to produce parallel code. Then we describe the work that has been done by similar tools elsewhere.

2.1 The Polyhedral Model

Many scientific applications spend most of their time in nested loops. In order to extract parallelism or improve performance one needs to optimize or transform these loops. The polyhedral model is a mathematical framework for such nested loop optimizations and transformations. For example, composing a number of optimizations is easy if the optimizing transformations satisfy closure properties.

In the polyhedral model, we represent each instance of each statement in a loop program as an *iteration point*, in a space called *iteration domain* of the statement. The iteration domain is described with a set of linear inequalities forming a convex polyhedron. The dependencies in the program are expressed as affine functions. Loop nests that are called Affine Control Loops (ACLs), also called SCoPs (Static Control Parts) [3] can be represented in this model.

As an example, consider the code in 2.1 (left). This program has a 2D iteration

```
for(i=1;i<5;i++) {
 for(j=1;j<5;j++) {
  A[i][j] = A[i-1][j] +
                A[i][j-1];
 }
}
```

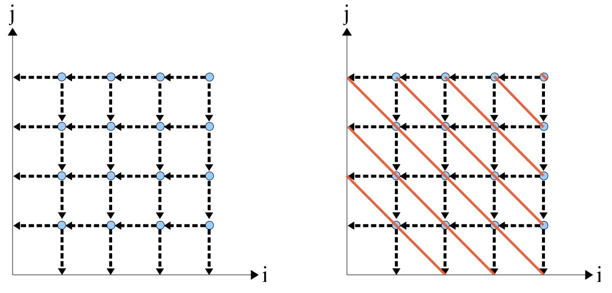


Figure 2.1: A simple doubly nested loop and its iteration space, together with a possible wavefront parallelization (diagonal lines).

space bounded by four inequalities represented as $\{i, j \,|\, 1 \leq i < 5 \wedge 1 \leq j < 5\}$. The two dependencies are functions $(i, j \rightarrow i-1, j)$ and $(i, j \rightarrow i, j-1)$. The iteration space of this loop nest is illustrated in 2.1. Note that these iteration points are enough to describe what is computed in the program. However they are not enough to reproduce the loop nest shown in 2.1 and let alone to generate code which can be run in parallel. We need to have some more information to generate code which can be run in parallel or in sequential.

2.1.1 Schedule

A schedule is an affine function θ, which determines precisely a logical time stamp for the execution of each instance of the statement. In our example a simple scheduling function $\theta = (i, j \rightarrow i, j)$ represents the sequential lexicographical execution of the original loop nest. Many loop transformations can be expressed as schedules. For example, loop permutation to permute i and j loops in the original program, corresponds to a different schedule $\theta = (i, j \rightarrow j, i)$.

2.1.2 Processor allocation

Processor allocation is also an affine function λ, which specifies which processor executes an instance of a statement. In our example $\lambda = (i, j \rightarrow j)$ is a valid processor allocation if the schedule is given by $\theta = (i, j \rightarrow i + j)$ as shown in the 2.1 (right).

2.1.3 Memory allocation

This specifies where the result of each instance of the statement is stored in memory. This allows us to find a memory allocation which is efficient and optimal. In our example, the simple memory allocation can be given as $(i, j \rightarrow i, j)$ for the variable A.

2.2 ClooG

ClooG [Chunky Loop Generator] [1] is a loop generator that produces loops in our code generators. ClooG is an open software and library to generate loops by scanning the polyhedra. The input to ClooG is a set of domains given by a set of affine constraints, the context and scattering functions. The context consists of the language used and the constraints on the parameters in the program. ClooG takes the set of domains and produces code that scans each integral point in the union of the domains. The scanning order can be specified through scattering functions and therefore schedule in itself can be a scattering function.

2.3 OpenMP

OpenMP [Open Multi-Processing `http://openmp.org`] is an application programming interface (API) to support shared memory parallel programming in C/C++ and Fortran on various architectures including Linux and Windows platform. It is a simple, portable and scalable model which provides simple API's to develop parallel programs. It consists of set of compiler directives, library routines and some environment variables or run-time support which help in the execution of the program.

OpenMP achieves parallelism by the use of Multi-threading using the fork-join protocol, where a master thread forks a number of slave threads and the given task is divided among the threads. These threads run concurrently, and once the execution is done the threads join back to the master thread. The task that has to be parallelized is annotated by a set of compiler directives. OpenMP gives a platform where we can write code which can range from fine grained parallel code to coarse grained parallel code.

2.4 Related Work

There are many tools existing which uses the polyhedral model. In this section we describe the various tools like MMAlpha, Pluto, Graphite and PoCC available which is based on the polyhedral model.

2.4.1 MMAlpha

MMAlpha [6] is a programming environment which is based on the polyhedral model with similar goals to the AlphaZ system. It is used to design parallel architectures like systolic array architecture from a set of recurrence equations specified as an Alpha program.

Alpha is a functional language which involves equations along with the variables with their domains. There are two significant differences between MMAlpha and the AlphaZ system. MMAlpha emphasizes on hardware synthesis, and does not target other platforms and it is based on Mathematica which has a high learning curve, especially for developers.

2.4.2 Pluto

Pluto [2] is an automatic parallelization tool that is based on the polyhedral model. It is a fully automatic polyhedral source-to-source program optimizer tool that takes C loop nests and generate tiled and parallelized code. It uses the polyhedral model to explicitly model tiling to find good ways of extracting coarse grained parallelism and locality. Because it is automatic, it follows a specific strategy in choosing transformations. It first chooses a set of tiling hyperplanes. Then it transforms the code so that in the transformed space, these hyperplanes are reflected in a new set of fully permutable indices. In addition, it seeks if possible, a set of schedule hyperplanes that lead to synchronization-free parallelism. Finally, it generates sequential and OpenMP parallel code.

Pluto is based on partitioning-based approaches, where they identify a set of sequential and parallel loops which can be run with minimum synchronization where as our framework is based on schedule/allocation-based approach. The focus of our AlphaZ system is to provide an environment to try many different ways of transforming a program. We provide a code generation framework where the user can play with the different types of schedules and target different architectures. One could use AlphaZ to find a good sequence of transformations, that could be turned into an automatic optimizer, and write code generators by extending the existing code generators.

2.4.3 Graphite

Graphite [10] is an optimization framework for high-level loop nest optimizations that is being developed as a branch of GCC. Its emphasis is to extract ACLs from programs that GCC encounters, which is significantly more complex than what research tools are expected to handle, and perform loop optimizations that are known to be beneficial. Graphite takes C programs as input whereas we take equational programs and it helps us in utilizing the power of polyhedral model.

2.4.4 Polyhedral Compiler Collections

Polyhedral Compiler Collections [11] (PoCC) is a framework for source-to-source program optimizations, designed to combine multiple tools that utilize the polyhedral model. Like AlphaZ, they also seek to provide a framework for developing tools like Pluto, and other automatic parallelizers. Our framework starts from equational representation so that we have a larger space to explore.

