Mini Assignment 2 : CS3423

Introduction to Polly

Saksham Mittal CS16BTECH110032

November 12, 2018

Contents

1 Polly architecture

Polly uses a polyhedral model(a mathematical representation) to represent and transform loops and some other control flow structures. This eventually helps polly tool to find optimal loop structure by using highly optimized linear programming libraries.
These transformations helps to do :

- Constant propogation through arrays.
- Removing dead loop iterations
- Optimizing loops for cache locality
- Optimizing arrays
- Applying advanced automatic parallelization
- Driving vectorization
- or, in some other software pipelining

Within polly tool, a variety of code transformations and analysis are performed on the mathematical model. The polly architecture states that we can consider the pipeline as a series of passes, that can be divided into differetn conceptual phases(same as used in **-O1**, **-O2**, **-O3** mode of clang/opt). The different phases are :

- Canonicalization
- Inliner cycle
- Target Specialization

Canonicalization : This is a scalar canonicalization phase which simplifies the IR by removing most of the redundant code, and hence focussing mostly on scalar optimizations. It broadly consists of the following passes :

- -mem2reg
- -InstCombine
- -CFGSimplify

Inliner cycle : This is a set of Simple loop optimizations, and the inliner itself. These passes do the same function as above pass, but without losing semantic information. The LLVM inliner tries to inline the methods, runs the Canonicalization passes and again tries to inline the simplified methods.
The Scalar Simplification passes consists of:

- -InstCombine
- -CGFSimplify

The simple loop optimizations are:

- -LoopRotate
- -LoopUnswitch
- -LoopUnroll
- -LoopDelete

Target Specialization : In this pass, the IR complexity is increased to take advantage of target specific features, which enhances the performance on the targetted device. This involves the following optimizations :

- Loop Vectorization
- Loop Distribution
- SLP Vectorization

2 Placing polly in pipeline

Conceptually, polly can be placed at 3 different places in the pipeline. As an early optimizer, as an late optimizer as part of Target Spcialization, or even in the inliner cycle. The first two are discussed next. The third is not a good idea, as it disturbs the inliner.

2.1 At early optimizer(-polly-position=early)

This is the deafault position of polly tool nowadays in the pipeline. The benefit is that LLVM-IR processed by the polly is still very close to the input code. This is important for better understanding of the code, and hence better for feedback.

2.2 At later optimizer(-polly-position=before-vectorizer)

Since, inliner had been run before the vectorizer, the polly can benefit from this. Because full inlining cycle has been run before this, even heavily templated C++ code can take benefit from polly. One of the drawbacks for this position can be that since LLVM has already performed so many optimizations on the IR before it reached polly, the effects of polly can be not seen.

3 Polly performance

We compared the performance of polly with respect to time taken for the Matrix Multiplication program to excecute. The matrix multiplication code contains a lot of loops and there is a lot of scope for optimizations as is seen from the results. As we can see from the results, the time for execution is reduced greatly on using polly with -O3 optimization.

3.1 Without polly or any optimization

Time taken = 80 s(approx.)

3.2 Polly with -O1

Time taken = 4 s(aprrox.)

3.3 Polly with -O2, -O3

Time taken = 1 s(approx.)

4 SCoPs and dependences

4.1 SCoP : Static Control Part

A Scop is the polyhedral representation of a control flow region detected by the Scop detection. It is generated by translating the LLVM-IR and abstracting its effects.

A Scop consists of:

- A set of statements executed in the Scop.
- A set of global parameters Those parameters are scalar integer values, which are constant during execution.
- A context This context contains information about the values the parameters can take and relations between different parameters.

Consider the following code :

```
for(i = 1; i <= n; i++) {
    for(j = 1; j <= n; j++) {
        if(i <= n + 2 - j)
            S(i,j);
    }
}
```

The iteration domain generated for the given code is:

$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ -1 \\ n \\ n+2 \end{pmatrix} \geq 0$$

The iteration domain of these loops is specified using a set of linear inequalities defining a polyhedron.

4.2 Loop Dependence

Loop dependence is an analysis process which is used to find dependencies within iterations of a loop to determine various relationships among statements.
With the knowledge of these dependencies, we can allow different processors to work on different non-independent parts of loop in parallel.
These dependencies also specify which statements to excecute before which statements, and whcih statements can be excecuted in parallel with respect to other statements in the loop.

The loop dependence can be classified into two parts:

- Data Dependence
- Control Dependence

4.2.1 Data Dependence

These are used to show and study the relationships between the variables in the code.

They are of 3 types:

- **True dependence :** A true dependence occurs when a location in memory is written to before it is read. In refernce to loops, a statement S in jth iteration, may have a true dependence on S in (j + 1)th iteration. This may happen because the value written in jth iteration may be read in (j + 1)th iteration.

```
for(j = 1; j < n; j++)
    S: a[j] = a[j-1];
```

- **Anti Dependence :** An anti dependence occurs when a location in memory is read before that same location is written to.

  Example for this is :

  ```
  for(j = 0; j < n; j++)
      S: b[j] = b[j+1];
  ```

- **Output Dependence :** An output dependence occurs when a location in memory is written to before that same location is written to again in another statement.

  Example for this is :

  ```
  for(j = 0; j < n; j++)
      S1: c[j] = j;
      S2: c[j+1] = 5;
  ```

4.2.2 Control Dependence

These are required for analysis when we are also considering dependencies between different statements of the loop. These can be introduced by the algorithm itself, apart from the code.
They are also responsible for the order in which instructions occur within the execution of the code.

5 Analysis of transformed code

We tried to use different flags like `-O3`, `-mllvm -polly`, `-mllvm -polly -mllvm -polly-parallel` while compiling IR. On observing the code, we can see that polly optimises the IR to a great extent if loops are present.
In our particular example it doesn't even generate IR for loop if the loop is not affecting the output of the program. But on the other hand, of the loop affects te output, it generates the IR for it.

Consider the following code :

```
#include <stdio.h>
int main()
{
    int a[100], b[100], c[100];
    for(int i = 0;i<100;i++)
    {
        a[i] = b[i] * c[i];
        printf("%d", a[i]);
    }
```

```
    printf("%d", a[0]);
    return 0;
}
```

Without the printf() statement, the alloca and store instructions on a, b, and c were not being generated in IR, and only return instructions were being generated.
Another thing we noticed is that if the number of iterations is less(200 to 500), it may even unroll the loop completely.
The IR generated on adding the printf() statement in the loop affects causes the loop's IR to be generated. The IR can be seen at LINK.

6 Bibilography

The resources used in writing this report are listed as follows:

- **http://polly.llvm.org/docs/Architecture.html**
- **http://polyhedral.info/**
- **http://web.cs.ucla.edu/ pouchet/doc/cc-article.10.pdf**
- **https://github.com/llvm-mirror/polly/blob/master/docs/experiments/matmul/matmul.c**