PolyOpt/Fortran

A Polyhedral Optimizer for Fortran in the ROSE compiler
Edition 0.1, for PolyOpt/Fortran 0.1.0
Feb 21st 2012

Louis-Noël Pouchet
Mohanish Narayan

This manual is dedicated to PolyOpt/Fortran version 0.1.0, a framework for Polyhedral Optimization for Fortran in the ROSE compiler.

Table of Contents

1 Introduction

PolyOpt/Fortran is a polyhedral loop optimization framework for Fortran, integrated in the ROSE compiler. It is based on PolyOpt, a polyhedral loop optimization framework for C, also implemented in ROSE. (To avoid any ambiguity, the existing optimization framework for C will be referred to as PolyOpt/C.)

PolyOpt/Fortran 0.1.0 is an extension of PolyOpt/C which adds Fortran specific capabilities. Although the Fortran functionality will eventually be merged with the main development of PolyOpt, for now it is still in experimental stage and remains a separate project. This documentation of PolyOpt/Fortran is based on the documentation of PolyOpt/C, created by Louis-Noel Pouchet. The Fortran-specific parts of the documentation were written by Mohanish Narayan.

The main features of PolyOpt/Fortran are:

- Automatic extraction of regions that can be optimized in the polyhedral model
- Full support of PoCC (the Polyhedral Compiler Collection) analysis and optimizations
  - Dependence analysis with Candl
  - Program transformations for tiling and parallelism with Pluto
  - Code generation with CLooG
  - Parametric tiling with PTile
  - Numerous under-the-hood functionalities and optimizations

Note: only a subset of Fortran is currently supported by PolyOpt/Fortran. If analysis of C programs is needed, PolyOpt/C should be used instead. Analysis of C++ programs is not supported.

Communication: Please contact directly Louis-Noel Pouchet pouchet@cse.ohio-state.edu or Mohanish Narayan narayanm@cse.ohio-state.edu for any questions. PoCC is also available as a stand-alone software on sourceforge

2 Specifics of Polyhedral Programs

2.1 Static Control Parts

Sequences of (possibly imperfectly nested) loops amenable to polyhedral optimization are called *static control parts* (SCoP) [5], roughly defined as a set of consecutive statements such that all loop bounds and conditionals are affine functions of the surrounding loop iterators and parameters (variables whose value is unknown at compile time but invariant in the loop nest considered). In addition, for effective data dependence analysis we require the array access functions to also be affine functions of the surrounding iterators and parameters.

For instance, a valid affine expression for a conditional or a loop bound in a SCoP with three loops iterators *i,j,k* and two parameters *N,P* will be of the form *a.i* + *b.j* + *c.k* + *d.N* + *e.P* + *f*, *a,b,c,d,e,f* are arbitrary (possibly 0) integer numbers.

The following program is a SCoP:

```
do i = 1, N
   do j = 1, N
      A(j,i) = A(j,i) + u1(i)*v1(j)
      if(N - i > 2) then
         A(j,i) = A(j,i) - 2
      end if
   end do
end do
```

Numerous elements can break the SCoP property, for instance:

- `if` conditionals involving variables that are not loop iterators or a parameters, e.g., `if (A(i,j) == 0)`.
- `if` conditionals involving loop iterators and/or a parameter to form a non-affine expression, e.g., `if (i * j == 0)`.
- Non-affine `do` initialization or test condition, e.g., `do j = 1, i*i`.
- Non-affine array access, e.g., `A(i*N,j % i)` or `A(B(i))`.
- Use of `division` operator, e.g., `A(j / i)` or `do i = x / 3, 10)`.

2.2 Additional Restrictions in PolyOpt/Fortran

PolyOpt/Fortran automatically extracts maximal regions that can be optimized in the Polyhedral framework. We enforce numerous additional constraints to ensure the correctness of the SCoP extraction, in particular due to dependence analysis consideration:

- The only allowed control statements are `do` and `if`.
- There is no function call in the SCoP, unless it is a Fortran intrinsic math function with no side-effects.
- `goto`, `exit` and `cycle` statements are forbidden.
- Arrays represent distinct memory locations (one per accessed array cell), and arrays are not aliased (note: **no check is performed by PolyOpt/Fortran for this property, it is the responsibility of the user to not feed ill-formed arrays**).
- Loops increment by step of one.

2.3 Allowed Control-Flow Operations

PolyOpt/Fortran supports a wide range of affine expressions, in particular conjunctions of affine constraints can be used to bound the space. In all the following, we recall that an affine expression must involve only surrounding loop iterators and parameters (scalar variables that are invariant during the SCoP execution).

