Assignment 3: Experimenting with program optimizations

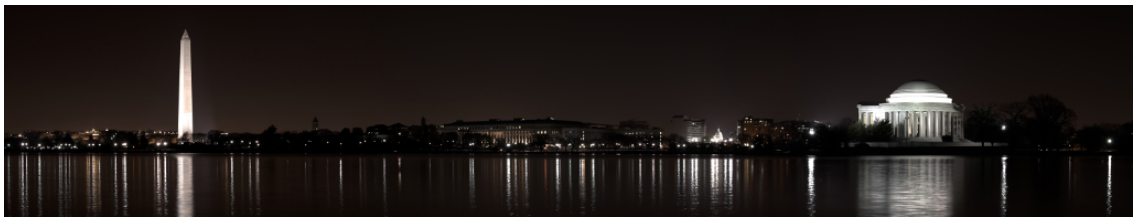
Roel Jordans, Martijn Koedam

2021

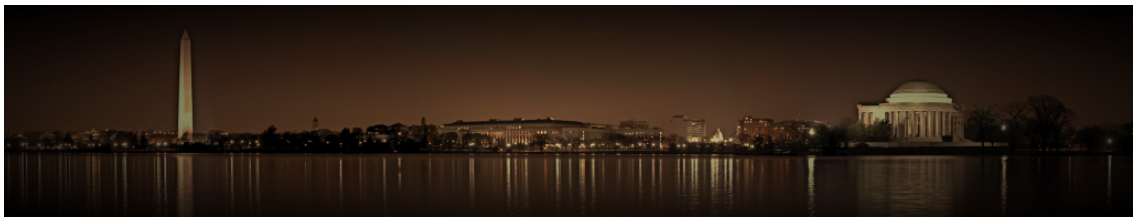
Organization

In the last two assignments we worked on the Atmel AVR microcontroller. In this assignment we are going to move up in performance and target the superscalar processor. Namely typical desktop computer processors such as the Intel Core I7. The application we target will, instead of blinking an LED, process a 6 megapixel image. In the first part of the assignment we are going to do optimizations by hand, we will turn off the compiler optimizations almost completely (optimization level -O1). In the second part, we will have a look at auto-vectorization. Finally, the third and final part analyses the potential of optimizing through the polyhedral model.

The application we are going to use is a type of application people use almost daily; they take a picture with a mobile phone, 'fix it' and then upload it to the web. So if you create a nice picture like this:



Then 'fix' (instagrammify) the image to make it look like this:



An application which does these enhancements usually exists of several filtering loops chained together. This application is not different. We start with a local color enhance algorithm [1], a blur, and a spotlight effect (darkening to the outside).

The goal of this assignment is to get a feeling about available optimizations, the impact they make, what the compiler can and cannot do, how to measure performance and what different performance indicators there are.

[1] `http://www.ipol.im/pub/art/2011/gl_lcc/`

1 Loop unrolling and program optimization, the hard way

Exercise 1

First we are going to clone the application:

```
$ git clone /home/pcp20/material/repos/assignment3.git
```

Inside the new directory you can type `make` to build the application. To run the application type the following:

```
$ ./filter in.png out.png
```

Run this application and look at the performance statistics on 3 different machines,

- `co10.ics.ele.tue.nl`,
- `co17.ics.ele.tue.nl`,
- `co24.ics.ele.tue.nl`, and
- `co26.ics.ele.tue.nl`.

These machines have different generations of CPU's. You can use `lscpu` to see what kind of CPU is available. (i7 950, i7 6700, i9 9990, and AMD Ryzen Threadripper 2920X).

Note

For the rest of this assignment run the code either on `c23.ics.ele.tue.nl` or on `co24.ics.ele.tue.nl`.

1.1 Measuring performance

As you saw in the previous exercise, the program itself already prints out some numbers for each step. Namely the time spend at that point. However time might not be the best measurement to see if the application is optimized, given between these machines there is already a factor 4 difference.