Chapter 3

AlphaZ System

In this chapter we explain the AlphaZ system. AlphaZ is a system, which is based on a mathematical framework called polyhedral model. The input to this system is an alphabets program. Alphabets is an equational language based on Alpha [9].

AlphaZ system consists of three main components: a program transformation engine, a data-structure called the TM (Target Mapping) and a code generator framework. The latter two are closely linked by a module called the Verifier, that checks the legality of the specified schedule, and memory and processor allocations. In addition, AlphaZ also provides a shell based interactive interface designed for users of the system. Such an environment where users can incrementally apply transformations and see the transformed equations is important for manual exploration of the space of valid transformations.

Section 3.1 describes the transformation engine, the representation of the IR and the back-end engine which provides the basic polyhedral operations/transformations. Section 3.2 describes the data-structure for the time, processor, tiling and memory specification. Section 3.3 describes the generalized change of basis transformation that is used in the code generator framework.

3.1 Transformation Engine

The transformation engine consists of a back-end engine called COREquaitons [5], a set of transformations and a state handling mechanism. The basic unit in the system is a *domain qualified statement*, namely an equation, associated with a polyhedral iteration space called its *domain*. A program is a collection of such equations. Each statement has an expression (the right hand side, rhs of the assignment). Data reordering is a first-class entity in our system, therefore there is no left-hand side (lhs) to the domain qualified statements, and the expressions on the rhs also do not refer to memory location but rather to instances of other domain-qualified statements. These recurrences can be obtained from an affine control loop through an exact data-flow analysis [4].

The Intermediate-Representation (IR) is essentially a list of polyhedral equations and an equational language called Alphabets is used at the core of the system. This separates computation from schedules and memory allocation. It is derived from an earlier language called Alpha [9] and it includes non-affine dependence functions, and iterative (e.g., while) computations. An informative description of Alpha is given by Wilde [12]. Reductions were added to the Alpha language by Le Verge in 1992 [8].

A simple alphabets program consists of the following.

1. A system name
2. A list of parameters
3. A list of input variable declarations
4. A list of local variable declarations
5. A list of output variable declarations
6. A list of equations defining the output and local variables

Let us consider the matrix-matrix multiplication (MMM) as shown in 3.1(left).

```
for(i=0;i<N;i++){
 for(j=0;j<N;j++){
  c[i][j] = 0;
  for(k=0;k<N;k++){
   C[i][j] += A[i][k] *
                B[k][j];
  }
 }
}
```

```
\\Alphabets program.

affine MMM {N|N>0}
given    int A,B {i,j | 0<=i<N && 0<=j<N};
returns  int C {i,j | 0<=i<N && 0<=j<N};
through
      C = reduce(+, [k], A[i,k] * B[k,j]));
.
```

Figure 3.1: Matrix-Matrix Multiplication.

The alphabets program consists of a system name MMM along with the parameter N with a constraint $N > 0$. A list of input variables A, B and an output variable C. MMM is a classic example which explains the reduction along the k^{th} index. Therefore the equation for variable C can be expressed using a reduction as shown in 3.1(right).

The alphabets program is parsed by the back-end engine to construct an AST with domain and equation information. An external back-end engine called COREquations [5] is used to perform basic polyhedral operations, such as change of basis, substitute by definition, cut, image/preimage of domains, affine function manipulation and so on.

Because the system depends on the back-end engine for basic polyhedral operations, the program is frequently transformed at the back-end engine, and then reloaded to the front end system. A state handling mechanism keeps this transparent to the user so that the user will always see a consistent version of the program. To reduce the network load, some of the information is cached on the system side, and the number of transfers are minimized.

Code generator apply a series of transformations to the program during code generation, if an exception occurs while applying the transformation the state handling mechanism reverts back to the state before the transformation is applied.

And it also makes sure that the user will not see the transformations that were applied by the code generator in the process of generating code by reverting to the state when the code generator was called.

3.2 Target Mapping (TM)

Target mapping is a data structure to store the information about time-processor specification, tiling specification and memory allocation. It consists a list of affine functions for each variable for time-processor specification, tiling specification and affine functions with mod factors (optional) for memory allocation.

In order to generate code, some code generators need some additional information apart from the time-processor specification and memory allocation. Some code generators need statement orderings if they want the program to follow a specific execution order inside the loop body. These statement orderings are partial orders between two variables in the program and these are stored in a data structure which is associated with TM. Currently, the tiling specification is also implemented as a similar auxiliary annotation.

The user can also specify the loop types of the resulting code. Typical code generators in the Polyhedral model that use CLooG [1] produce loop nests where each dimension corresponds to a dimension in the schedule, if it is a fully sequential program. When some dimensions of the iteration space can be executed in parallel, the user could specify them as the parallel loops in the loop types of the TM. The default assumption is that all parallel loops are innermost. However, the user can specify loop types to specify outer parallel programs, or even interleaved sequential and parallel loops.

3.3 GeneralizedCoB

GeneralizedCoB performs a change of basis transformation even if the given transformation τ is not square. Change of basis transformation also called space-time mapping transformation provides one-to-one mapping between a time-processor dimension and the loop index and, it also aligns each time and processor dimension to a distinct axis. Since the change of basis transformation in the back-end engine is currently not able to find the inverse of a function and also it requires inverse when there is a non-square transformation, the GeneralizedCoB tries to find the left inverse in context for the transformation τ and if found it applies the change of basis transformation along with the inverse of the transformation τ^{-1}. This is a useful transformation since most of the times the variables in the program have non-square transformations and the user is spared of finding the inverse of the transformation that they want to apply.

Chapter 4

Code Generator

In this chapter we first illustrate the code generation with an example. Later, we describe the framework and how the core code generator is implemented, and explain how various extensions are constructed.