SCoP extraction is a syntactic process so there are clear definitions of what is allowed in `do init , test , increment` and `if (condition)` statements. We note that if the loop iterator of a `do` statement is used outside the scope of the loop, or is assigned in the loop body, the *loop will conservatively not be considered for SCoP extraction* since PolyOpt/Fortran may change the exit value of loop iterators.

Note: `do` loops can be terminated with either an `end do` or a labelled `continue` statement. The latter is accepted as a valid loop only if there is no reference to the label (used in the `continue` statement) except in the corresponding `do` statement. In the output code, the `continue` statement will be replaced with an `end do` statement.

2.3.1 In `do` initialization statement

`init` can be of the form `var = expressionLb`. The `expressionLb` is an affine expression, or possibly a conjunction of expressions with the `max(expr1, expr2)` operator.

As an illustration, all loops in the following code form a valid SCoP.

```
   do i = max(max(N,M),P), N + P
      do 10 j = max(i-2,0), N
         A(j,i) = A(j,i) - 2
10    continue
   end do
```

Some examples of incorrect loop lower bound include:

- `do i = max(a,b) + max(c,d), ...`: not a valid conjunction.

- `do i = max(a,b) + P, ...`: not a valid conjunction.

2.3.2 In `do` test statement

`test` must be of the form `expressionUb`. The `expressionUb` is an affine expression, or possibly a conjunction of expressions with the `min(expr1, expr2)` operator.

As an illustration, all loops in the following code form a valid SCoP.

```
do i = 1, min(min(P,Q),R)
   do j = 1, min(i,P)
      A(j,i) = A(j,i) - 2
   end do
end do
```

Some examples of incorrect loop upper bound include:

- `do i = 1, min(a,b) + min(c,d), ...`: not a valid conjunction.

2.3.3 In `do` increment statement

Loops must increment by step of one.

2.3.4 In `if` conditional statement

For `if` statements, the conditional expression can be an arbitrary affine expression, or a conjunction of affine expressions with the `.AND.` operator. `min` and `max` are currently not allowed. Future releases may allow `min` and `max` provided a `max` constrains the lower bound of a variable and a `min` constraints the upper bound of a variable. Most standard comparison operators are allowed: `<, <=, ==, >=, >`. Note that `else` clause is not allowed, nor is `!=`.

As an illustration, all loops in the following code form a valid SCoP.

```
if (i > max(M,N) .AND. j == 0)
   if (k < 32 && k < min(max(i,j),P))
      A(i,j) = 42;
```

Some examples of incorrect conditional expressions include:

- `if (i == 0 .OR. c == 0)`: disjunctions are not allowed.

- `if (i < max(A,B))`: not a valid `max` constraint.
- `if (i == 42/5)`: not an integer term.

2.3.5 Examples

We conclude by showing some examples of SCoPs automatically detected by PolyOpt/Fortran. Note that the only constraints for the statements (e.g., `R,S,T` in the next example) involve the lack of unsafe function calls (functions not recognized by PolyOpt/Fortran as side-effects free) , *at most one variable is assigned in the statement*, and using affine functions to dereference arrays.

```
  alpha = 43532;
  beta = 12313;
  do i = 1, N
R:      v1(i) = (i+1)/N/4.0;
S:      w(i) = 0.0;
        do j = 1, N
T:         A(i,j) = ((DATA_TYPE) i*j) / N;
        end do
  end do
```

```
  do j = 2, M
      stddev(j) = 0.0;
      do i = 1, N
       stddev(j) = stddev(j) + (data(i,j) - mean(j)) * (data(i,j) - mean(j));
      end do
      stddev(j) /= float_n;
      stddev(j) = sqrt(stddev(j));
      if(stddev(j) .LE. eps) then
         stddev(j) = 1.0D0
      end if
   end do
```

3 Optimization Paths

Three main optimization paths are available in PolyOpt/Fortran. They are geared towards improving data locality for fewer data cache misses, and both coarse- and fine-grain shared memory parallelization with OpenMP. They will be applied on all Static Control Parts that were automatically detected in the input program. Program transformations are generated via the PoCC polyhedral engine.

3.1 `--polyopt-fixed-tiling`

3.1.1 Description

This path automatically computes and applies a complex, SCoP-specific sequence of loop transformations to enable parallel blocked (if possible) execution of the SCoP. The default tile size is 32, and can be specified at compile time only. Parallel loops are marked with OpenMP pragmas, inner-most vectorizable loops are marked with ivdep pragmas. Parallel or pipeline-parallel tile execution is achieved when tiling is possible.

