# Introduction to Polly

# Polly Introduction

Polly is a loop optimizer for LLVM which finds and exploits interesting loop patterns/kernels. Each kernel is represented as a mathematical model describing the computations/ memory-accesses precisely. Upon analysis and code-transformations, Polly regenerates the optimized LLVM-IR.

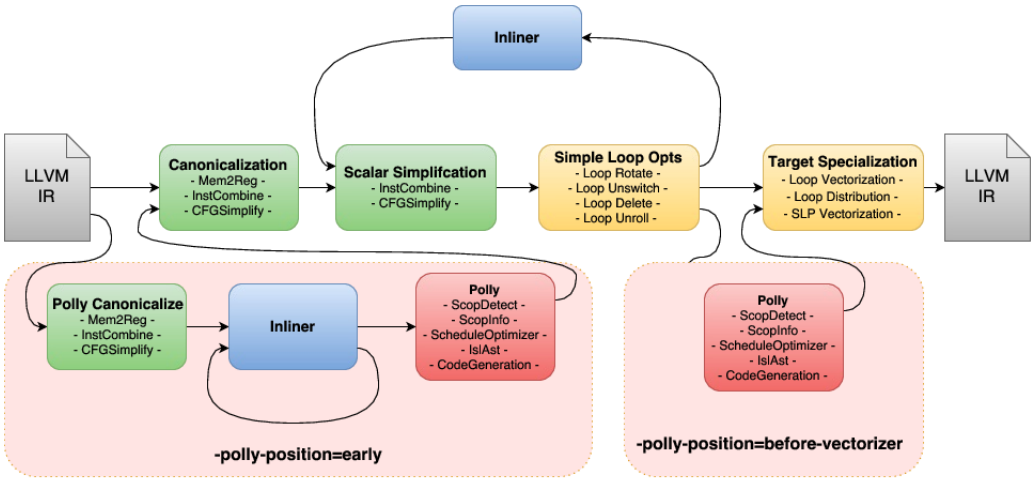
Conceptually, standard LLVM pass pipeline happens in three phases :

- **Canonicalization:** the goal is to optimize the given IR by simplifying and removing as much as possible using only scalar optimizations.
- **Inliner Cycle:** This is also a set of scalar optimizations grouped as follows:
  - Scalar Simplification:
  - Simple Loop Optimizations:This does loop optimizations like Loop Rotate, Loop Unroll, Loop Switch, Loop Delete etc.
  - Inliner: It aligns the function and the cycle goes on.
- **Target Specialization:** Takes advantage of target specific features that maximize the execution performance on the device we target though results in complex IR generation. Eg : Vectorization, Loop unrolling, some loop transformations (e.g., distribution) that expose more vectorization opportunities.

Polly can conceptually be run at three different positions in the pass pipeline. As an early optimizer before the standard LLVM pass pipeline, as a later optimizer as part of the target specialization sequence, and theoretically also with the loop optimizations in the inliner cycle. First two options are in current scope.

- **Early Phase:** Running Polly early before the standard pass pipeline has following pros/cons:
  - LLVM-IR processed by Polly is still very close to the original input code
  - Transformations applied by LLVM doesn't change the IR in ways not easily understandable for the programmer.
  - On the other hand, codes that require inlining to be optimized wonâĂŹt benefit if Polly is scheduled at this position
- **Before Vectorizer:** Running Polly right before the vectorizer has the following pros/cons:
  - Heavily templated C++ code could theoretically benefit from Polly as full in-lining cycle has been run already.

- No need to run additional canonicalization passes as IR passed has already been canonicalized.
- Detection of loop kernels is generally very fast not affecting general compile time.
- Polly becomes less efficient due to the many optimizations that LLVM runs before Polly the IR because of presence of additional scalar dependencies.




# Experiments

I execute following programs to optimize C program performing matrix multiplication of two $2000 \times 2000$ matrices using LLVM and Polly options.

| Optimization | Command | Time (s) |
|---|---|---|
| No Optimization | gcc matmul.c | 43.77 |
| O3 | clang -O3 matmul.c | 30.051 |
| Polly | clang -mllvm -polly matmul.c | 43.16 |
| Polly + O3 | clang -O3 -mllvm -polly matmul.c | 0.681 |
| Polly Loop Tiling | clang -mllvm -polly-tiling matmul.c | 42.819 |
| Polly + Polly Parallel | clang -mllvm -polly -mllvm -polly-parallel matmul.c | 42.662 |
| Polly Vectorizer | clang -mllvm -polly-vectorizer=polly matmul.c | 42.47 |

# SCoPs and Dependencies

Static Control parts(SCoPs) are the regions of code that can be handled in polyhedral compilation. Dependencies are the data and instruction dependencies such for one instruction needs to be executed before the other can be executed, the data needed to perform that instruction.

### SCoPs detected

```
:: isl ast :: init_array ::

if (1)

    for (int c0 = 0; c0 <= 1999; c0 += 1)
      for (int c1 = 0; c1 <= 1999; c1 += 1)
        Stmt_for_body3(c0, c1);

else
    {  /* original code */ }

:: isl ast :: main :: %for.cond1.preheader---%for.end30

if (1)

    for (int c0 = 0; c0 <= 1999; c0 += 1)
      for (int c1 = 0; c1 <= 1999; c1 += 1) {
        Stmt_for_body3(c0, c1);
        for (int c2 = 0; c2 <= 1999; c2 += 1)
          Stmt_for_body8(c0, c1, c2);
      }

else
    {  /* original code */ }
```

## Conclusion

From the results we can see that optimizations like vectorization, Polly Parallel, Loop-Tiling don't result in significant improvement. O3 performs a decent optimizations which when combined with polly results in least execution time. Patterns in IR:

- Induction variables are repetitively loaded
- Allocating and comparing induction variables is done in separate sections.
- Code changes considerably with optimizations like tiling.