CS3423 - AUG 2019 : COMPILER II MINI ASSIGNMENT-2

Vijay Tadikamalla [1]

CS17BTECH11040

CONTENTS

LIST OF FIGURES

LIST OF TABLES

[1] *Department of Computer Science and Engineering, Indian Institute of Technology, Hyderabad*

1 POLLY AND ITS OPTIMIZATION CAPABILITIES

Polly is a high-level loop and data-locality optimizer and optimization infrastructure for LLVM. It uses an abstract mathematical representation based on integer polyhedra to analyze and optimize the memory access pattern of a program.

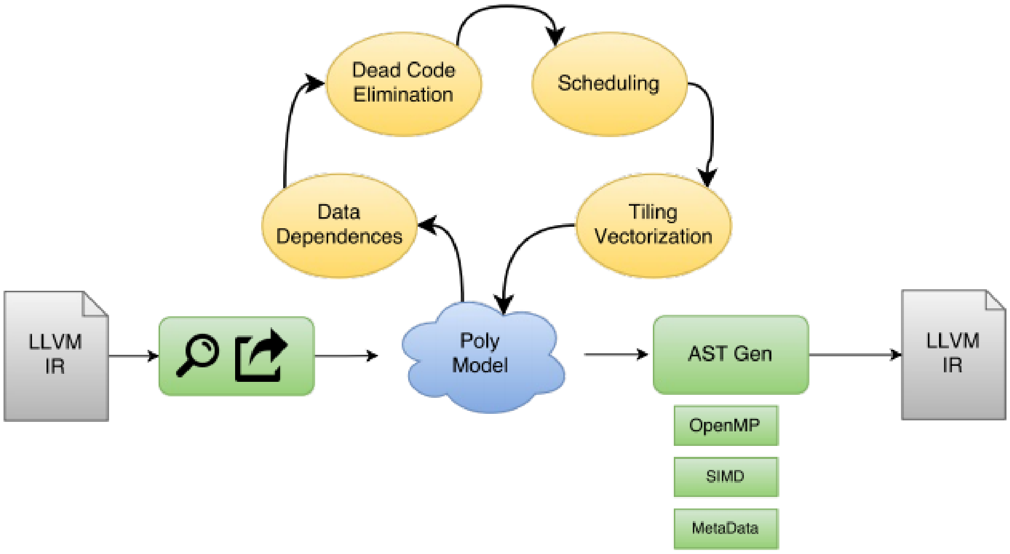


Figure 1: Polly in the LLVM pass pipeline

- Polyhedral compilation encompasses the compilation techniques that rely on the representation of programs, especially those involving nested loops and arrays,and that exploit combinatorial and geometrical optimizations on these objects to analyze and optimize the programs.
- Polyhedral representations can be manipulated or optimized with algorithms whose complexity depends on their structure and not on the number of elements they represent.
- Polyhedral techniques are the symbolic counterpart, for structured loops (but without unrolling them), of compilation techniques (such as scheduling, lifetime analysis, register allocation) designed for acyclic control-flow graphs or unstructured loops.

Polyhedral Compilation and its optimization capabilities are used in many fields and research areas like:

- Program analysis
- Scheduling theory
- For program verification like checking Data Race conditions
- For developmenting parallelizing compiler.

2 OBSERVATIONS

The results in the Table 1 show the comparison between two programs (with and without Polly optimisations) executions time.