The Pluto module is used to compute the loop transformation sequence, in the form of a series of affine multidimensional schedules.

Giving the flag `--polyopt-fixed-tiling` to PolyOpt/Fortran is equivalent to giving the sequence:

- `--polyopt-pluto-fuse-smartfuse`
- `--polyopt-pluto-tile`
- `--polyopt-pluto-parallel`
- `--polyopt-pluto-prevector`
- `--polyopt-generate-pragmas`

3.1.2 Example

Given the input program:

```
do i = 1, n
  do j = 1, n
     C(j,i) = C(j,i) * beta
     do k = 1, n
        C(j,i) = C(j,i) + A(k,i) * B(j,k) * alpha
     end do
  end do
end do
```

One can *optionally* specify a file to set the tile sizes, to override the default *32* value. This file must be called `tile.sizes`, and be stored in the current working directory. It must contain one tile size per dimension to be tiled. For example:

```
$> cat tile.sizes
16 64 1
```

The result of `--polyopt-fixed-tiling` on the above example, with the specified `tile.sizes` file is shown below. Note, if a `tile.sizes` file exists in the current working directory it will *always* be used.

```
      IF (n >= 1) THEN
!$omp parallel do  private(c2, c6, c4)
      DO c1 = 0, floor(real(n) / real(16)), 1
      DO c2 = 0, floor(real(n) / real(64)), 1
      DO c4 = max(1,16 * c1), min(n,16 * c1 + 15), 1
!dir$ ivdep
!dir$ vector always
!dir$ simd
      DO c6 = max(1,64 * c2), min(n,64 * c2 + 63), 1
      C(c6,c4) = C(c6,c4) * beta
      END DO
      END DO
      END DO
      END DO
!$omp end parallel do
!$omp parallel do  private(c3, c2, c6, c4)
      DO c1 = 0, floor(real(n) / real(16)), 1
      DO c2 = 0, floor(real(n) / real(64)), 1
      DO c3 = 1, n, 1
      DO c4 = max(1,16 * c1), min(n,16 * c1 + 15), 1
!dir$ ivdep
!dir$ vector always
!dir$ simd
      DO c6 = max(1,64 * c2), min(n,64 * c2 + 63), 1
      C(c6,c4) = C(c6,c4) + A(c3,c4) * B(c6,c3) * alpha
      END DO
      END DO
      END DO
      END DO
      END DO
!$omp end parallel do
      END IF
```

3.2 `--polyopt-parametric-tiling`

3.2.1 Description

NOTE: The parametric tiling path is still experimental, and correctness of the generated code is not guaranteed in all cases. In particular, a known issue is when parametric tiling is applied on a loop nest where the outer loop is sequential (wavefront creation is required) and the inner loops are permutable but not fusable. We are working hard to fix this remaining problem.

To the best of our knowledge, the generated code is correct when all statements in a (possibly imperfectly nested) loop nest can be maximally fused. Remember that polyhedral transformations are automatically computed before the parametric tiling pass to enforce this property on the code when possible. The above issue impacts only program parts where it is not possible to exhibit a polyhedral transformation making either the outer loop parallel, or all loops fusable in the loop nest. This is not a frequent pattern, for instance none of the 28 benchmarks of the PolyBench 2.0 test suite exhibit this issue.

This path automatically computes and applies a complex, SCoP-specific sequence of loop transformations to enable parallel blocked execution of the SCoP. The generated code is parametrically tiled when possible, and the tile sizes can be specified at runtime via the `param_tile_sizes[]` array. By default, the tile sizes are set to 32. Parallel loops are marked with OpenMP pragmas.

The Pluto module is used to compute a loop transformation sequence that makes tiling legal, when possible, and the PTile module performs parametric tiling. Parallel or pipeline-parallel tile execution is achieved if tiling is possible.