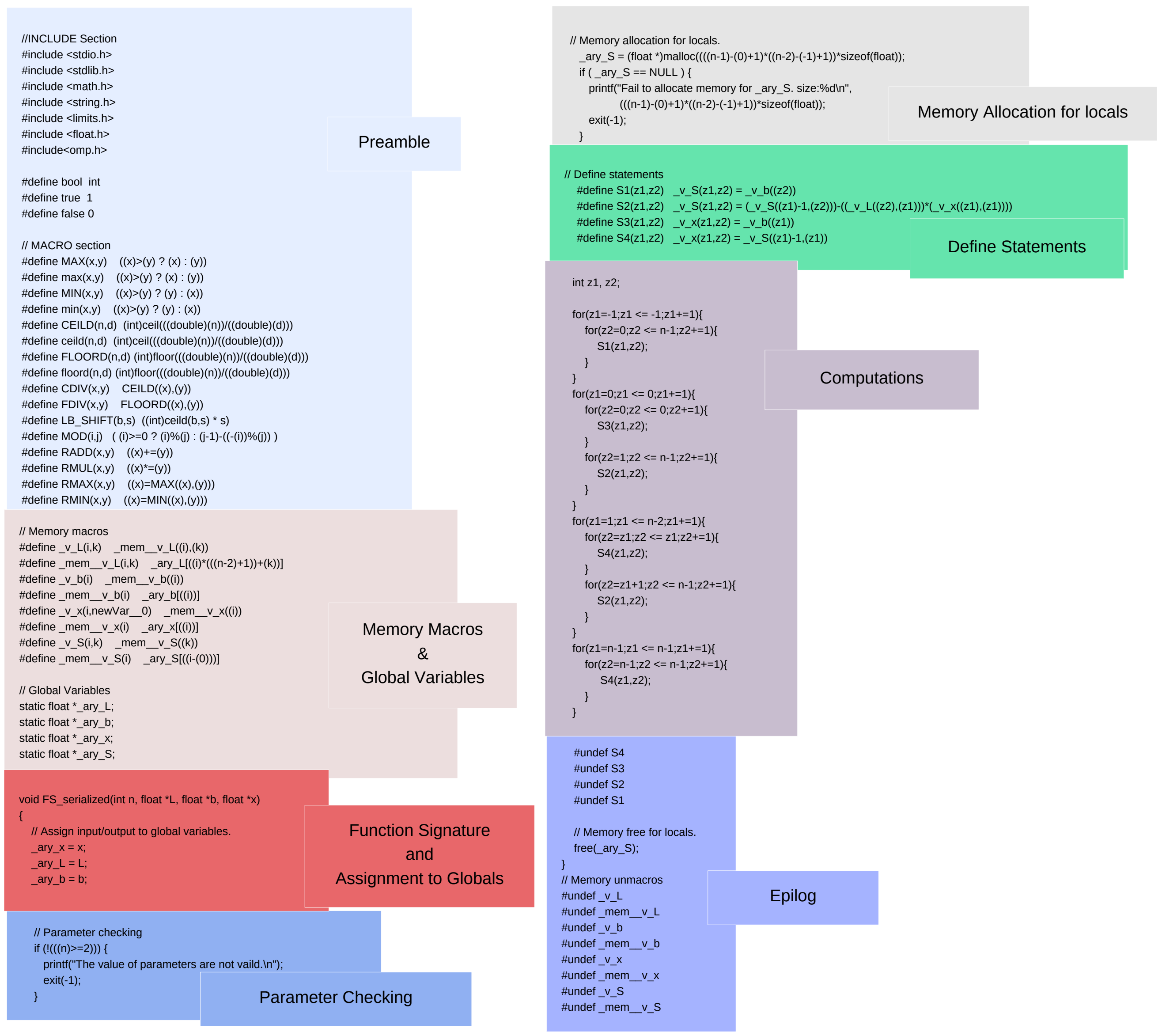
4.1 An example : Forward Substitution

Let us consider the alphabets program shown in Figure 4.1. It describes the equations to solve a lower triangular system of equations using the well known forward substitution algorithm. Let us say that we want to generate fully sequential C code from this program. We must specify to the generator, a fully sequential schedule for each variable. For this example, we choose the sequential schedule for the variable `x` to be $(i \rightarrow i,\, i)$ indicating that the $i-th$ value of `x` is computed at the (2-dimensional) time instant $(i,\, i)$. Similarly, we choose the schedule for the variable `S` to be $(i,\, k \rightarrow k,\, i)$. In addition, the code generator must be given a "memory spec", namely the memory address to which the local variables in the program are allocated. For our example, let us assume that this choice for the variable `S` is $(i,\, k \rightarrow i)$ which implies that a single dimensional array will be used to store the 2-dimensional set of values that will be computed in the program.

The output of our code generator is shown in 4.2, each of whose components are explained below.

```
affine FS_serialized {n | n>1} // Size Parameters
given
   float L {i,k| 0<i<n && 0<=k<i}; // Input Matrix
   float b {i| 0<=i<n}; // Input Vector
returns
   float x {i| 0<=i<n}; // Output Vector
using
   float S {i,k| 0<=i<n && -1<=k<i}; // Local variable for summation
through
// List of equations
   x[i] = case
          {| i==0} : b[i];
          {| i>0} : S[i,i-1];
          esac;
   S[i,k] = case
            {|k==-1} : b[i];
            {|k>-1} : S[i,k-1] - L[i,k]*x[k];
            esac;
. // Period terminated
```

Figure 4.1: Forward-Substitution alphabets program



```
//INCLUDE Section
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <limits.h>
#include <float.h>
#include<omp.h>

#define bool  int
#define true  1
#define false 0

// MACRO section
#define MAX(x,y)    ((x)>(y) ? (x) : (y))
#define max(x,y)    ((x)>(y) ? (x) : (y))
#define MIN(x,y)    ((x)>(y) ? (y) : (x))
#define min(x,y)    ((x)>(y) ? (y) : (x))
#define CEILD(n,d)  (int)ceil(((double)(n))/((double)(d)))
#define ceild(n,d)  (int)ceil(((double)(n))/((double)(d)))
#define FLOORD(n,d) (int)floor(((double)(n))/((double)(d)))
#define floord(n,d) (int)floor(((double)(n))/((double)(d)))
#define CDIV(x,y)    CEILD((x),(y))
#define FDIV(x,y)    FLOORD((x),(y))
#define LB_SHIFT(b,s)  ((int)ceild(b,s) * s)
#define MOD(i,j)   ( (i)>=0 ? (i)%(j) : (j-1)-((-(i))%(j)) )
#define RADD(x,y)    ((x)+=(y))
#define RMUL(x,y)    ((x)*=(y))
#define RMAX(x,y)    ((x)=MAX((x),(y)))
#define RMIN(x,y)    ((x)=MIN((x),(y)))

// Memory macros
#define _v_L(i,k)    _mem__v_L((i),(k))
#define _mem__v_L(i,k)    _ary_L[((i)*(((n-2)+1))+(k))]
#define _v_b(i)    _mem__v_b((i))
#define _mem__v_b(i)    _ary_b[((i))]
#define _v_x(i,newVar__0)    _mem__v_x((i))
#define _mem__v_x(i)    _ary_x[((i))]
#define _v_S(i,k)    _mem__v_S((k))
#define _mem__v_S(i)    _ary_S[((i-(0)))]

// Global Variables
static float *_ary_L;
static float *_ary_b;
static float *_ary_x;
static float *_ary_S;

void FS_serialized(int n, float *L, float *b, float *x)
{
   // Assign input/output to global variables.
   _ary_x = x;
   _ary_L = L;
   _ary_b = b;

   // Parameter checking
   if (!(((n)>=2))) {
      printf("The value of parameters are not vaild.\n");
      exit(-1);
   }

 // Memory allocation for locals.
   _ary_S = (float *)malloc((((n-1)-(0)+1)*((n-2)-(-1)+1))*sizeof(float));
   if ( _ary_S == NULL ) {
      printf("Fail to allocate memory for _ary_S. size:%d\n",
                (((n-1)-(0)+1)*((n-2)-(-1)+1))*sizeof(float));
      exit(-1);
   }

// Define statements
   #define S1(z1,z2)   _v_S(z1,z2) = _v_b((z2))
   #define S2(z1,z2)   _v_S(z1,z2) = (_v_S((z1)-1,(z2)))-((_v_L((z2),(z1)))*(_v_x((z1),(z1))))
   #define S3(z1,z2)   _v_x(z1,z2) = _v_b((z1))
   #define S4(z1,z2)   _v_x(z1,z2) = _v_S((z1)-1,(z1))

   int z1, z2;

   for(z1=-1;z1 <= -1;z1+=1){
      for(z2=0;z2 <= n-1;z2+=1){
         S1(z1,z2);
      }
   }
   for(z1=0;z1 <= 0;z1+=1){
      for(z2=0;z2 <= 0;z2+=1){
         S3(z1,z2);
      }
      for(z2=1;z2 <= n-1;z2+=1){
         S2(z1,z2);
      }
   }
   for(z1=1;z1 <= n-2;z1+=1){
      for(z2=z1;z2 <= z1;z2+=1){
         S4(z1,z2);
      }
      for(z2=z1+1;z2 <= n-1;z2+=1){
         S2(z1,z2);
      }
   }
   for(z1=n-1;z1 <= n-1;z1+=1){
      for(z2=n-1;z2 <= n-1;z2+=1){
         S4(z1,z2);
      }
   }

   #undef S4
   #undef S3
   #undef S2
   #undef S1

   // Memory free for locals.
   free(_ary_S);
}
// Memory unmacros
#undef _v_L
#undef _mem__v_L
#undef _v_b
#undef _mem__v_b
#undef _v_x
#undef _mem__v_x
#undef _v_S
#undef _mem__v_S
```

