Polly Documentation

Release 8.0-devel

The Polly Team

Aug 10, 2018

Contents

CHAPTER 1

Release Notes 8.0 (upcoming)

In Polly 8 the following important changes have been incorporated.

Warning: These releaes notes are for the next release of Polly and describe the new features that have recently been committed to our development branch.

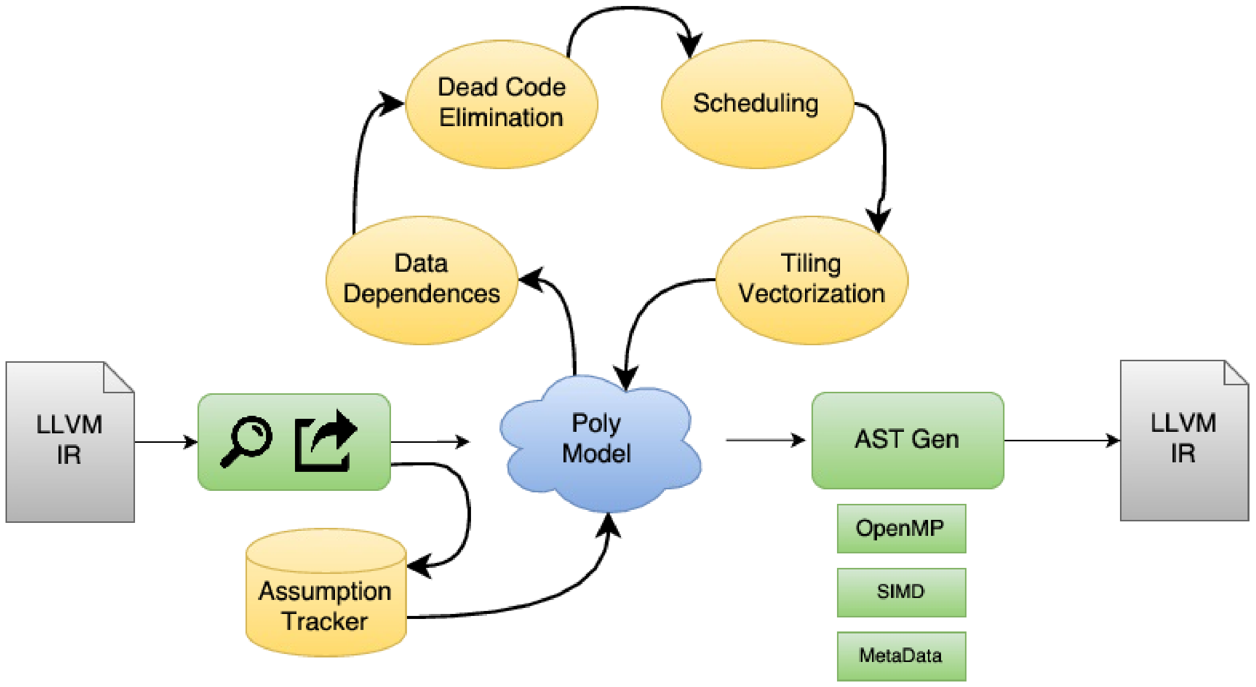
- Change ...

CHAPTER 2

Using Polly

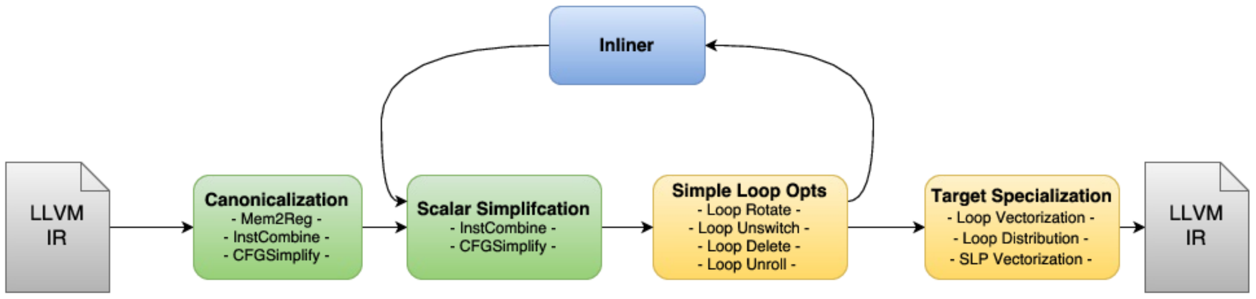
2.1 The Architecture

Polly is a loop optimizer for LLVM. Starting from LLVM-IR it detects and extracts interesting loop kernels. For each kernel a mathematical model is derived which precisely describes the individual computations and memory accesses in the kernels. Within Polly a variety of analysis and code transformations are performed on this mathematical model. After all optimizations have been derived and applied, optimized LLVM-IR is regenerated and inserted into the LLVM-IR module.