Giving the flag `--polyopt-parametric-tiling` to PolyOpt/Fortran is equivalent to giving the sequence:

- `--polyopt-pluto-fuse-smartfuse`
- `--polyopt-pluto-parallel`
- `--polyopt-codegen-use-ptile`
- `--polyopt-codegen-insert-ptile-api`

3.2.2 Example

The Parametric tiling API requires to use the function `ParamTileSizeVectorInit(int*, int, int)` to fill-in the tile sizes. This function takes an array of integers, the number of

tile size parameters, and a unique identifier for the SCoP. This function can be in another compilation unit. It allows to select the tile size at run-time, before the computation starts. A trivial definition of the function can be the following:

```
 subroutine ParamTileSizeVectorInit
&  (param_tile_sizes, nestedLoops, scopId)
 integer nestedLoops, scopId, i
 integer , dimension(50) :: param_tile_sizes
 do i = 1 , nestedLoops
    param_tile_sizes(i) = 32
 end do
 end
```

Note: Parametric tiling without using the Parametric tiling API can be achieved by not providing the `--polyopt-codegen-insert-ptile-api` option, in which case the tile sizes will be initialized to a fixed value.

The result of `--polyopt-parametric-tiling` on the above `dgemm` example is shown below.

```
      INTEGER :: tmpLb
      INTEGER :: tmpUb
      INTEGER :: c2t1
      REAL :: T1c1
      REAL :: T1c2
      INTEGER :: c3t1
      REAL :: T1c3
      INTEGER :: c1t1
      INTEGER :: c3
      INTEGER :: c1
      INTEGER :: c2
      INTEGER :: param_tile_sizes(50)

      CALL ParamTileSizeVectorInit(param_tile_sizes,3,1)
      T1c1 = param_tile_sizes(3)
      T1c2 = param_tile_sizes(2)
      T1c3 = param_tile_sizes(1)
      IF (n >= 1) THEN
      tmpLb = NINT(real(-1 + 1 / T1c1 * 2))
      tmpUb = NINT(real(n * (1 / T1c1)))
!$omp parallel do private(c2t1, c1, c2)
      DO c1t1 = tmpLb, tmpUb, 1
      DO c2t1 = NINT(real(-1 + 1 / T1c2 * 2)), NINT(real(n * (1 / T1c2))), 1
      DO c1 = max(real(c1t1 * T1c1),real(1.00000)),
     *                   min(real(c1t1 * T1c1 + (T1c1 + -1)),real(n)), 1
      DO c2 = max(real(c2t1 * T1c2),real(1.00000)),
     *                   min(real(c2t1 * T1c2 + (T1c2 + -1)),real(n)), 1
      C(c2,c1) = C(c2,c1) * beta
      END DO
      END DO
      END DO
      END DO
      tmpLb = NINT(real(-1 + 1 / T1c1 * 2))
      tmpUb = NINT(real(n * (1 / T1c1)))
!$omp parallel do private(c2t1, c3t1, c1, c2, c3)
      DO c1t1 = tmpLb, tmpUb, 1
      DO c2t1 = NINT(real(-1 + 1 / T1c2 * 2)), NINT(real(n * (1 / T1c2))), 1
      DO c3t1 = NINT(real(-1 + 1 / T1c3 * 2)), NINT(real(n * (1 / T1c3))), 1
      DO c1 = max(real(c1t1 * T1c1),real(1.00000)),
     *                   min(real(c1t1 * T1c1 + (T1c1 + -1)),real(n)), 1
      DO c2 = max(real(c2t1 * T1c2),real(1.00000)),
     *                   min(real(c2t1 * T1c2 + (T1c2 + -1)),real(n)), 1
      DO c3 = max(real(c3t1 * T1c3),real(1.00000)),
     *                   min(real(c3t1 * T1c3 + (T1c3 + -1)),real(n)), 1
      C(c2,c1) = C(c2,c1) + A(c3,c1) * B(c2,c3) * alpha
      END DO
      END DO
      END DO
      END DO
      END DO
      END DO
      END IF
```

3.3 --polyopt-parallel-only

3.3.1 Description

This path automatically computes and applies a complex, SCoP-specific sequence of loop transformations to enable parallel execution of the SCoP while improving data locality. In contrast to the other paths, no tiling is applied on the generated program. Parallel loops are marked with OpenMP pragmas. The Pluto module is used to compute a loop transformation sequence that expose coarse-grain parallelism when possible.