Figure 4.2: Blocks in the output program

4.1.1 Preamble

The preamble consists of the necessary include statements and some common macro definitions. Of these, most are pretty standard. The special ones are "accumulation" macros for reduction operators in Alpha, and are defined as follows.

```
#define RADD(a,b) (a) += (b)
```

```
#define RMUL(a,b) (a) *= (b)
#define RMIN(a,b) (a) = MIN((a),(b))
#define RMAX(a,b) (a) = MAX((a),(b))
```

4.1.2 Memory Macros & Global Variables

Memory macros are defined to access the physical memory allocated for the Alphabets variables. For each variable in AlphaZ, two macros are defined. The first macro maps the iteration domain of the variable to its memory *domain*. The second macro maps points in the memory domain to a physical memory address. The macros produced for the FS example program are as follows.

```
// Memory macros
#define _v_L(i,k)           _mem__v_L((i),(k))
#define _mem__v_L(i,k)      _ary_L[((i)*(((n-2)+1))+(k))]
#define _v_b(i)             _mem__v_b((i))
#define _mem__v_b(i)        _ary_b[((i))]
#define _v_x(i,newVar__0)   _mem__v_x((i))
#define _mem__v_x(i)        _ary_x[((i))]
#define _v_S(i,k)           _mem__v_S((k))
#define _mem__v_S(i)        _ary_S[((i-(0)))]
```

In addition to these macros, and for compatibility with the code generators, that produce the default, demand driven code, a list of static global declarations for all the variables in the program is also produced. These act as the global pointers for the input and output variables. The local variables will be allocated memory using these declarations.

4.1.3 Function Signature and Assignment to Globals

The function name is the same as the system name of the alphabets program. The signature consists of all the parameters, input variables and the output variables. The signature for the FS example program is as follows.

```
 void FS_serialized(int n, float *L, float *b, float *x)
```

The input and output variables have to be assigned to the previously declared static global pointers.

4.1.4 Parameter checking

An Alphabets program has a parameter declaration and hence generated *C* function is called must verify that the value of the size parameters passed as arguments are legal, i.e., satisfy the constraints of this parameter domain declared in the Alphabets program. For our example, the following code is generated for parameter checking.

```
// Parameter checking
if (!(((n)>=2))) {
   printf("The value of parameters are not vaild.\n");
   exit(-1);
}
```

4.1.5 Memory allocation for locals

Memory for the local variables are allocated using the malloc system call. It allocates a 1-dimensional array for all variables and the previously generated memory macros ensure that the references in the later code are correct. In our example, the variable `S` is a local variable. The code produced will be the following.

```
// Memory allocation for locals.
  _ary_S = (float *)malloc((((n-1)-(0)+1))*sizeof(float));
  if ( _ary_S == NULL ) {
      printf("Fail to allocate memory for _ary_S. size:%d\n",
             (((n-1)-(0)+1))*sizeof(float));
      exit(-1);
  }
```

4.1.6 Define statements

An Alphabets program consists of a set of normalized equations, where each equation has a set of clauses, each of which can be viewed as a "domain-guarded" expression. The next part of the generated *C* code is a set of statements, one for each clause of each equations. For our FS example, the following statement definition macros are produced.

```
// Define statements
  #define S1(z1,z2)  _v_S(z1,z2) = _v_b((z2))
  #define S2(z1,z2)  _v_S(z1,z2) =
                     (_v_S((z1)-1,(z2)))-((_v_L((z2),(z1)))*(_v_x((z1),(z1))))
  #define S3(z1,z2)  _v_x(z1,z2) = _v_b((z1))
  #define S4(z1,z2)  _v_x(z1,z2) = _v_S((z1)-1,(z1))
```

In the produced code we can notice that for every case expression of the variables `x` and `S` in the alphabets program, we have a corresponding statement macro.

4.1.7 Computations

This part of the code produced consists of a set of loops in which the statements are executed and constitutes the actual computation of the program. The generated loops for our FS example program is the following.

```
int z1, z2;
for(z1=-1;z1 <= -1;z1+=1){
   for(z2=0;z2 <= n-1;z2+=1){
      S1(z1,z2);
   }
}
for(z1=0;z1 <= 0;z1+=1){
   for(z2=0;z2 <= 0;z2+=1){
      S3(z1,z2);
   }
   for(z2=1;z2 <= n-1;z2+=1){
      S2(z1,z2);
   }
}
for(z1=1;z1 <= n-2;z1+=1){
   for(z2=z1;z2 <= z1;z2+=1){
      S4(z1,z2);
   }
   for(z2=z1+1;z2 <= n-1;z2+=1){
      S2(z1,z2);
   }
}
for(z1=n-1;z1 <= n-1;z1+=1){
   for(z2=n-1;z2 <= n-1;z2+=1){
      S4(z1,z2);
   }
}
```

4.1.8 Epilog

This block of the code produced consists of undefine statements, memory free for locals and memory unmacros. The undefine statements consists of a list of undefine macros for the statement macros generated earlier. This keeps the macros local to this function and will not have any conflicts if some macros with same name are defined in the application in which this program is plugged into. The memory free for locals contains a list of free statements for all the local variables declared. The memory unmacros contains the undefine macros for all the memory macros declared earlier.

4.2 Framework

4.3 shows the code generator framework in our AlphaZ system. Now we will describe how each of the code fragments described above is produced by our core code generator. Many of the sections are straightforward, and obvious to generate, so we will focus on Statement & Domain Collector, Loop Generator, Statement Pretty Printer and Memory Access Generator.