Under linux there is a nice performance profiling application `perf`[2]. This gives more detailed information about the execution of the application, below is example output of `perf stat ./filter in.png out.png`[3]:

```
 25078.767142 task-clock (msec)    #   0.997 CPUs utilized
       2,916 context-switches      #   0.116 K/sec
          16 cpu-migrations        #   0.001 K/sec
      11,057 page-faults           #   0.441 K/sec
81,356,583,982 cycles              #   3.244 GHz
48,965,949,149 stalled-cycles-frontend # 60.19% frontend cycles idle
17,251,327,876 stalled-cycles-backend # 21.20% backend cycles idle
93,871,655,420 instructions        #   1.15  insns per cycle
                                   #   0.52  stalled cycles per insn
12,298,556,810 branches            # 490.397 M/sec
   73,014,313 branch-misses        #   0.59% of all branches

 25.150977590 seconds time elapsed
```

[2] https://perf.wiki.kernel.org/index.php/Main_Page
[3] Perf talks about the CPU frontend and backend.

Exercise 2

Try to explain the different fields in the above statistics, see how they relate to each other and what field(s) is/are the most important to see if the program is optimized. Explain why the results can differ between multiple runs and propose a possible solution (take a look at `man perf stat`).

`Perf` can display more interesting information about the execution of the application, to get a list of events it can monitor for check `perf list`. To get a more detailed breakdown of time spend in your application check `man perf-record` and `man perf-report`.

Exercise 3

Use `perf record` with the right flags to determine the part of the application that takes the most time.

1.2 Optimizations

First thing we do, is create a reference output, this is a simple (but not complete) test to see if we did not change the program when applying the optimizations.

We provide a small tool `imgdiff` that can compare two images, returns 0 if there are no differences and 1 otherwise and indicate the number differences and the maximal distance[4].

Why would we want to accept differences in outputs? The following exercise demonstrates how a small change, while correct, can result in a different value.

Exercise 4

Some optimizations, even though the code code has not changed might result in different output. For example:

```
1 float v1= -1.9513026f, v2= 0.31476471f, v3= 3.1415927f;
2 float b = v1-v2+v3;
```

versus

```
1 float v1= -1.9513026f, v2= 0.31476471f, v3= 3.1415927f;
2 float b = v1+v3-v2;
```

Can you reproduce this? and if so explain why the output is different.

To optimize the application we are going to apply some of the techniques we have seen during the lecture. It is important to keep an eye on the output of the `perf stat` tool, to see the performance and where the possible bottleneck is. This whole exercise is not as trivial as you might think, because you can reduce cycles, but cause more stalls in the fontend giving you the same execution time in the end.

[4] A common way for a program to indicate that it finished successful is using the return value of the main function. Returning 0 is the standard for indicating no error.

Exercise 5

Start by investigating some of the techniques seen during the lecture to main.c filter loops and localcorrection.c. Try to find which of the following techniques can be applied to the provided code and try to predict the advantages of each of them:

1. Loop interchange
2. Loop invariant code motion
3. Common subexpression elimination
4. If conversion
5. Loop unrolling
6. Loop fusion

We are not going to optimize the png generation code. This is just required code to get the image into the memory and write it out again in a readable format. You might have seen in the Makefile we always compiled this code with *-O3*.

1.3 Vectorization

The last manual optimization we are going to do is to rewrite one of the hot loops (code that gets executed the most) using SSE, SSE2 and SSE3 vector operations. Below is the loop from *localcorrection.c* that we are going to rewrite.

```
1  float sum = 0.0f;
2  for ( t = -radius; t <= radius; t++ ) {
3      if ( nx + t >= w ) {
4          sum = sum + Mask[2 * w - 2 - nx - t + ny * w] * Kernel[t + radius];
5      }
6      else{
7          sum = sum + Mask[abs ( nx + t ) + ny * w] * Kernel[t + radius];
8      }
9  }
10 Mask2[nx + ny * w] = (float) sum;
```

Exercise 6

Rewrite the above loop using intrinsics[a] or inline assembly using SSE, SSE2 and/or SSE3. Below are some hints to get you started:

1. Move the control (if/else) out of the loop.
2. Remove abs from the 'hot' part of the code.
3. unroll the loop.

After this you should have 2 loops remaining; one to handle the corner cases and one to handle the bulk of the work. The last loop is suitable for vectorization.

[a] Intel has a nice website with an overview of the available intrinsics: `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`

2 Loop unrolling and program optimization, the easy way?

In the previous part we tried to optimize the application by hand. This is often a lot of work for very little gain. In this section we are going to start from scratch again and try to let the compiler do the brunt of the work. You will notice that this isn't a 'flip a switch and everything is good' exercise, as it also requires effort from the developer to write the code in a way that the compiler knows it can do these optimizations.