Giving the flag `--polyopt-parallel-only` to PolyOpt/Fortran is equivalent to giving the sequence:

- `--polyopt-pluto-fuse-smartfuse`
- `--polyopt-pluto-parallel`
- `--polyopt-generate-pragmas`

3.3.2 Example

The result of `--polyopt-parallel-only` on the above `dgemm` example is shown below. Note that pre-vectorization is disabled in this mode, fixed tiling must be enabled for it to be active. To prevent the distribution of the two statements, the user can rely on the fine-tuning flags, e.g., `--polyopt-pluto-fuse-maxfuse`.

```
      IF (n >= 1) THEN
!$omp parallel do  private(c2)
      DO c1 = 1, n, 1
      DO c2 = 1, n, 1
      C(c2,c1) = C(c2,c1) * beta
      END DO
      END DO
!$omp end parallel do
!$omp parallel do  private(c3, c2)
      DO c1 = 1, n, 1
      DO c2 = 1, n, 1
      DO c3 = 1, n, 1
      C(c2,c1) = C(c2,c1) + A(c3,c1) * B(c2,c3) * alpha
      END DO
      END DO
      END DO
!$omp end parallel do
      END IF
      END PROGRAM
```

4 Fine-tuning Optimizations

PolyOpt/Fortran offers numerous tuning possibilities, use `--polyopt-help` for a comprehensive list. We distinguish two main categories of options that impact how the program will be transformed: (1) options that control how SCoP are extracted; and (2) options that control how each individual SCoP is transformed.

4.1 SCoP Detection

The following options are available to control how SCoP extraction is being performed, and in particular how non-compliant features are handled.

- `--polyopt-safe-math-func`: This flag is not used in PolyOpt/Fortran. Fortran intrinsic functions (not intrinsic subroutines) are allowed in SCoPs as long as they do not have side effects. Currently this is determined using the function `matchAgainstIntrinsicFunctionList` in the Rose library. Also these functions cannot occur in `if` conditions, `do` loop bounds or `array` access functions.
- `--polyopt-approximate-scop-extractor`: Over-approximate non-affine array accesses to scalars (all array cells are approximated to be read/written for each array reference).
- `--polyopt-scop-extractor-verbose=1`: Verbosity option. Reports which functions have been analyzed.
- `--polyopt-scop-extractor-verbose=2`: Verbosity option. Reports which SCoPs have been detected.
- `--polyopt-scop-extractor-verbose=3`: Verbosity option. Reports which SCoPs have been detected.
- `--polyopt-scop-extractor-verbose=4`: Verbosity option. Reports which SCoPs have been detected, print their polyhedral representation, print all nodes that broke the SCoP.

4.2 Tuning Optimizations

The following options are available to control PoCC, the polyhedral engine. In particular, those control the Pluto module that is responsible for computing the loop transformation to be applied to the SCoP.

- `--polyopt-pocc-verbose`:
- `--polyopt-pluto`: Activate the Pluto module.
- `--polyopt-pluto-tile`: Activate polyhedral tiling.
- `--polyopt-pluto-parallel`: Activate coarse-grain parallelization.

- `--polyopt-pluto-prevector`: Activate fine-grain parallelization.
- `--polyopt-pluto-fuse-<maxfuse,smartfuse,nofuse>`: Control which fusion heuristic to use (default is smartfuse).

5 Known Limitations

Following are few known issues with PolyOpt/Fortran.

5.1 Using a loop iterator exit value after the loop

If the value of a loop iterator is being read outside the loop, after a construct which breakes the SCoP, then PolyOpt/Fortran may not work correctly. e.g in the following code the value of `iterator i` is being read after the subroutine call (which is a SCoP breaking construct).

```
do i = 1, 10
  a(i) = i
end do

call some_function()

do j = 1, i
  a(j) = j
end do
```

We are currently working on fixing our implementation for this case.

6 Troubleshooting

In PolyOpt/Fortran, polyhedral programs are a constrained subset of Fortran programs and it can be difficult at start to understand why a program is not detected as a SCoP. Try using the `--polyopt-scop-extractor-verbose=4` option, and reading the papers referenced below.

For any other problems, please contact directly Louis-Noel Pouchet pouchet@cse.ohio-state.edu or Mohanish Narayan narayanm@cse.ohio-state.edu.

7 References

[**1**] M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parameterized Tiling Revisited. In *International Symposium on Code Generation and Optimization (CGO'10)*, Apr 2010.

[**2**] Cédric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT'04)*, Sept 2004.

[**3**] Uday Bondhugula and Albert Hartono and J. Ramanujam and P. Sadayappan. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, Jun 2008.

[**4**] Paul Feautrier. Dataflow Analysis of Array and Scalar References. In *Intl. Journal of Parallel Programming*, 20(1):23–53, 1991.

[**5**] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II, Multidimensional time. In *Intl. Journal of Parallel Programming*, 21(5):389–420, 1992.

[**6**] Louis-Noel Pouchet, Uday Bondhugula, Cdric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan and Nicolas Vasilache. Loop Transformations: Convexity, Pruning and Optimization. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'11)*, Jan 2011.