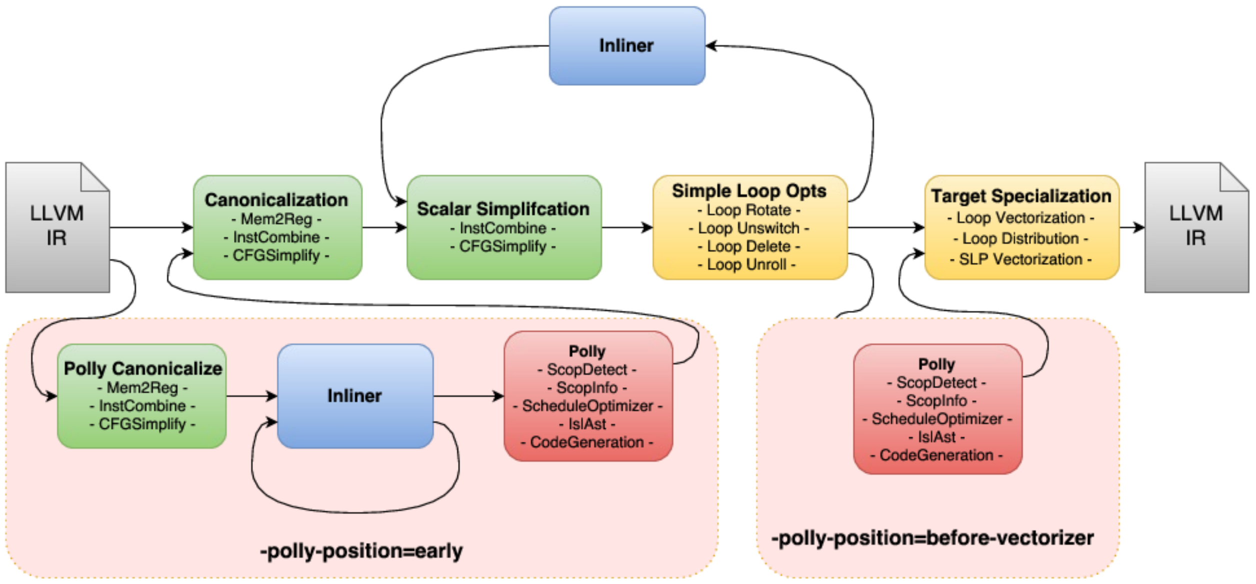


2.1.1 Polly in the LLVM pass pipeline

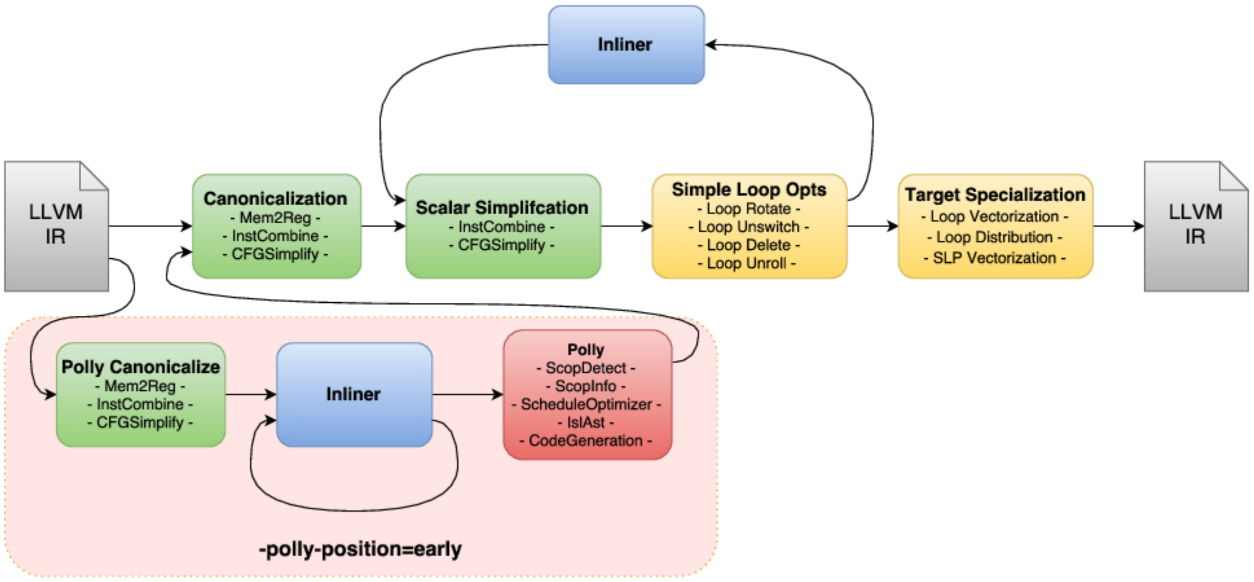
The standard LLVM pass pipeline as it is used in -O1/-O2/-O3 mode of clang/opt consists of a sequence of passes that can be grouped in different conceptual phases. The first phase, we call it here **Canonicalization**, is a scalar canonicalization phase that contains passes like -mem2reg, -instcombine, -cfgsimplify, or early loop unrolling. It has the goal of removing and simplifying the given IR as much as possible focusing mostly on scalar optimizations. The second phase consists of three conceptual groups that are executed in the so-called **Inliner cycle**, This is again a set of **Scalar Simplification** passes, a set of **Simple Loop Optimizations**, and the **Inliner** itself. Even though these passes make up the majority of the LLVM pass pipeline, the primary goal of these passes is still canonicalization without loosing semantic information that complicates later analysis. As part of the inliner cycle, the LLVM inliner step-by-step tries to inline functions, runs canonicalization passes to exploit newly exposed simplification opportunities, and then tries to inline the further simplified functions. Some simple loop optimizations are executed as part of the inliner cycle. Even though they perform some optimizations, their primary goal is still the simplification of the program code. Loop invariant code motion is one such optimization that besides being beneficial for program performance also allows us to move computation out of loops and in the best case enables us to eliminate certain loops completely. Only after the inliner cycle has been finished, a last **Target Specialization** phase is run, where IR complexity is deliberately increased to take advantage of target specific features that maximize the execution performance on the device we target. One of the principal optimizations in this phase is vectorization, but also target specific loop unrolling, or some loop transformations (e.g., distribution) that expose more vectorization opportunities.


Inliner
LLVM IR
Canonicalization - Mem2Reg - - InstCombine - - CFGSimplify -
Scalar Simplifcation - InstCombine - - CFGSimplify -
Simple Loop Opts - Loop Rotate - - Loop Unswitch - - Loop Delete - - Loop Unroll -
Target Specialization - Loop Vectorization - - Loop Distribution - - SLP Vectorization -
LLVM IR