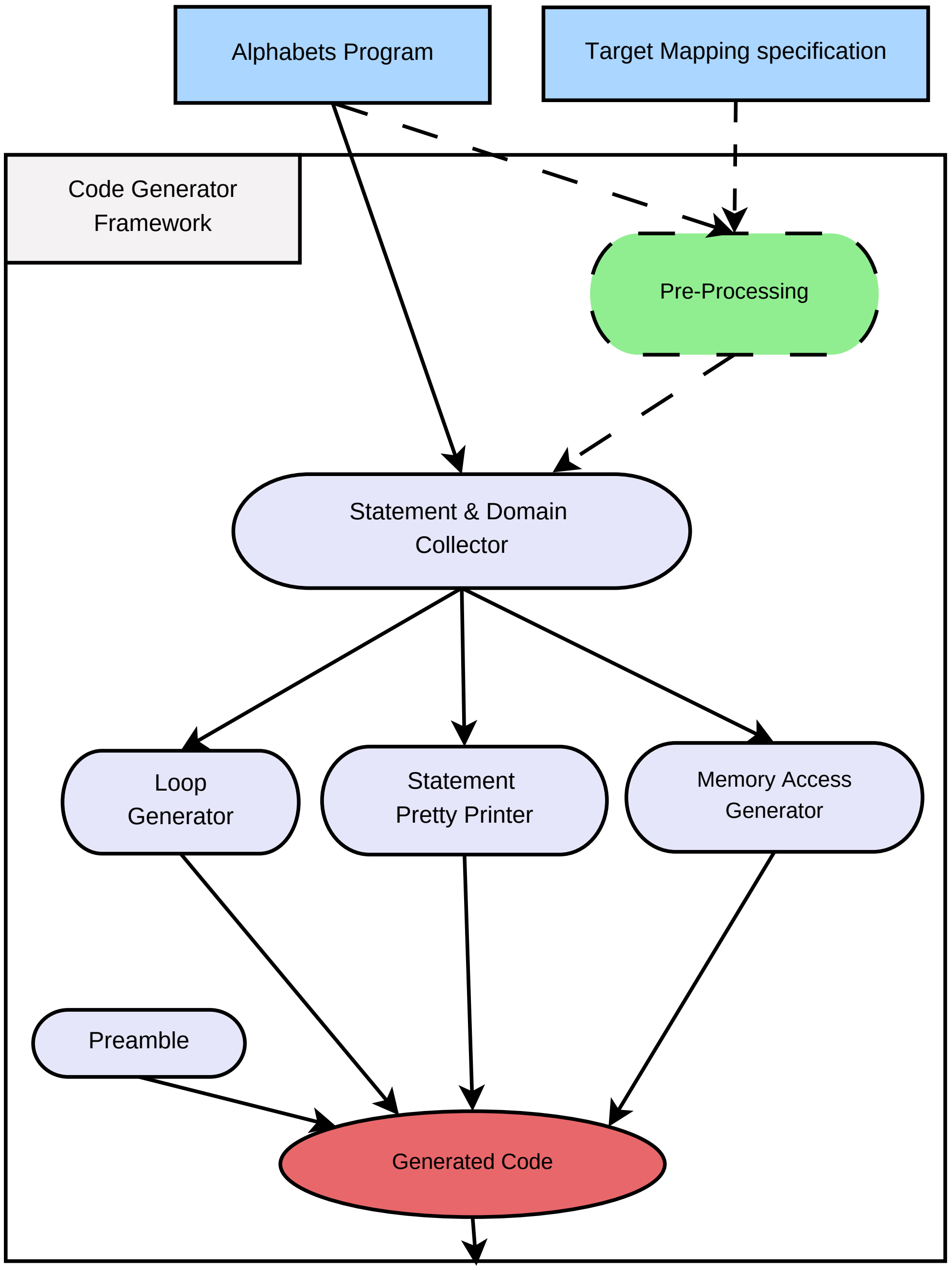


Figure 4.3: Code generator framework.

4.2.1 Statement & Domain Collector

Statement & Domain Collector is a visitor that visits all the equations in the program and collects the statements and the corresponding context domains. The algorithm it uses is as follows.

Algorithm 1

```
Input : A list of equations.
Output: Two separate list of statements and domains.
For each branch of each expression in the program,
1. let $var$ be the variable on the lhs of the equation, and $expr$ be the sub-expression corresponding to this branch (either a child sub-expression of the restrict expression or the sub-expression itself)
2. Construct a statement of the form,

        $var(z) = expr;$

where $z$ is a point in the domain of the expression.
3. Store the above constructed statement in a list.
4. Find the context domain of the $expr$ node and store it in a second list.
```

4.2.2 Loop Generator

The Loop Generator produces a set of loops that constitutes the computation in the program. The loops are generated using a tool called ClooG [1]. The Algorithm 2 explains how the loops are produced using ClooG. ClooG allows us to specify a scanning order to follow using scattering functions. In our code generator framework, we provide identity as the scattering function. The Loop Generator takes a list of domains and a full order list as input. The full order list specifies a total ordering between the variables.

Algorithm 2

Input : A list of domains and the full order list.
Output: A set of loops.
1. Construct the ClooG input, for every variable *var* in the full order list,
1.1. Add the context domain information for each statement of the variable *var* to the input.
1.2. Construct an identity scattering function for each context domain added and add the scattering function to the input.
1.3. Include the necessary information about the language used, index names, parameters etc in the input.
2. Call ClooG with the input to generate a set of loops which corresponds to the computations.

4.2.3 Statement Pretty-Printer

The Statement Pretty-Printer constructs a set of macros which map a statement to a statement number and set of macros which undefines the same.

Algorithm 3

Input : A list of statements and the full order list.
Output: Two set of macros to map statement to a statement number and unmap the same.
For every variable *var* in the full order list,
1. Construct a macro of the form,

```
#define Sn(z) var(z) = expr;
```

where, Sn denotes the statement number.
2. Construct one more macro of the form,

```
#undef Sn
```

4.2.4 Memory Access Generator

Memory Access Generator constructs memory macros that define access to the physical memory allocated for the Alphabets variables. For each variable in the alphabets program, two macros are defined. The first macro maps the iteration domain of the variable to its memory *domain*. The second macro maps points in the memory domain to a physical memory address.

If a schedule is specified for the alphabets variable and the schedule is applied as a transformation. The transformation transforms all the output and local variables to a common space-time co-ordinate iteration space, the domain of the memory access function has the same number of dimensions as the number of space and time coordinates. This index point must be transformed to derive the index point in the original variable declaration. In order to do that, We take the inverse of the transformation applied to the variable and then compose it with its memory map function.

In our example for the variable S, the transformation applied is $(i, k \rightarrow k, i)$ whose inverse is $(k, i \rightarrow i, k)$, and the given memory map function is also identity $(i, k \rightarrow i)$. Therefore the macro is

```
_var_S(i,k) mem_v_S((k))
```

The input variables are not transformed to a common co-ordinate space and the memory map function for input variables is always identity.

The physical memory for every n-dimensional array variable is allocated as a 1-dimensional array. Therefore the memory access functions for all variables should now be a memory access function to this allocated memory, using a standard doping vector as given by the second memory macro.

