Assignment 3c: Experimenting with the polyhedral model

Roel Jordans

2016

Organization

In the previous parts of this assignment you have seen some of the results that can be achieved with manual optimization and by guiding the compiler optimizations. This third and final part will look at the potential of optimizing through the polyhedral model. The tools that we are using for this are still in a rather experimental phase so they won't give us much better performance yet. Nevertheless, we will try to show you their strengths and current limitations.

1 Enabling polyhedral modeling

Polyhedral modeling and optimization in LLVM is performed through the Polly framework[1]. To enable Polly we normally need to add it to the LLVM project and load it into the compiler as an extension. However, we already did this for you and configured LLVM to automatically load the Polly extensions into the compiler by default.

You can pass options to Polly directly through `opt` and the `opt` binary on the server will list a whole lot of options for Polly if you pass it the `--help-hidden` option. These options are also available through the compiler driver (calling `clang`) but in that case you will need to pass them in combination with the `-mllvm` flag.

Note

Make a new checkout of the exercise 3 repository so that we can play around with a clean copy again for this part of the assignment.

[1] `http://polly.llvm.org`

Exercise 1

Looking at the `--help-hidden` output of `opt` will show you lots of options, through this you can see that there is a `-polly` option to enable polyhedral optimization in the `-O3` pipeline. Next to that, you can also find a flag called `-polly-dot`. The goal of that flag is to provide graphical information on your function about which parts could be classified.
To get this to work with our Makefile for assignment 3 pass the following compilation options to `clang`[a]

```
-mllvm -polly -mllvm -polly-dot
```

As you can see above, each Polly option needs to have the `-mllvm` prefix if you are using `clang`. Compiling the local color correction filter function with these options should now give you a `scops.contrast_enhance.dot`. Which represents[b] the SCoPs found within the `contrast_enhance` function. You will see that there are no parts in the CFG of this function which now marked green and that most of it wasn't considered for some reason[c].

[a] As part of the `CFLAGS`, remember the `+=` operator it allows you to split the line into multiple parts that you can easily disable by commenting them out.

[b] You can view the contents of a dot-file using `xdot` (you'll need to have X-forwarding enabled), or you can translate it into a pdf file using `dot -Tpdf -o`*`out.pdf in.dot`* on the commandline.

[c] The reasons are actually given at the top of each reason (the rectangular boxes surrounding parts of the CFG.

As you can see, one of the most frequently occurring excuse for not adding pieces of our function to the model is that the region isn't considered profitable. Polly already does a profitability analysis when analyzing the code and will discard smaller regions by default. *Currently* one of the main strengths of Polly is that it can do things like loop fusion. But in order for that to work you will still need to have a region that contains multiple loops. As a result, it will happily ignore any region that has a single smaller loop in it or which only has loops with a very low iteration count. To overcome this and help us in getting our code analyzable, Polly offers a flag called `-polly-process-unprofitable`. Which will tell Polly to skip the early analyzability checks and build the model anyway.

Exercise 2

Add the `-polly-process-unprofitable` flag to your CFLAGS[a] and check the SCoP detection graph to see how much more of our code got analyzed. What are the remaining parts of the code that didn't get analyzed yet?

[a] Remember that you need to prefix Polly flags with `-mllvm` when using them from `clang`.

2 Improving the code analyzability

So, looking at the graph and starting at the top we can find that there are several reasons why blocks haven't been analyzed yet.

- Call instructions; the polyhedral model doesn't like it when you leave the function that we're analyzing. Things like calling `malloc` interrupt the control flow and may have all kinds of funny side effects. Another example is the `exp` function in block `for.body62`.
- Non affine access functions; somehow the code we've ended up with is mixing 32-bit and 64-bit counters and, as a result, is seeing a whole bunch of sign-extensions (`sext`) happening.

We'll have to accept for now that the first and last block of this function will contain the memory allocation and deallocation for our buffers but we may be able to do something about the other function calls. For example block `for.end49` represents the `malloc` at line 105. Looking at the code shows us

that we can easily move that allocation upwards and put it next to the other allocations. This should give us a larger region again which we can analyze and (hopefully) optimize.