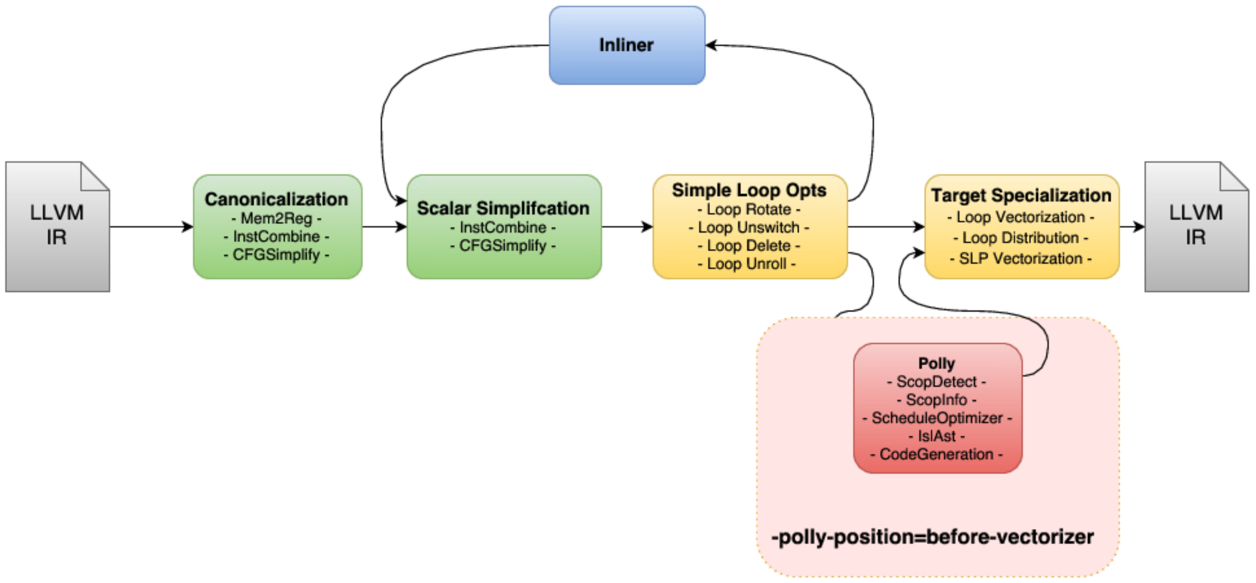
Polly can conceptually be run at three different positions in the pass pipeline. As an early optimizer before the standard LLVM pass pipeline, as a later optimizer as part of the target specialization sequence, and theoretically also with the loop optimizations in the inliner cycle. We only discuss the first two options, as running Polly in the inline loop, is likely to disturb the inliner and is consequently not a good idea.


Inliner
LLVM IR
Canonicalization - Mem2Reg - - InstCombine - - CFGSimplify -
Scalar Simplifcation - InstCombine - - CFGSimplify -
Simple Loop Opts - Loop Rotate - - Loop Unswitch - - Loop Delete - - Loop Unroll -
Target Specialization - Loop Vectorization - - Loop Distribution - - SLP Vectorization -
LLVM IR
Polly Canonicalize - Mem2Reg - - InstCombine - - CFGSimplify -
Inliner
Polly - ScopDetect - - ScopInfo - - ScheduleOptimizer - - IslAst - - CodeGeneration -
-polly-position=early
Polly - ScopDetect - - ScopInfo - - ScheduleOptimizer - - IslAst - - CodeGeneration -
-polly-position=before-vectorizer

Running Polly early before the standard pass pipeline has the benefit that the LLVM-IR processed by Polly is still very close to the original input code. Hence, it is less likely that transformations applied by LLVM change the IR in ways not easily understandable for the programmer. As a result, user feedback is likely better and it is less likely that kernels that in C seem a perfect fit for Polly have been transformed such that Polly can not handle them any more. On the other hand, codes that require inlining to be optimized won't benefit if Polly is scheduled at this position. The additional set of canonicalization passes required will result in a small, but general compile time increase and some random run-time performance changes due to slightly different IR being passed through the optimizers. To force Polly to run early in the pass pipleline use the option *-polly-position=early* (default today).


Inliner
LLVM IR
Canonicalization - Mem2Reg - - InstCombine - - CFGSimplify -
Scalar Simplifcation - InstCombine - - CFGSimplify -
Simple Loop Opts - Loop Rotate - - Loop Unswitch - - Loop Delete - - Loop Unroll -
Target Specialization - Loop Vectorization - - Loop Distribution - - SLP Vectorization -
LLVM IR
Polly Canonicalize - Mem2Reg - - InstCombine - - CFGSimplify -
Inliner
Polly - ScopDetect - - ScopInfo - - ScheduleOptimizer - - IslAst - - CodeGeneration -
-polly-position=early

Running Polly right before the vectorizer has the benefit that the full inlining cycle has been run and as a result even heavily templated C++ code could theoretically benefit from Polly (more work is necessary to make Polly here really effective). As the IR that is passed to Polly has already been canonicalized, there is also no need to run additional canonicalization passes. General compile time is almost not affected by Polly, as detection of loop kernels is generally very fast and the actual optimization and cleanup passes are only run on functions which contain loop kernels that are worth optimizing. However, due to the many optimizations that LLVM runs before Polly the IR that reaches Polly often has additional scalar dependences that make Polly a lot less efficient. To force Polly to run before the vectorizer in the pass pipleline use the option *-polly-position=before-vectorizer*. This position is not yet the default for Polly, but work is on its way to be effective even in presence of scalar dependences. After this work has been completed, Polly will likely use this position by default.


Inliner
LLVM IR
Canonicalization - Mem2Reg - - InstCombine - - CFGSimplify -
Scalar Simplifcation - InstCombine - - CFGSimplify -
Simple Loop Opts - Loop Rotate - - Loop Unswitch - - Loop Delete - - Loop Unroll -
Target Specialization - Loop Vectorization - - Loop Distribution - - SLP Vectorization -
LLVM IR
Polly - ScopDetect - - ScopInfo - - ScheduleOptimizer - - IslAst - - CodeGeneration -
-polly-position=before-vectorizer

2.2 Using Polly with Clang

This documentation discusses how Polly can be used in Clang to automatically optimize C/C++ code during compilation.

Warning: Warning: clang/LLVM/Polly need to be in sync (compiled from the same SVN revision).

2.2.1 Make Polly available from Clang

Polly is available through clang, opt, and bugpoint, if Polly was checked out into tools/polly before compilation. No further configuration is needed.