For our example, the variable S, the macro is given below, and requires the code generator to determine the bounding box of the image of the variable domain by the first memory macro.

```
#define mem_v_S(i) _ary_S[((i-(0)))]
```

4.3 Scheduled code generator

The input for this code generator module is an equational program, a target mapping (TM) and a full order list. The output of this code generated is a *C* code which follows the given schedule and the memory allocation. The TM specifies the time-processor transformation, memory specification, tiling specification and full order list. If the TM is given, the code generator module first invokes the Verifier which checks for the legality of the schedule, processor and memory allocation maps. The verifier is a module which is independent of the code generator, and outside the scope of this thesis.

The time-processor (TP) component of the TM is used to transform the domains of all the variables in the program so as to align them into a single, common coordinate space. In this new space, all variables have the same number of dimensions, equal to the number of dimensions of the combined schedule and processor allocation, thus providing a one-to-one mapping between a T-P dimension and the eventually generated loop index. This transformation also aligns each time and processor dimension to a distinct axis. This corresponds to the Pre-Processing module of the code generator framework. The alignment step is independent of the Verifier, and the subsequent code generator modules are easily implemented as sophisticated "pretty-printer". The variables with reductions are handled differently, as explained in the Chapter 5.

This code-generator produces loop programs which execute the computations in the order specified by the schedule component of the TM. It is used as the base engine for other code generators that produce parallel codes for different target architectures. This code generator reuses a number of modules from the demand driven code generator [7]. The demand driven code generator is a code generator that produces executable code for any legal Alphabets program without any additional information like TM and full order list etc. The Statement & Domain Collector module which eventually produces drastically different code, has an almost identical visitor structure with that of the demand driven code generator. One difference between the two is in handling reductions, where the scheduled code generator allows the TM to specify the specific order of individual accumulations in a reduction, whereas, the demand driven code considers reduction as atomic, and implements it with a loop. Because of this, a minor modification is required in Statement & Domain Collector and the Statement Pretty Printer to handle reductions. These modifications are explained in the Chapter 5.

4.4 Shared Memory Parallel code generator

The next code generator that we have developed using the same framework is the shared memory parallel code generator which is an extension of the scheduled code generator. This code generator generates parallel programs using OpenMP and is therefore limited to shared memory architectures. The inputs for this code generator are a TM with a parallel schedule, synchronizations (specified as special "synchronization domains" which are eventually treated as special local variables) and the loop types for every index. The loop types specify whether the loop corresponding to an index is a parallel or a time loop. This code generator is a simple extension of the scheduled code generator and the extensions are adding OpenMP pragmas for specifying parallel loops and synchronizations. The output from the Loop Generator module is post processed to add parallel pragmas, and the Statement Pretty Printer is modified to support synchronizations.

It generates parallel code corresponding to the given parallel schedule. The user can specify a schedule where parallel and time loops are interleaved by giving the order of parallel and time loops in the loop types. The necessary changes to support this code generator are divided into two parts, Handling synchronization and Post-processing.

4.4.1 Handling synchronization

In an OpenMP program, we may need to insert barrier synchronizations at certain points to ensure legality. These synchronizations can be deduced automatically by the verifier and it is given to the code generator as a list of synchronization domains which are eventually added as special local variables in the program. These variables will have a domain in which they are to be executed and are also given a TM of their own to place them in the transformed iteration domain of the program (after applying CoB transformation). The equations for these variables are defined to be dummy that has a dependency on themselves.

The process of generating the statements for these synchronization variables is same as for any other variable in the program. The only place where it is different is while pretty printing the statements in the Statement Pretty Printer module of the code generator framework. The statements corresponding to the synchronization variables are changed to the following

```
#define Sn(z) #pragma omp barrier
```

where Sn is the statement number and z is any point in the domain.

4.4.2 Post-processing

The loop type for each index is specified as an input for the code generator. This loop type tells us whether a loop corresponding to an index is a time loop or a parallel loop. This information is needed to modify the ClooG output. The ClooG AST which is called CLAST is modified by inserting the necessary OpenMP pragmas using a simple visitor which prints the CLAST. This is done in the Post-Processing module of the code generator framework. The algorithm to insert the necessary pragma is given in Algorithm 4.

Algorithm 4 Inserting OpenMP pragmas

```
Input : A set of loops and loop type.
Output : A set of loops with necessary annotations.
Insert #pragma omp parallel default(private) for the entire computation.
For every for-loop,
1. Find the index name $i$.
2. If $i$ is specified to be parallel in loop type then,
3. Insert #pragma omp for before the for statement.
4. And also insert #pragma omp parallel shared $\psi$ after the for statement for
the entire section of code inside the for-loop
where $\psi$ is the set of all index variable names of the for-loops encountered before
this for-loop and the set of program parameters.
end
```

Chapter 5

Handling Reductions

In this chapter, first we illustrate how reductions are handled with an example. Later we describe what is a reduction, constraints on the reductions that the code generator can handle (all of which can be satisfied by appropriately pre-processing the input program), the details of this pre-processing, and how the code is generated for the reductions in the scheduled code generators and how to handle the nested reductions in a program.

5.1 An example

```
affine MMM {N|N>0}
given    int A,B {i,j | 0<=i<N && 0<=j<N};
returns  int C {i,j | 0<=i<N && 0<=j<N};
through
   C = reduce(+, (i,j,k->i,j), (i,j,k->i,k)@(A)*(i,j,k->k,j)@(B));
.
```

Figure 5.1: Matrix-Matrix Multiplication with reduction.

```
affine MMM {N|N>0}
given   int A,B {i,j | 0<=i<N && 0<=j<N};
returns int C {i,j | 0<=i<N && 0<=j<N};
using   int tempC {i,j,k | 0<=(i,j,k)<N};
through
    tempC[i,j,k] = A[i,k] * B[k,j];
    C = reduce(+, (i,j,k->i,j),
               tempC[i,j,k]));
.
```

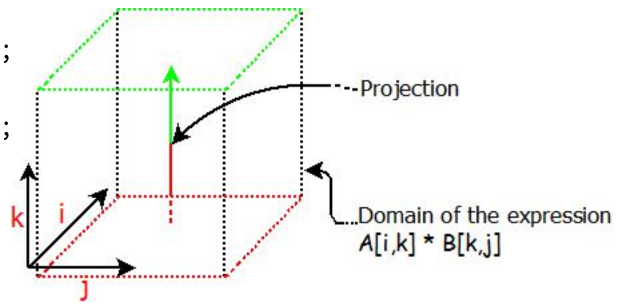


Figure 5.2: Matrix-Matrix Multiplication after pre-processing.