Next stop is the `exp` function call. This is currently listed as an actual function call but most architectures (especially the x86 architectures we're considering here) have a special operation that implements this function. So why didn't it get translated yet[2]? Right! We didn't put the fast math flag on yet so the compiler isn't allowed to do any reordering between the function call and the cast to double precision floating point. Adding the `-ffast-math` flag should solve this one.

Exercise 3

Great, that sounds like a few easy fixes. Move the `malloc` and add the compiler flag and check the new version of the detection graph to see which problems remain.

Great, that gives us the whole initialization stuff analyzed, but how about these sign-extensions in the main loops? These seem to really destroy the analyzability of large parts of our code...

Exercise 4

This is sadly enough a limitation of Polly. The problem here is that Polly doesn't really model integer overflow yet, and loop counters for loops with many iterations tend to be present in the interesting parts of the application. So, to avoid this, Polly currently models everything with 64-bit loop counters to avoid[a] problems with this. However, our own code is using normal `int` variables to store the sizes of the image as well as the loop induction variables. Changing these variables to use the `long` type[b] makes sure that they are also stored in 64-bit registers, which avoids the insertion of the sign-extend operations.

[a] Avoiding isn't really the best word here but it usually makes the possible issues appear much later.

[b] You might also want to update the `abs` calls in the filter to use `labs` which does the same but works on `long` values.

Great, compiling our code again and inspecting the analysis graph shows us that the analysis error changed but is still present. It is no longer complaining about something to do with a sign extend but now complains about a new problem. Yay, progress!

This time it has something to do with the access function we find in `for.cond105.preheader`, and in particular has to do with using the result of operation `%19` in an access function. This instruction can be found a bit further below in the `if.else` block in the graph. It is a select operation, which selects either the value `%add109`, or it's negated version, depending on if it was positive or negative before. In short, it calculates the absolute value of `%add109` which gets used in an index calculation[3]. Sounds familiar?

Exercise 5

To fix this we'll need to edit our code a bit. Replace the kernel lines with a version that doesn't require the `abs` computation. You can still keep the if statement within the loop though. All you need to do here is to split the else case in two[a]. First check if the stuff that's within the `abs` brackets will be positive, then execute the access without the `abs` call, otherwise execute a version which replaces the `abs` call with a simple negated version of the original expression (thus emulating the `abs` with normal instructions).

[a] You might want to do this for both the horizontal and vertical kernel.

Great, that fixed most of it. Almost the entire CFG of the filtering function should be analyzable now for you. The only bits that aren't are the top and bottom blocks which contain the allocation (`malloc`)

[2] This is where I expect you to jump up and shout something about keeping floating point computations the same.

[3] You could also have found out about this by looking at the `-Rpass-missed=polly` report from the compiler, it complains quite explicitly about this access to `Mask` being non-affine.

and de-allocation (`free`) code. Now let's see what Polly can do for us regarding transformations!

3 Code generation from the polyhedral model

To check what Polly actually found from its analysis we can add the `-Rpass-analysis=polly` option to `clang`[4]. This should show you an optimization report which explains some of the constraints that Polly found in its modeling. You will find that Polly reports that several of the arrays have possible aliasing in the code, and that it has found the constraints for the loop iterators that are required to guarantee them from not overflowing their integer ranges. Polly will automatically insert a check for these conditions when generating new code for the analyzed part, if these conditions do not hold (for example, if there is aliasing in your program) then it will simple execute the original version of the loop nest.

Exercise 6

You can inspect the structure of the generated code using the Polly debug output. To do this add the following flags to your CFLAGS:

```
CFLAGS += -mllvm -debug-only=polly-ast
```

Which will show you the AST for the code generated by Polly during compilation. From this output you can see the condition which wraps the generated code together with the loop structure generated from the polyhedral model. Do you still recognize the loops in your program?