2.2.2 Optimizing with Polly

Optimizing with Polly is as easy as adding -O3 -mllvm -polly to your compiler flags (Polly is only available at -O3).

```
clang -O3 -mllvm -polly file.c
```

2.2.3 Automatic OpenMP code generation

To automatically detect parallel loops and generate OpenMP code for them you also need to add -mllvm -polly-parallel -lgomp to your CFLAGS.

```
clang -O3 -mllvm -polly -mllvm -polly-parallel -lgomp file.c
```

2.2.4 Automatic Vector code generation

Automatic vector code generation can be enabled by adding -mllvm -polly-vectorizer=stripmine to your CFLAGS.

```
clang -O3 -mllvm -polly -mllvm -polly-vectorizer=stripmine file.c
```

2.2.5 Isolate the Polly passes

Polly's analysis and transformation passes are run with many other passes of the pass manager's pipeline. Some of passes that run before Polly are essential for its working, for instance the canonicalization of loop. Therefore Polly is unable to optimize code straight out of clang's -O0 output.

To get the LLVM-IR that Polly sees in the optimization pipeline, use the command:

```
clang file.c -c -O3 -mllvm -polly -mllvm -polly-dump-before-file=before-polly.ll
```

This writes a file 'before-polly.ll' containing the LLVM-IR as passed to polly, after SSA transformation, loop canonicalization, inlining and other passes.

Thereafter, any Polly pass can be run over 'before-polly.ll' using the 'opt' tool. To found out which Polly passes are active in the standard pipeline, see the output of

```
clang file.c -c -O3 -mllvm -polly -mllvm -debug-pass=Arguments
```

The Polly's passes are those between '-polly-detect' and '-polly-codegen'. Analysis passes can be omitted. At the time of this writing, the default Polly pass pipeline is:

```
opt before-polly.ll -polly-simplify -polly-optree -polly-delicm -polly-simplify -
↪polly-prune-unprofitable -polly-opt-isl -polly-codegen
```

Note that this uses LLVM's old/legacy pass manager.

For completeness, here are some other methods that generates IR suitable for processing with Polly from C/C++/Objective C source code. The previous method is the recommended one.

The following generates unoptimized LLVM-IR ('-O0', which is the default) and runs the canonicalizing passes on it ('-polly-canonicalize'). This does /not/ include all the passes that run before Polly in the default pass pipeline. The '-disable-O0-optnone' option is required because otherwise clang adds an 'optnone' attribute to all functions such that it is skipped by most optimization passes. This is meant to stop LTO builds to optimize these functions in the linking phase anyway.

```
clang file.c -c -O0 -Xclang -disable-O0-optnone -emit-llvm -S -o - | opt -polly-
↪canonicalize -S
```

The option '-disable-llvm-passes' disables all LLVM passes, even those that run at -O0. Passing -O1 (or any optimization level other than -O0) avoids that the 'optnone' attribute is added.

```
clang file.c -c -O1 -Xclang -disable-llvm-passes -emit-llvm -S -o - | opt -polly-
↪canonicalize -S
```

As another alternative, Polly can be pushed in front of the pass pipeline, and then its output dumped. This implicitly runs the '-polly-canonicalize' passes.

```
clang file.c -c -O3 -mllvm -polly -mllvm -polly-position=early -mllvm -polly-dump-
↪before-file=before-polly.ll
```

2.2.6 Further options

Polly supports further options that are mainly useful for the development or the analysis of Polly. The relevant options can be added to clang by appending -mllvm -option-name to the CFLAGS or the clang command line.

Limit Polly to a single function

To limit the execution of Polly to a single function, use the option -polly-only-func=functionname.

Disable LLVM-IR generation

Polly normally regenerates LLVM-IR from the Polyhedral representation. To only see the effects of the preparing transformation, but to disable Polly code generation add the option polly-no-codegen.

Graphical view of the SCoPs

Polly can use graphviz to show the SCoPs it detects in a program. The relevant options are -polly-show, -polly-show-only, -polly-dot and -polly-dot-only. The 'show' options automatically run dotty or another graphviz viewer to show the scops graphically. The 'dot' options store for each function a dot file that highlights the detected SCoPs. If 'only' is appended at the end of the option, the basic blocks are shown without the statements the contain.

Change/Disable the Optimizer

Polly uses by default the isl scheduling optimizer. The isl optimizer optimizes for data-locality and parallelism using the Pluto algorithm. To disable the optimizer entirely use the option -polly-optimizer=none.

Disable tiling in the optimizer

By default both optimizers perform tiling, if possible. In case this is not wanted the option -polly-tiling=false can be used to disable it. (This option disables tiling for both optimizers).

Import / Export

The flags -polly-import and -polly-export allow the export and reimport of the polyhedral representation. By exporting, modifying and reimporting the polyhedral representation externally calculated transformations can be applied. This enables external optimizers or the manual optimization of specific SCoPs.

Viewing Polly Diagnostics with opt-viewer

The flag -fsave-optimization-record will generate .opt.yaml files when compiling your program. These yaml files contain information about each emitted remark. Ensure that you have Python 2.7 with PyYaml and Pygments Python Packages. To run opt-viewer:

```
llvm/tools/opt-viewer/opt-viewer.py -source-dir /path/to/program/src/ \
   /path/to/program/src/foo.opt.yaml \
   /path/to/program/src/bar.opt.yaml \
   -o ./output
```

Include all yaml files (use *.opt.yaml when specifying which yaml files to view) to view all diagnostics from your program in opt-viewer. Compile with PGO to view Hotness information in opt-viewer. Resulting html files can be viewed in an internet browser.

2.3 How to manually use the Individual pieces of Polly

2.3.1 Execute the individual Polly passes manually

This example presents the individual passes that are involved when optimizing code with Polly. We show how to execute them individually and explain for each which analysis is performed or what transformation is applied. In this example the polyhedral transformation is user-provided to show how much performance improvement can be expected by an optimal automatic optimizer.