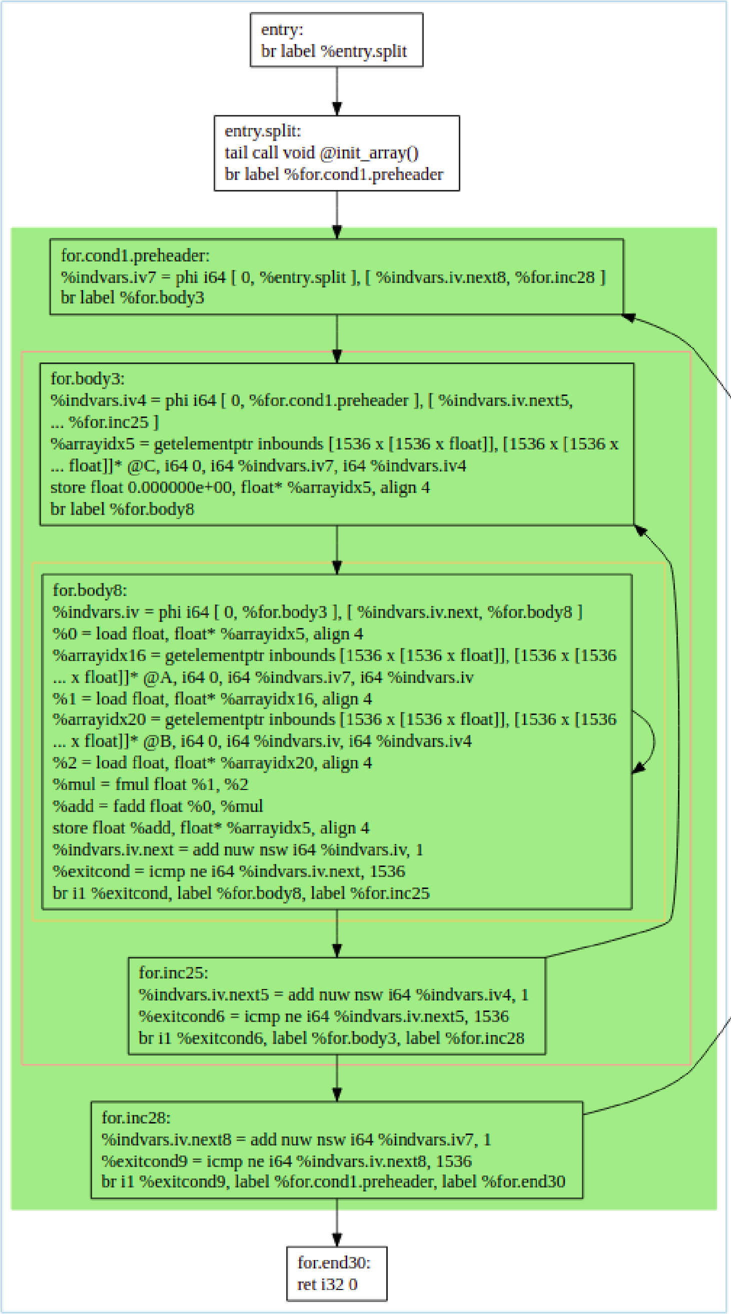
Table 1: Execution time comparison.

	O3 optimisation	**Polly with O3 optimisation**
Size of Matrix (N)	Time(in sec)	Time(in sec)
N = 1000	0.885	0.089
N = 1536	6.5	0.32
N = 2000	18.5	0.63

The following results were obtained on the system with following specifications:
Architecture: x86_64
CPU(s): 12
Model name: Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60GHz