Right, so now we can start applying optimizations. Polly can do this by itself but it currently still lacks a proper model of the memory hierarchy so it just uses some hard-coded numbers for things like vectorization width and tile sizes (which probably could be tuned much better for most applications). However, it also provides us with some command-line arguments that we can use to override the default behaviour. For this assignment we'll focus on two of the available optimizations, strip-mining and tiling. Strip-mining is used for outer loop vectorization and tiling helps us to control the cache locality of the data that our program accesses. Both could be very beneficial for our application if done properly.

Exercise 7

The first step that we'll try is the outer loop vectorization. To achieve this we'll pass Polly[a] the option `-polly-vectorizer=stripmine` and have a look at the resulting AST code. Did anything change for you? Nothing much changed for me...
The problem here is that we have a reasonably sized code fragment that we are analyzing which has quite a few memory dependencies. One of the trade-offs in code optimization is the time taken to perform the optimization versus its effects. In our case Polly decided to bail out because the transformation simply took too long. Luckily we can also disable the timeout by adding the `-polly-dependences-computeout=0` option to Polly. Doing this will significantly increase the compilation time but you should now also see that the generated AST has been transformed. New annotations have been added to show which loops are now known to be parallel, extra loop levels have been added, and the innermost loop now has a fixed iteration count to enable direct vectorization by LLVM's usual loop vectorizer (which is scheduled somewhere after the Polly optimizations). Looking at the output you can also find that Polly now decided to add a level of tiling to our loops. Can you figure out the tile and vector sizes that were used?

[a] Remember the `-mllvm` prefix for Polly options.

So, Polly managed to perform some transformations to our code. However, it only used hard-coded settings which probably don't match our target processor. Both the tile sizes and vector sizes are still

[4] Optimization reports are part of `clang` itself so you don't need the `-mllvm` prefix for this option.

standard numbers and other values would probably improve the performance further if you take the actual vector width and cache sizes into account.

Exercise 8

So, let's try to tune the vector length a bit more. It seems that Polly assumed a rather short vector length and it seems like a good idea to increase this parameter for our platform. To increase this you can use the `-polly-prevect-width=N` flag, with `N` being the selected vectorization width. Try a few values[a] and see if you can observe the effect on the generated AST.

[a] You can also provide values that are larger than the actual vector size of the system, Polly will happily generate the loop structure and the auto-vectorizer will then see how this maps on the instruction-set.

Similar to the vectorization, we can also control the tiling behaviour of Polly. We can set a default tile size, which will get applied to all loop levels[5] equally, or we can set a 2nd level tile size, which allows us to have different tiling sizes for 1st and 2nd level caches in our processor.

Exercise 9

The default tile size can be overridden in Polly using the `-polly-default-tile-size=N` option. Playing around with this option should result in differences in the frontend- and backend-stalls for our application as observed using `perf`. However, putting the tile sizes of all our loop levels to the same size is probably not the best approach.
You can also try to use the `-polly-tile-sizes=N,M,...` option, which takes a comma separated list of tile sizes to use for different loop levels. You can use the generated AST and `perf` statistics to explore the possibilities here.
Second level tiling is enabled using the `-polly-2nd-level-tiling` option, and can be controlled similarly to the normal tiling using options such as `-polly-2nd-level-tile-sizes=N,M,...`. The main difference being the extra `2nd-level` prefix to the options.

Next to the options described here Polly actually offers[6] you a whole lot more possibilities[7]. Still, these options are limited to providing fixed numbers which will then be used for all the loop nests in your code equally. Much better results can probably be obtained when applying these transformations in a more controlled manner. However, to do so will require more thorough modeling of the underlying processor architecture and memory hierarchy[8].

As you can see though, applying code transformations through the polyhedral framework is relatively easy to do compared to the manual methods. Automatic code generation for the AST helps ensuring that you only need to focus on the actual tuning of the transformations and don't need to fear that you accidentally introduce incorrectly transformed code. The current tools are, however, quite limited in the decision making process but the potential of these methods seems significant.

[5] But only if the loop range is actually larger than this tile size.
[6] You can check the help output of opt or the source code for Polly to find out which options exist and what they are supposed to do.
[7] Although not all of them are working equally well at this moment.
[8] This might make a nice graduation project for someone.