1. Create LLVM-IR from the C code

Polly works on LLVM-IR. Hence it is necessary to translate the source files into LLVM-IR. If more than one file should be optimized the files can be combined into a single file with llvm-link.

```
clang -S -emit-llvm matmul.c -o matmul.s
```

2. Prepare the LLVM-IR for Polly

Polly is only able to work with code that matches a canonical form. To translate the LLVM-IR into this form we use a set of canonicalication passes. They are scheduled by using '-polly-canonicalize'.

```
opt -S -polly-canonicalize matmul.s > matmul.preopt.ll
```

3. Show the SCoPs detected by Polly (optional)

To understand if Polly was able to detect SCoPs, we print the structure of the detected SCoPs. In our example two SCoPs are detected. One in 'init_array' the other in 'main'.

```
$ opt -polly-ast -analyze -q matmul.preopt.ll -polly-process-unprofitable
```

```
:: isl ast :: init_array :: %for.cond1.preheader---%for.end19

if (1)

    for (int c0 = 0; c0 <= 1535; c0 += 1)
      for (int c1 = 0; c1 <= 1535; 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 <= 1535; c0 += 1)
```

(continues on next page)

(continued from previous page)

```
      for (int c1 = 0; c1 <= 1535; c1 += 1) {
        Stmt_for_body3(c0, c1);
        for (int c2 = 0; c2 <= 1535; c2 += 1)
          Stmt_for_body8(c0, c1, c2);
      }

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

4. Highlight the detected SCoPs in the CFGs of the program (requires graphviz/dotty)

Polly can use graphviz to graphically show a CFG in which the detected SCoPs are highlighted. It can also create '.dot' files that can be translated by the 'dot' utility into various graphic formats.

```
$ opt -view-scops -disable-output matmul.preopt.ll
$ opt -view-scops-only -disable-output matmul.preopt.ll
```

The output for the different functions:

- view-scops : main, init_array, print_array
- view-scops-only : main-scopsonly, init_array-scopsonly, print_array-scopsonly

5. View the polyhedral representation of the SCoPs

```
$ opt -polly-scops -analyze matmul.preopt.ll -polly-process-unprofitable
```

```
[...]Printing analysis 'Polly - Create polyhedral description of Scops' for␣
↪region: 'for.cond1.preheader => for.end19' in function 'init_array':
    Function: init_array
    Region: %for.cond1.preheader---%for.end19
    Max Loop Depth:  2
        Invariant Accesses: {
        }
        Context:
        {  :  }
        Assumed Context:
        {  :  }
        Invalid Context:
        {  : 1 = 0 }
        Arrays {
            float MemRef_A[*][1536]; // Element size 4
            float MemRef_B[*][1536]; // Element size 4
        }
        Arrays (Bounds as pw_affs) {
            float MemRef_A[*][ { [] -> [(1536)] } ]; // Element size 4
            float MemRef_B[*][ { [] -> [(1536)] } ]; // Element size 4
        }
        Alias Groups (0):
            n/a
        Statements {
            Stmt_for_body3
                Domain :=
                    { Stmt_for_body3[i0, i1] : 0 <= i0 <= 1535 and 0 <= i1
↪<= 1535 };
```

(continues on next page)

(continued from previous page)

```
                Schedule :=
                    { Stmt_for_body3[i0, i1] -> [i0, i1] };
                MustWriteAccess :=      [Reduction Type: NONE] [Scalar: 0]
                    { Stmt_for_body3[i0, i1] -> MemRef_A[i0, i1] };
                MustWriteAccess :=      [Reduction Type: NONE] [Scalar: 0]
                    { Stmt_for_body3[i0, i1] -> MemRef_B[i0, i1] };
        }
[...]Printing analysis 'Polly - Create polyhedral description of Scops' for␣
↪region: 'for.cond1.preheader => for.end30' in function 'main':
    Function: main
    Region: %for.cond1.preheader---%for.end30
    Max Loop Depth:  3
    Invariant Accesses: {
    }
    Context:
    {  :  }
    Assumed Context:
    {  :  }
    Invalid Context:
    {  : 1 = 0 }
    Arrays {
        float MemRef_C[*][1536]; // Element size 4
        float MemRef_A[*][1536]; // Element size 4
        float MemRef_B[*][1536]; // Element size 4
    }
    Arrays (Bounds as pw_affs) {
        float MemRef_C[*][ { [] -> [(1536)] } ]; // Element size 4
        float MemRef_A[*][ { [] -> [(1536)] } ]; // Element size 4
        float MemRef_B[*][ { [] -> [(1536)] } ]; // Element size 4
    }
    Alias Groups (0):
        n/a
    Statements {
        Stmt_for_body3
            Domain :=
                { Stmt_for_body3[i0, i1] : 0 <= i0 <= 1535 and 0 <= i1 <=␣
↪1535 };
            Schedule :=
                { Stmt_for_body3[i0, i1] -> [i0, i1, 0, 0] };
            MustWriteAccess :=  [Reduction Type: NONE] [Scalar: 0]
                { Stmt_for_body3[i0, i1] -> MemRef_C[i0, i1] };
        Stmt_for_body8
            Domain :=
                { Stmt_for_body8[i0, i1, i2] : 0 <= i0 <= 1535 and 0 <= i1
↪<= 1535 and 0 <= i2 <= 1535 };
            Schedule :=
                { Stmt_for_body8[i0, i1, i2] -> [i0, i1, 1, i2] };
            ReadAccess :=       [Reduction Type: NONE] [Scalar: 0]
                { Stmt_for_body8[i0, i1, i2] -> MemRef_C[i0, i1] };
            ReadAccess :=       [Reduction Type: NONE] [Scalar: 0]
                { Stmt_for_body8[i0, i1, i2] -> MemRef_A[i0, i2] };
            ReadAccess :=       [Reduction Type: NONE] [Scalar: 0]
                { Stmt_for_body8[i0, i1, i2] -> MemRef_B[i2, i1] };
            MustWriteAccess :=  [Reduction Type: NONE] [Scalar: 0]
                { Stmt_for_body8[i0, i1, i2] -> MemRef_C[i0, i1] };
    }
```

6. Show the dependences for the SCoPs

```
$ opt -polly-dependences -analyze matmul.preopt.ll -polly-process-
↪unprofitable
```

