Polly Phone call 10th April 2014

Agenda

Participants: tobias@grosser.es
sebpop@gmail.com
cib123@googlemail.com newbie, just want to listen in
Johannes
zinob@codeaurora.org
dmpots@gmail.com
stephen.neuendorffer@gmail.com

High Level Synthesis

Computation of precise types

Tobias recently tested a patch to derive minimal data types for each subexpression that in the newly generated code. This has shown to be beneficial in high-level synthesis as it reduces the LUT units used significantly.

Notes from Stephen (from an intern we had last summer)

About usability of Polly in the Vivado tool chain. In short, there are various issues to be solved, but all of them look tractable:

> Remarks on Polly dependencies analysis:
> - it generate dependences between basic blocks (and not between instructions)
> This is right. See http://llvm.org/PR12402
>
> - it is exact in the case where memory accesses are exact
> - it is an over-approximation (conservative) in the case they are not (MayWrite)
>
> To replace the actual Vivado dependencies analysis there are some requirements:
> - scalar dependences are not needed (-polly-ignore-scalar-dependences)

This option does not exist in public polly. Should probably be discussed on the mailing list.

- load/store has to be treated independently even in the same basic-block

This needs more details.

Scop detection is somewhat conservative. We need to have a way to specify a scop explicitly and analyze dependencies in the scop. If no dependencies can be analyzed, then a fallback to 'conservative' dependencies would have to happen. (This probably isn't really a problem in polly as much as a limitation to applying it in our context).

We try to allow conservative approximations. If some item is currently not recognized, but treated conservatively please report this as individual bug.

Analysis is limited to regions that are:

- not the top-level of a function

Probably just an engineering issue.

- contains at least one loop

This is an optimization for compile time.

- contains only signed arithmetic (a big limitation)

This is not too common in C code, but it should definitely be fixed.

- contains only loops in loop-simplify form, for which SCEV analysis can compute the trip count and for which this trip count is affine (depends only on parent loops induction variables and parameters in an affine way)

Loop-simplify form does not seem to be an issue, loops can just be translated in this form. The requirement of SCEV could in some cases probably be removed, but in general SCEV is needed here.

- contains only if/switch for which SCEV analysis can compute the branch condition, for which this condition is affine, for which the "condition" basic block has exactly 2 successors, and the condition is a direct comparison (not a 'or'/'and' composition of comparisons) (a big limitation)

'and' / 'or' could be added in conditions.

- contains only phi functions that refers to scalar outside the region (a big limitation)

Could probably be supported without too much problems.

- contains only functions call that do not have side effects, always return, are direct calls and are not intrinsics which access memory
- contains no alloca instruction (a big limitation)

Why is this a big issue? Allocs in loops sound surprising.

- contains only load and store for which the SCEV analysis can compute the access functions, for which this access function is affine, and that does not alias with anybody

The upcoming runtime alias checks should allow aliasing accesses.

LLVM SCEV analysis returns a getBackedgeTakenCount that may take negative values if considered signed. The correct way to interpret the getBackedgeTakenCount is to consider it unsigned and bounded by the size of its type.

http://llvm.org/PR17187

Polly ignores signed and unsigned wrap: it assumes every SCEV is signed and evolve without wrap.

Yes, we need to add runtime checks for this.

Any loop with statically known bound will have its induction variable bit-width minimized using integer wrap. The SCEV analysis will then return a getBackedgeTakenCount on the same type than the induction variable (which is annoying, but correct) and with a value usually negative computed from the exit condition of the loop (which is a comparison with a negative value, else the bit-width could still be reduced).
Polly generate a Domain for the loop which is between 0 and the getBackedgeTakenCount (included, because the first iteration do not use the backedge). As the later is negative, the Domain is empty, and no dependences are generated.

isl_pw_aff_mod allows to do a modulo by a constant and might solve the bug in a cleaner way.

A test case would be handy

Delinearization (Tobias)

First patch was committed last week. Still not enabled by default as bugs need to be fixed (See discussion on mailing list)

Sven mentioned that we should look at Armins implementation and his email on llvm-dev.

Some open issues: http://llvm.org/bugs/show_bug.cgi?id=19336

The use cases that benefit here are C99/D multi-dimensional arrays, the boost ublas library, http://julialang.org

Use of parts or ideas of Polly in core LLVM (Tobias)

At Euro LLVM there have been discussions if certain components of Polly could be beneficial for core LLVM. The question itself has been raised by several people, some had themselves positive experiences with Polly, others proposed this as a step to better align the Polly development with the LLVM community.

It should be noted that we take a conservative approach here to ensure that only high quality and generally useful concepts reach core LLVM. Some ideas, e.g. the SCEV based delinearization is directly beneficial to LLVM, so it is already today directly contributed to core LLVM. Other components, such as a ILP based dependence analysis, high-level loop optimizations or even GPU code generation are more complex and will require community discussions to evaluate their benefit.
We aim to enable such discussions by providing data about the benefits and costs involved in using Polly. As a first step we set up correctness and performance testers (http://llvm.org/perf).

Next steps:

- Enable the use of Polly analysis passes

At the moment Polly canonicalizes the IR before it is analyzed. We need to remove the need for such canonicalization to enable read-only analyzes passes.

- Remove need for -polly-indvars pass
- Remove need for -independent-blocks (requires SCEV codegen)
- Remove need for -polly-prepare
    - Model scalars/PHIs directly in Polly

- Analyze compile time overhead

We need to be more aggressive in bailing out early in case there is nothing to be gained by using Polly.

- Analyze Polly compile-time performance and ensure we do not increase compile time or run-time unnecessarily.

- Make speedups accessible without user interaction

Test cases like polybench, ublas, linear algebra in Julia, can be nicely optimized with Polly. However, at the moment this only works with additional user provided flags such as -polly-ignore-aliasing. We want to do this fully automatically

- Run-time alias check
- Guess loop bounds from static array sizes

i- Replace the use of GMP in isl with a MIT licensed integer library (imath)

This is the last missing piece to make Polly fully MIT GPL free. Qualcomm contributed irst patches to do so.

- Necessary discussions

After sufficient data is available to start an educated discussion, we should gather more community feedback on the mailing list. Some topics that already raised:

- Identification of components useful for core LLVM

Dependence analysis, loop optimization, …?

- Integer linear programming

At the moment we use isl as integer linear programming library. How to best make such functionality available in LLVM needs to be discussed.

- Testing

The question on how to use bugpoint with Polly most efficiently needs to be addressed.

Overview of current projects (+owners) (Johannes)

- Delinearization (Sebastian)
- Run time alias checks (Zino has patch, Sebastian)
- Reductions (Johannes)
- OpenMP code generator for isl-codegen/scev codegen
- Assume inbounds array accesses (Zino - no ETA)

Valid:

```
float A[100][100]
for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
                A[i][j] = 1;
```

Valid:

```
float A[100][100]
for (i = 0; i < 80; i++)
        for (j = 0; j < 103 j++)
                A[i][j] = 1;
```

A[i * 100 + j] = A[79 * 100 + 102] = A[7900 + 102] = A[8002] <- valid

```
f( int p[][100]) {
```

Qualcomm uncommitted patches:

- separation_class in isl code generation