Consider the example in 5.1 with a schedule $(i, j, k \to k, i, j)$ for the expression body inside the reduction of the variable C. The pre-processing introduces a temporary variable tempC. The right hand side of the equation for the variable tempC is the expression inside the reduction. The expression inside the reduction is replaced by an access to the variable tempC. The program after pre-processing is shown in 5.2 (left). The domain of the temporary variable is the cube in 5.2 (right). The tempC variable has the TPSpec of the variable C, since the TPSpec for the variable corresponds to the expression body inside the reduction. The final statement for the variable C when the code is generated is

`RSUM(C[i,j], A[i,k]*B[k,j]);`

where `RSUM(a,b)` is `a += b;`

The code generated for the computation in our example is shown in 5.3.

```
// Define statements
#define S1(z1,z2,z3)   RADD(_v_C((z2),(z3)),(_v_A((z2),(z1)))*(_v_B((z1),(z3))))
int z1, z2, z3;
for(z1=0;z1 <= N-1;z1+=1){
    for(z2=0;z2 <= N-1;z2+=1){
        for(z3=0;z3 <= N-1;z3+=1){
            S1(z1,z2,z3);
        }
    }
}
#undef S1
```

Figure 5.3: Computation for Matrix-Matrix Multiplication.

5.2 Reductions

Reduction in the Alpha language is an accumulation of values along a projection based on the associative and commutative binary operator over the domain of the expression body. It consists of a operator which is both associative and commutative, a projection function and an expression. The syntax of reduce expression in alphabets is:

$reduce(\oplus, (z \to f(z)), Expr)$

where $\oplus$ is a commutative and associative operator, z is any point in the domain of the expression body, f is the projection function, *Expr* is the expression body of the reduction.

The value of the reduction at any point z, is obtained by applying the associative and commutative operator $\oplus$ to the values of *Expr* at all the points in the domain of *Expr* along the projection f.

Reductions are important since it is very easy to write a program with reduction expressions instead of a serialized program. In most of the languages which support reductions, the high level reduction abstraction is removed during the compilation process and most compilers do it pretty soon. In our system, we delay it as much as possible, and the user never has to do the serialization but instead our code generators do it by taking a schedule for the reduction.

In the example 5.1 the associative and commutative operator is addition `(+)`, $(i, j, k \rightarrow i, j)$ is the projection function and $A[i, k] * B[k, j]$ is the expression inside the reduction.

5.3 Constraints on input programs

Our code generator imposes the following constraints on the input program with respect to reductions.

1. If there is a reduction, the reduction expression should be the only expression on the right hand side of an equation.

2. Every reduction thus defined should have a time-processor specification (TPSpec) for the *expression body* inside the reduction. This TPSpec describes the time and processor allocation for the body of the reduction, not for the result.

3. The answer variables into which the reductions are finally accumulated, are assumed to be appropriately initialized elsewhere in the generated code.

In our example, we cannot produce code as shown in 5.3 without specifying a schedule for the expression body inside the reduction and therefore, the second constraint is important. We specify a schedule for the expression body inside the reduction by specifying a schedule for the variable C.

5.4 Pre-Processing to handle reductions

In order to generate code from an alphabets program with reductions, some pre-processing is done to the program. This pre-processing is not seen by the user in the produced code.

1. For every reduction, the expression inside each reduction is replaced by a

temporary variable. The domain of the temporary variable is the context domain of the original reduction body expression.

2. The TPSpec for the temporary variable is the TPSpec of the reduction body expression.

3. Collect the information about the variables with reductions. Information like the temporary variables added to replace the expression inside the reduction, the projection function, the reduction operator etc are needed when modifying the statements for the variables with reductions without going through the visitor framework again.

Since the TPSpec is given to the expression body inside the reduction, the expression body has to be transformed to the common iteration domain based on the TPSpec. We cannot apply the transformation on the expression body of a reduction, therefore we separate the expression body of reduction and introduce a temporary variable. Then, we assign the TPSpec of the original expression body to the temporary variable. Now this temporary variable can be treated as any other variable and the transformation is applied and the statements are generated for the same temporary variable.

In our example, the temporary variable introduced is tempC and the TPSpec for this variable is same as the TPSpec of the expression body inside the reduction which is $(i, j, k \rightarrow k, i, j)$.

5.5 Code generation for Reductions

Once the pre-processing is done to the program. The change of basis transformation (CoB) constructed by taking the time-processor specification (TPSpec) of the TM is applied to all the variables except the variables which have reduction. Since the variables with reduction have a TPSpec associated to the expression body inside the reduction and the CoB transformation is applied to the temporary variable introduced, there is no need to apply the CoB transformation to the variable with the reduction. The CoB transformation is not applied to the variable which has reduction because the transformation is given only to the expression inside the reduction. While collecting the statements, the statements for the variables with reductions are neglected. Now, before generating the actual statements from the collected statements, we modify the statements for the temporary variables which are introduced to replace the expression inside the reductions. Algorithm 5 explains how the statements that are generated for the temporary variables are modified to generate the new statements for the variables with reduction.

Algorithm 5

Input : A list of statements for the temporary variables *temp*, introduced to replace the expression body of the reductions.

$$temp[\tau(z)] \; = expr;$$

where τ is the change of basis transformation.
Output: A list of statements for the variables with reductions.

$$op(var[\gamma(z)],\, expr);$$

For every variable *var* with reduction,
1. Find the left inverse in context of the change of basis transformation (τ) for the expression body inside the reduction.
2. Compose τ^{-1} with the projection function f to form a function $\gamma = f \circ \tau^{-1}$.
3. Construct the variable expression $var[\gamma(z)]$ with γ as the access function on the domain of the expression body of the reduction.
4. Now, construct a statement

$$op(var[\gamma(z)],\, expr);$$

where z is any point in the domain of the expression body of the reduction, *op* corresponds to any of the commutative and associative operator $RSUM$, $RMUL$, $RMIN$, $RMAX$. etc as given in the reduction operator of a reduction and the *expr* is the right hand side of the statement of the temporary variable *temp*.
Remove the list of statements for the temporary variable from the list of statements and insert the list of above constructed statements for the variables with reduction.

5.6 Nested Reductions

If a program has reductions, as explained earlier we require a time-processor specification (TPSpec) for the expression body inside the reduction given as the TPSpec of the result variable. We then separate the expression body by introducing a temporary variable. This temporary variable is transformed based on the TPSpec of the expression body inside the reduction. Then we store the result of this temporary variable in the variable with reduction. If we have a program with nested reductions or reduction combined with another expression in an equation, the current AlphaZ system does not provide support for giving the TM for a reduction itself. Therefore, the scheduled code generators in AlphaZ cannot handle these type of programs. Consider the program in 5.4. Since we can give TPSpec for only one expression body of a reduction, we have to write the program such that the program meets the constraints with respect to reductions. In order to do the same, we have a simple pre-processing step called NormalizeReductions. The example program after NormalizeReductions is shown in 5.5.

```
affine product {N | (N)>0} // Parameters
given // Input Matrices
  float A, B, C {i, k | 0<=i<N && 0<=k<N};
returns // Output Matrix
 float D {i, j| 0<=i<N && 0<=j<N};
through // List of equations
  D[i,j] = reduce(+, [k], reduce(+, [], B[i,k] * C[k,j]) * A[k,j]);
.
```

Figure 5.4: Nested Reductions example.

```
affine product {N | (N)>0} // Parameters
given    // Input Matrices
    float A, B, C {i, k | 0<=i<N && 0<=k<N};
returns  // Output Matrix
   float D {i, j| 0<=i<N && 0<=j<N};
Using    // Local Variables
  float AlphaZTempRed {i, j | 0<=i<N && 0<=j<N};
through // List of equations
  AlphaZTempRed[i,j] = reduce(+, [k], B[i,k] * C[k,j]);
  D[i,j] = reduce(+,  [k],  AlphaZTempRed[i,k] *  A[k,j]);
.
```

Figure 5.5: Nested Reductions after Normalization.