```
[...]Printing analysis 'Polly - Calculate dependences' for region: 'for.
↪cond1.preheader => for.end19' in function 'init_array':
        RAW dependences:
                {  }
        WAR dependences:
                {  }
        WAW dependences:
                {  }
        Reduction dependences:
                n/a
        Transitive closure of reduction dependences:
                {  }
[...]Printing analysis 'Polly - Calculate dependences' for region: 'for.
↪cond1.preheader => for.end30' in function 'main':
        RAW dependences:
                { Stmt_for_body3[i0, i1] -> Stmt_for_body8[i0, i1, 0] : 0 <=␣
↪i0 <= 1535 and 0 <= i1 <= 1535; Stmt_for_body8[i0, i1, i2] -> Stmt_for_
↪body8[i0, i1, 1 + i2] : 0 <= i0 <= 1535 and 0 <= i1 <= 1535 and 0 <= i2 <=␣
↪1534 }
        WAR dependences:
                {  }
        WAW dependences:
                { Stmt_for_body3[i0, i1] -> Stmt_for_body8[i0, i1, 0] : 0 <=␣
↪i0 <= 1535 and 0 <= i1 <= 1535; Stmt_for_body8[i0, i1, i2] -> Stmt_for_
↪body8[i0, i1, 1 + i2] : 0 <= i0 <= 1535 and 0 <= i1 <= 1535 and 0 <= i2 <=␣
↪1534 }
        Reduction dependences:
                n/a
        Transitive closure of reduction dependences:
                {  }
```

7. Export jscop files

```
$ opt -polly-export-jscop matmul.preopt.ll -polly-process-unprofitable
```

```
[...]Writing JScop '%for.cond1.preheader---%for.end19' in function 'init_
↪array' to './init_array___%for.cond1.preheader---%for.end19.jscop'.

Writing JScop '%for.cond1.preheader---%for.end30' in function 'main' to './
↪main___%for.cond1.preheader---%for.end30.jscop'.
```

8. Import the changed jscop files and print the updated SCoP structure (optional)

Polly can reimport jscop files, in which the schedules of the statements are changed. These changed schedules are used to descripe transformations. It is possible to import different jscop files by providing the postfix of the jscop file that is imported.

We apply three different transformations on the SCoP in the main function. The jscop files describing these transformations are hand written (and available in docs/experiments/matmul).

No Polly

As a baseline we do not call any Polly code generation, but only apply the normal -O3 optimizations.

```
$ opt matmul.preopt.ll -polly-import-jscop -polly-ast -analyze -polly-
↪process-unprofitable
```

```
[...]
:: isl ast :: main :: %for.cond1.preheader---%for.end30

if (1)

    for (int c0 = 0; c0 <= 1535; c0 += 1)
      for (int c1 = 0; c1 <= 1535; c1 += 1) {
        Stmt_for_body3(c0, c1);
        for (int c3 = 0; c3 <= 1535; c3 += 1)
          Stmt_for_body8(c0, c1, c3);
      }

else
    {  /* original code */ }
[...]
```

Loop Interchange (and Fission to allow the interchange)

We split the loops and can now apply an interchange of the loop dimensions that enumerate Stmt_for_body8.

```
$ opt matmul.preopt.ll -polly-import-jscop -polly-import-jscop-
↪postfix=interchanged -polly-ast -analyze -polly-process-unprofitable
```

```
[...]
:: isl ast :: main :: %for.cond1.preheader---%for.end30

if (1)

    {
      for (int c1 = 0; c1 <= 1535; c1 += 1)
        for (int c2 = 0; c2 <= 1535; c2 += 1)
          Stmt_for_body3(c1, c2);
      for (int c1 = 0; c1 <= 1535; c1 += 1)
        for (int c2 = 0; c2 <= 1535; c2 += 1)
          for (int c3 = 0; c3 <= 1535; c3 += 1)
            Stmt_for_body8(c1, c3, c2);
    }

else
    {  /* original code */ }
[...]
```

Interchange + Tiling

In addition to the interchange we now tile the second loop nest.

```
$ opt matmul.preopt.ll -polly-import-jscop -polly-import-jscop-
↪postfix=interchanged+tiled -polly-ast -analyze -polly-process-unprofitable
```

```
[...]
:: isl ast :: main :: %for.cond1.preheader---%for.end30

if (1)

    {
      for (int c1 = 0; c1 <= 1535; c1 += 1)
        for (int c2 = 0; c2 <= 1535; c2 += 1)
          Stmt_for_body3(c1, c2);
      for (int c1 = 0; c1 <= 1535; c1 += 64)
        for (int c2 = 0; c2 <= 1535; c2 += 64)
          for (int c3 = 0; c3 <= 1535; c3 += 64)
            for (int c4 = c1; c4 <= c1 + 63; c4 += 1)
              for (int c5 = c3; c5 <= c3 + 63; c5 += 1)
                for (int c6 = c2; c6 <= c2 + 63; c6 += 1)
                  Stmt_for_body8(c4, c6, c5);
    }

else
    {  /* original code */ }
[...]
```

Interchange + Tiling + Strip-mining to prepare vectorization

To later allow vectorization we create a so called trivially parallelizable loop. It is innermost, parallel and has only four iterations. It can be replaced by 4-element SIMD instructions.

```
$ opt matmul.preopt.ll -polly-import-jscop -polly-import-jscop-
↪postfix=interchanged+tiled -polly-ast -analyze -polly-process-unprofitable
```

```
[...]
:: isl ast :: main :: %for.cond1.preheader---%for.end30

if (1)

    {
      for (int c1 = 0; c1 <= 1535; c1 += 1)
        for (int c2 = 0; c2 <= 1535; c2 += 1)
          Stmt_for_body3(c1, c2);
      for (int c1 = 0; c1 <= 1535; c1 += 64)
        for (int c2 = 0; c2 <= 1535; c2 += 64)
          for (int c3 = 0; c3 <= 1535; c3 += 64)
            for (int c4 = c1; c4 <= c1 + 63; c4 += 1)
              for (int c5 = c3; c5 <= c3 + 63; c5 += 1)
                for (int c6 = c2; c6 <= c2 + 63; c6 += 4)
                  for (int c7 = c6; c7 <= c6 + 3; c7 += 1)
                    Stmt_for_body8(c4, c7, c5);
    }

else
    {  /* original code */ }
[...]
```

9. Codegenerate the SCoPs

This generates new code for the SCoPs detected by polly. If -polly-import-jscop is present, transformations specified in the imported jscop files will be applied.