3 SCOPS

- Regions of code that can be handled in the polyhedral model are usually called Static Control Parts(SCoPs).
- SCoPs contain regular control flow free of exceptions and other constructs that provoke changes in control flow such as conditional expressions dependent on data (read from memory) or side effects of function calls.
- SCoP detection is a search algorithm that a compiler for imperative programming languages is using to find loops to be represented and optimized in the polyhedral model.



```
entry:
br label %entry.split

entry.split:
tail call void @init_array()
br label %for.cond1.preheader

for.cond1.preheader:
%indvars.iv7 = phi i64 [ 0, %entry.split ], [ %indvars.iv.next8, %for.inc28 ]
br label %for.body3

for.body3:
%indvars.iv4 = phi i64 [ 0, %for.cond1.preheader ], [ %indvars.iv.next5,
... %for.inc25 ]
%arrayidx5 = getelementptr inbounds [1536 x [1536 x float]], [1536 x [1536 x
... float]]* @C, i64 0, i64 %indvars.iv7, i64 %indvars.iv4
store float 0.000000e+00, float* %arrayidx5, align 4
br label %for.body8

for.body8:
%indvars.iv = phi i64 [ 0, %for.body3 ], [ %indvars.iv.next, %for.body8 ]
%0 = load float, float* %arrayidx5, align 4
%arrayidx16 = getelementptr inbounds [1536 x [1536 x float]], [1536 x [1536
... x float]]* @A, i64 0, i64 %indvars.iv7, i64 %indvars.iv
%1 = load float, float* %arrayidx16, align 4
%arrayidx20 = getelementptr inbounds [1536 x [1536 x float]], [1536 x [1536
... x float]]* @B, i64 0, i64 %indvars.iv, i64 %indvars.iv4
%2 = load float, float* %arrayidx20, align 4
%mul = fmul float %1, %2
%add = fadd float %0, %mul
store float %add, float* %arrayidx5, align 4
%indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
%exitcond = icmp ne i64 %indvars.iv.next, 1536
br i1 %exitcond, label %for.body8, label %for.inc25

for.inc25:
%indvars.iv.next5 = add nuw nsw i64 %indvars.iv4, 1
%exitcond6 = icmp ne i64 %indvars.iv.next5, 1536
br i1 %exitcond6, label %for.body3, label %for.inc28

for.inc28:
%indvars.iv.next8 = add nuw nsw i64 %indvars.iv7, 1
%exitcond9 = icmp ne i64 %indvars.iv.next8, 1536
br i1 %exitcond9, label %for.cond1.preheader, label %for.end30

for.end30:
ret i32 0
```

Figure 2: SCoP graph of Main function

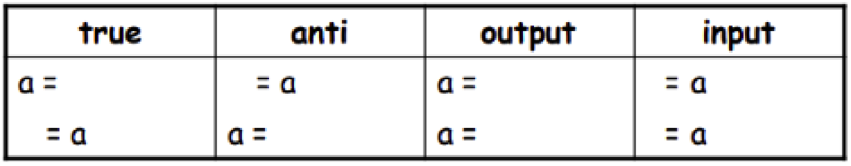
4 DEPENDENCIES

Consider the following example:
Given two program statements a and b, b depends on a if:

- b follows a (roughly)
- they share a memory location and one of them writes to it.
- Example:
  - a: x = y + 1;
  - b: z = x * 3;

The following are some of Dependencies observed in programs are:

- **True or flow dependence:** A writes a location that B later reads (read-after write or RAW)
- **Anti-dependence:** A reads a location that B later writes (write-after-read or WAR)
- **Output dependence:** A writes a location that B later writes (write-after-write or WAW)
- **Input dependence:** A reads a location that B later reads (read-after-read or RAR)



true	anti	output	input
a =	= a	a =	= a
= a	a =	a =	= a

Figure 3: Type of Dependencies

5 TRANSFORMATION ANALYSIS

The matrix multiplication code was optimized with several different optimization techniques:

- **O3:** All loops are heavily unrolled which provides a good performance boost.
- **Tiling:** By default Polly uses Tiling transformation pass. It is a loop transformation that exploits spatial and temporal locality of data accesses in loop nests. This transformation almost doubles the no. of loops. When we disable Tiling, we the observe that there is the significant reduction in the size of the code. The exact Tiling mechanism also heavily depends on the dependencies analysis done for the SCoP.
- **Vectorization:** It is a process of converting an algorithm from operating on a single value at a time to operate on a set of values at one time.
- **Loop interchange:** The loops are interchanged to optimize for cache locality. It is semantically equivalent and can improve code execution drastically depending on row major and column major memory access.

REFERENCES

- http://polly.llvm.org/docs/
- http://polyhedral.info/
- http://polly.llvm.org/docs/Architecture.html
- http://polyhedral.info
- http://impact.gforge.inria.fr/impact2016/papers/impact2016-kumar.pdf