NormalizeReductions replaces reductions within another reduction with a reference to a new local variable. The transformation will flatten nested reductions to a single nest of reductions per each equation. This transformation is useful when there are nested reductions or when a reduction occurs in an expression along with other expressions. It replaces the reduction in such cases by declaring a local variable. Now the local variable has reduction as the only expression for its equation. It follows the following steps.

1. For every reduction in the program, it checks if this is the only expression on the right hand side of an equation of a variable. If so, do nothing.

2. If it is not the only expression on the right hand side of an equation, then introduce a local variable in the program. The domain of the local variable is the context domain of the reduction expression.

3. The equation for the local variable is the reduction expression and therefore it is the only expression in the equation of the local variable.

4. Replace the occurrence of the reduction with the local variable.

Chapter 6

Conclusion and Future work

We have described the design and implementation of code generator framework in the AlphaZ system. We have developed two code generators in this framework, the scheduled code generator and the shared memory parallel code generator. We have also described its modularity by showing how other code generators can be easily implemented in this framework by extending the current ones.

Our code generators takes an Alphabets program and produce a sequential or OpenMP parallel C code which can be plugged into the application and executed without any changes. The code generators that have been developed supports a schedule for the expression inside the reduction, which now need not follow a sequential schedule all the time.

The shared memory parallel code generator takes a schedule and processor allocation where both the sequential and parallel loops are interleaved. Most of the parallel codes that are written are outer time/sequential loops and inner parallel loops which are fine grained parallel code. Since our shared memory parallel code generator can handle a time-processor specification where the time and parallel loops are not necessarily inner time and outer parallel, it can be used to experiment with any combination of time and parallel loops to see which of them produce efficient parallel code. The parallel code can be either coarse grained or fine grained parallel code depending upon the schedule and processor allocation. This helps us in understanding the architecture that the code is produced for and we can automate the process of finding a good schedule which eventually produces efficient code.

6.1 Future work

6.1.1 Initialization of the reduction

Our code generators currently assume that the initialization of the variable with reduction is done prior to the call to the code generators. In order to get rid of this assumption, we need to initialize the initial values in the code generator itself. The initialization should occur when the memory is accessed for the first time in the program. The algorithm to find the first point where it is written is given in the Algorithm 6. Here z' represents any point in the domain of the variable v.

Algorithm 6 Initializing a reduction variable.

```
For every variable $v$ with reduction,
Construct a polyhedron with the parameters $P$ & $z'$ and with the context domain of the expression ($dom_{expr}$) of the reduction as the domain of the polyhedron.
where $z' = f \circ \tau^{-1}(z)$ where $f$ is the projection function, $\tau$ is the CoB transformation applied and $z$ is any point in the domain of the reduction expression.
With this polyhedron, take the loop types given by the user as the cost function and make a call to the PIP library to find a set of points which are lexicographically minimum in the given polyhedron.
These set of points are the points where the initialization should be done.
For these points and construct a initialization statement,
    $v[f \circ \tau^{-1}(z)] = expr;$
Now, take the domain of these set of points and subtract it with the $dom_{expr}$ to get the domain of the remaining accumulation.
Construct a statement for this new domain as explained in handling reductions,
    $op(v[f \circ \tau^{-1}(z)], expr);$
```

6.1.2 Handling Multiple Reductions

If there exists a equation for a variable which has a binary expression with two reductions with the same binary operator between the reductions as the operators inside the reductions, We can optimize the memory such that they can use the same physical memory. Currently, because of the first constraint in the reduction that is, If a reduction occurs in the program, reduction expression should be the only expression in the equation; this optimization cannot be performed. We can allow such equations and handle them as explained in the Algorithm 7. The initialization should be done on only one of the reduction based on the schedule. The reduction with the schedule which is earlier than the other will be chosen to initialize.

Algorithm 7 Handling multiple reductions.

For every variable X with multiple reductions,

$$X \,=\, reduce(op,\, f1,\, expr1)\, op\, reduce(op,\, f2,\, expr2)$$

introduce temporary variables $temp1_X$ and $temp2_X$, where

$$temp1_X[\tau_1(z_1)] \,= expr1;$$

$$temp2_X[\tau_2(z_2)] \,= expr2;$$

after applying the CoB transformation, replace the above statements with

$$op(X[f_1 \circ \tau_1^{-1}(z_1)],\, expr1);$$

$$op(X[f_2 \circ \tau_2^{-1}(z_2)],\, expr2);$$

where $z_1 \,\&\, z_2$ are any point in the domain of the $expr1$ and $expr2$, op corresponds to any of the commutative and associative operator SUM, MUL, MIN, MAX. etc as given in the reduction operator of a reduction, $f_1 \,\&\, f_2$ are the projection functions and $\tau_1 \,\&\, \tau_2$ are the change of basis transformation that is applied (time-processor specification of the variable).

6.1.3 Migrating to MDSE

Model-driven software engineering (MDSE) is a software development methodology which focuses on creating models, or abstractions, more close to some particular domain concepts rather than computing (or algorithmic) concepts. It increases productivity by maximizing compatibility between systems, simplifying the process of design, and promoting communication between individuals and teams working on the system. Hence, we are adopting MDSE methodology in our development of the AlphaZ system. Currently, tiled code generators are written using MDSE and the code generators described in this thesis will also be migrated soon.

REFERENCES

[1] C. Bastoul. Code generation in the polyhedral model is easier than you think. pages 7–16, 2004.

[2] U. Bondhugula, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical Report OSU-CISRC-10/07-TR70, The Ohio State University, October 2007.

[3] A. Cohen, S. Girbal, and O. Temam. A polyhedral approach to ease the composition of program transformations. pages 292–303, 2004.

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

[5] G. Gupta. Corequations. http://www.corequations.com/.

[6] C. IRISA. The MMAlpha environment.

[7] D. Kim. Code generation from polyhedra, 2005.

[8] H. Le Verge. Reduction operators in alpha. In *PARLE '92: Proceedings of the 4th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 397–411, London, UK, 1992. Springer-Verlag.

[9] C. Mauras. Alpha: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones. 1989.

[10] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G. Silber, and N. Vasilache. Graphite: Polyhedral analyses and optimizations for gcc. page 2006, 2006.

[11] L. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Hybrid iterative and model-driven optimization in the polyhedral model. Technical Report 6962, INRIA Research Report, June 2009.

[12] D. K. Wilde. The alpha language. Technical report.