```
$ opt matmul.preopt.ll | opt -O3 > matmul.normalopt.ll
```

```
$ opt matmul.preopt.ll -polly-import-jscop -polly-import-jscop-
↪postfix=interchanged -polly-codegen -polly-process-unprofitable | opt -O3 >
↪ matmul.polly.interchanged.ll
```

```
Reading JScop '%for.cond1.preheader---%for.end19' in function 'init_array'␣
↪from './init_array___%for.cond1.preheader---%for.end19.jscop.interchanged'.
File could not be read: No such file or directory
Reading JScop '%for.cond1.preheader---%for.end30' in function 'main' from './
↪main___%for.cond1.preheader---%for.end30.jscop.interchanged'.
```

```
$ opt matmul.preopt.ll -polly-import-jscop -polly-import-jscop-
↪postfix=interchanged+tiled -polly-codegen -polly-process-unprofitable |␣
↪opt -O3 > matmul.polly.interchanged+tiled.ll
```

```
Reading JScop '%for.cond1.preheader---%for.end19' in function 'init_array'␣
↪from './init_array___%for.cond1.preheader---%for.end19.jscop.
↪interchanged+tiled'.
File could not be read: No such file or directory
Reading JScop '%for.cond1.preheader---%for.end30' in function 'main' from './
↪main___%for.cond1.preheader---%for.end30.jscop.interchanged+tiled'.
```

```
$ opt matmul.preopt.ll -polly-import-jscop -polly-import-jscop-
↪postfix=interchanged+tiled+vector -polly-codegen -polly-vectorizer=polly -
↪polly-process-unprofitable | opt -O3 > matmul.polly.
↪interchanged+tiled+vector.ll
```

```
Reading JScop '%for.cond1.preheader---%for.end19' in function 'init_array'␣
↪from './init_array___%for.cond1.preheader---%for.end19.jscop.
↪interchanged+tiled+vector'.
File could not be read: No such file or directory
Reading JScop '%for.cond1.preheader---%for.end30' in function 'main' from './
↪main___%for.cond1.preheader---%for.end30.jscop.interchanged+tiled+vector'.
```

```
$ opt matmul.preopt.ll -polly-import-jscop -polly-import-jscop-
↪postfix=interchanged+tiled+vector -polly-codegen -polly-vectorizer=polly -
↪polly-parallel -polly-process-unprofitable | opt -O3 > matmul.polly.
↪interchanged+tiled+openmp.ll
```

```
Reading JScop '%for.cond1.preheader---%for.end19' in function 'init_array'␣
↪from './init_array___%for.cond1.preheader---%for.end19.jscop.
↪interchanged+tiled+vector'.
File could not be read: No such file or directory
Reading JScop '%for.cond1.preheader---%for.end30' in function 'main' from './
↪main___%for.cond1.preheader---%for.end30.jscop.interchanged+tiled+vector'.
```

10. Create the executables

```
$ llc matmul.normalopt.ll -o matmul.normalopt.s && gcc matmul.normalopt.s -o␣
↪matmul.normalopt.exe
$ llc matmul.polly.interchanged.ll -o matmul.polly.interchanged.s && gcc␣
↪matmul.polly.interchanged.s -o matmul.polly.interchanged.exe
$ llc matmul.polly.interchanged+tiled.ll -o matmul.polly.interchanged+tiled.
↪s && gcc matmul.polly.interchanged+tiled.s -o matmul.polly.
↪interchanged+tiled.exe
$ llc matmul.polly.interchanged+tiled+vector.ll -o matmul.polly.
↪interchanged+tiled+vector.s && gcc matmul.polly.interchanged+tiled+vector.
↪s -o matmul.polly.interchanged+tiled+vector.exe
$ llc matmul.polly.interchanged+tiled+vector+openmp.ll -o matmul.polly.
↪interchanged+tiled+vector+openmp.s && gcc -fopenmp matmul.polly.
↪interchanged+tiled+vector+openmp.s -o matmul.polly.
↪interchanged+tiled+vector+openmp.exe
```

11. Compare the runtime of the executables

By comparing the runtimes of the different code snippets we see that a simple loop interchange gives here the largest performance boost. However in this case, adding vectorization and using OpenMP degrades the performance.

```
$ time ./matmul.normalopt.exe

real    0m11.295s
user    0m11.288s
sys     0m0.004s
$ time ./matmul.polly.interchanged.exe

real    0m0.988s
user    0m0.980s
sys     0m0.008s
$ time ./matmul.polly.interchanged+tiled.exe

real    0m0.830s
user    0m0.816s
sys     0m0.012s
$ time ./matmul.polly.interchanged+tiled+vector.exe

real    0m5.430s
user    0m5.424s
sys     0m0.004s
$ time ./matmul.polly.interchanged+tiled+vector+openmp.exe

real    0m3.184s
user    0m11.972s
sys     0m0.036s
```

2.4 Tips and Tricks on using and contributing to Polly

2.4.1 Commiting to polly trunk

- General reference to git-svn workflow

2.4.2 Using bugpoint to track down errors in large files

If you know the `opt` invocation and have a large `.ll` file that causes an error, `bugpoint` allows one to reduce the size of test cases.

The general calling pattern is:

- `$ bugpoint <file.ll> <pass that causes the crash> -opt-args <opt option flags>`

An example invocation is:

- `$ bugpoint crash.ll -polly-codegen -opt-args -polly-canonicalize -polly-process-unprofitable`

For more documentation on bugpoint, Visit the LLVM manual

2.4.3 Understanding which pass makes a particular change

If you know that something like *opt -O3 -polly* makes a change, but you wish to isolate which pass makes a change, the steps are as follows:

- `$ bugpoint -O3 file.ll -opt-args -polly` will allow bugpoint to track down the pass which causes the crash.

To do this manually:

- `$ opt -O3 -polly -debug-pass=Arguments` to get all passes that are run by default. `-debug-pass=Arguments` will list all passes that have run.
- Bisect down to the pass that changes it.

2.4.4 Debugging regressions introduced at some unknown earlier point

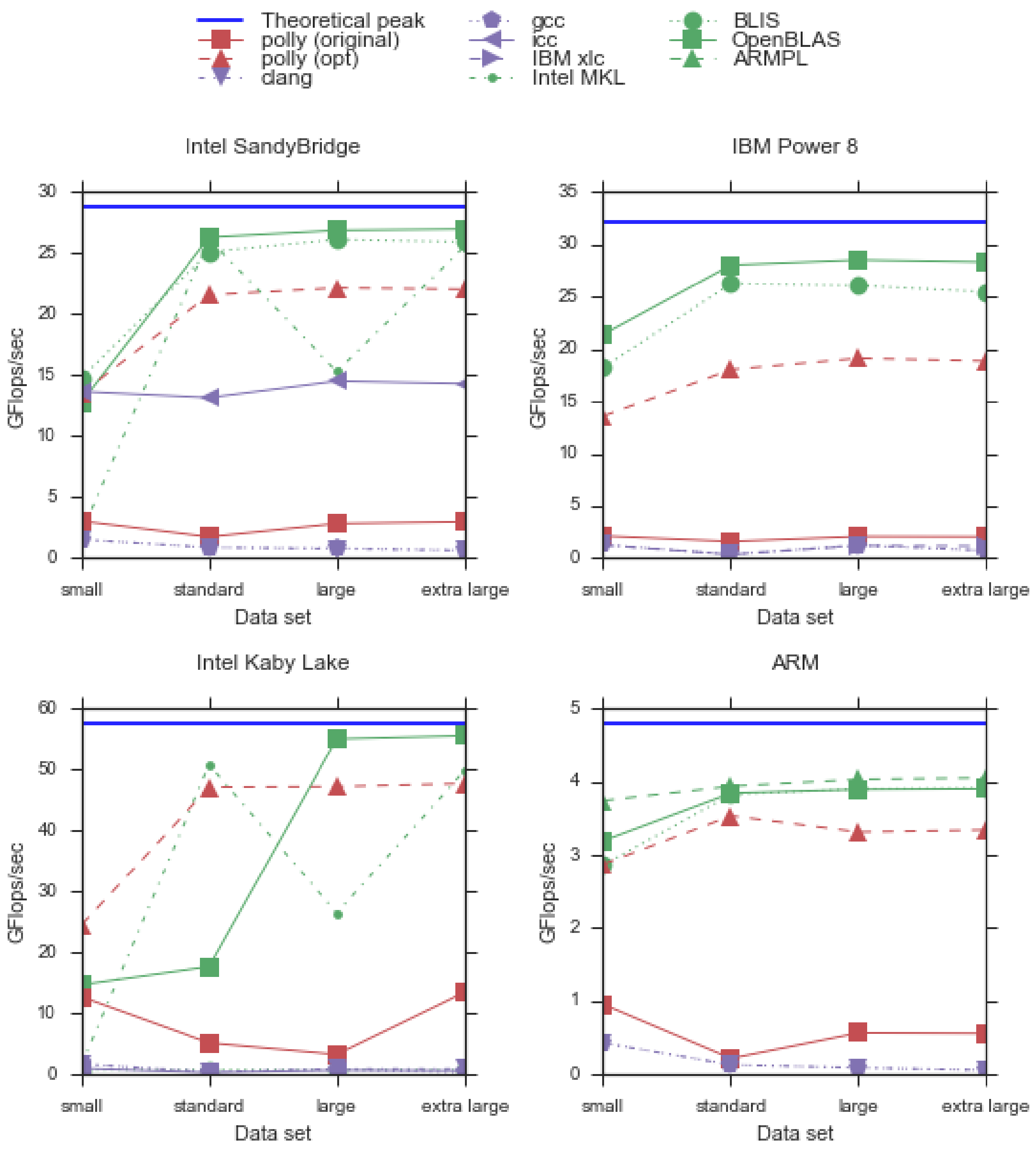
In case of a regression in performance or correctness (e.g., an earlier version of Polly behaved as expected and a later version does not), bisecting over the version history is the standard approach to identify the commit that introduced the regression.

LLVM has a single repository that contains all projects. It can be cloned at: https://github.com/llvm-project/llvm-project-20170507. How to bisect on a git repository is explained here https://www.metaltoad.com/blog/beginners-guide-git-bisect-process-elimination. The bisect process can also be automated as explained here: https://www.metaltoad.com/blog/mechanizing-git-bisect-bug-hunting-lazy. An LLVM specific run script is available here: https://gist.github.com/dcci/891cd98d80b1b95352a407d80914f7cf.

2.5 Performance

2.5.1 High-Performance Generalized Matrix Multiplication

Polly automatically detects and optimizes generalized matrix multiplication, the computation $C \leftarrow \alpha \quad C \quad \beta \quad A \quad B$, where A, B, and C are three appropriately sized matrices, and operations are originating from the corresponding matrix semiring, and α and β are constants, and beta is not equal to zero. It allows to obtain the highly optimized form structured similar to the expert implementation of GEMM that can be found in GotoBLAS and its successors. The performance evaluation of GEMM is shown in the following figure.



2.5.2 Compile Time Impact of Polly

Clang+LLVM+Polly are compiled using Clang on a Intel(R) Core(TM) i7-7700 based system. The experiment is repeated twice: with and without Polly enabled in order to measure its compile time impact.

The following versions are used:

- Polly (git hash 0db98a4837b6f233063307bb9184374175401922)
- Clang (git hash 3e1d04a92b51ed36163995c96c31a0e4bbb1561d)
- LLVM git hash 0265ec7ebad69a47f5c899d95295b5eb41aba68e)

ninja is used as the build system.

For both cases the whole compilation was performed five times. The compile times in seconds are shown in the following table.

Polly Disabled	Polly Enabled
964	977
964	980
967	981
967	981
968	982

The median compile time without Polly enabled is 967 seconds and with Polly enabled it is 981 seconds. The overhead is 1.4%.

- A list of Polly passes

CHAPTER 3

Indices and tables

- genindex
- search