Exercise 7

We get a clean checkout of the code from assignment 3a:

```
$ git clone /home/pcp20/material/repos/assignment3.git assignment3b/
```

We first determine a baseline by running perf again:

```
$ perf stat -r 5 ./filter in.png out.png
```

Note

For the rest of this assignment run the code either on `co23.ics.ele.tue.nl` or on `co24.ics.ele.tue.nl`.

2.1 Turning on compiler optimizations

Lets turn the compiler optimizations on and see how much the compiler can speed up the execution of the program.

Exercise 8

To turn on the compiler optimizations we edit the makefile and change the following line:

```
CFLAGS:=-I. -O1 -g3 -Wall -Wextra -std=c99 -D_POSIX_C_SOURCE=200000L -fno-unroll-
    loops
```

Test both of the following variants:

```
CFLAGS:=-I. -O0 -g3 -Wall -Wextra -std=c99 -D_POSIX_C_SOURCE=200000L
```

and

```
CFLAGS:=-I. -O3 -g3 -Wall -Wextra -std=c99 -D_POSIX_C_SOURCE=200000L
```

Did the compiler managed to speed up the application significantly?

2.2 Analyzing the optimizations passes in the compiler

In the last exercise we saw that turning on the compiler optimizations resulted in almost no speedup in program execution time (less then 1 %). This is strange, we managed to do a lot better by hand? Are compilers realy this bad at optimizations or is there another reason?

Lets start by getting more information from clang about why it does or does not optimize code, we can enable this extra output by passing:

```
-Rpass=<target>
-Rpass-missed=<target>
-Rpass-analysis=<target>
```

Where <target> is the pass you want to check. For example passing -Rpass=unroll will show:

```
localcolorcorrection.c:167:17: remark: unrolled loop by a factor of 4 with run-time
    trip count [-Rpass=loop-unroll]
              for ( nx = 0; nx < w; nx++ ) {
```

This indicates that the loop has been unrolled 4 times and code has been added to handle the case where w is not a multiple of 4.

Exercise 9

To enable information about all passes that support this, change the makefile to say: [a]

```
CFLAGS:=-I. -O3 -g3 -Wall -Wextra -std=c99 -D_POSIX_C_SOURCE=200000L -Rpass=.*
     -Rpass-missed=.*
```

Try to explain the different passes it shows and what each pass does. You can add more information by passing -Rpass-analysis=<target> to the CFLAGS.[b]

[a] .* is a regex that says match anything

[b] LLVM website has a lot of useful information, for example http://llvm.org/docs/Vectorizers.html and http://llvm.org/docs/Passes.html

Now that we can see why the compiler fails to optimize the code we can change it to enable the optimizations.

2.3 Tweaking the code to help the compiler

In this part we are going to see if we can help the compiler do a better job at applying the optimizations.

Exercise 10

Try modify the code so the compiler can make more optimizations. Do this by looking at the output of -Rpass-missed=<target> and -Rpass-analysis=<target> and applying some of the initial transformations you did in the previous exercise, like removing if statements from the innerloop.

While trying to make the compiler optimize the code we often see remarks like this:

```
remark: loop not vectorized: vectorization is not beneficial and is not explicitly
    forced
```

The compiler has an internal cost model to see if the overhead of vectorization is not larger then the gain. In the provided code it thinks the loop we rewrote in the last exercise of assignment 3a using 'intrinsic' is not suitable for vectorization. We however saw it did help obtain a speedup. We are going to tell the compiler to optimize the loop anyway.

Exercise 11

See what happens if your force a loop to be parallelized that the compiler does not find beneficial, you can add:

```
#pragma clang loop vectorize_width(4)
```

Check to see the result of this. Is the compiler always correct in thinking it is not beneficial?

Note

Use `objdump` to verify that the compiler is using vector instructions.

After this you should see a reduction of at least 33% in number of cycles and instructions. Can we help the compiler further?

2.4 Giving more hints to the compiler

One thing that is holding the compiler back is that it has no idea about the inputs of the program. Say this program is used in a mobile phone and always has the same input size, You could possibly vectorize more of the code. You can tell clang this by passing it assumptions:

```
__builtin_assume(n%4 == 0)
```

This tells the compiler that n is always a multiple of 4.

Exercise 12

Place this assume at the right places and check if the compiler uses this to improve the execution time. Use the `-Rpass-analysis=vectorize` to see if it changed the vectorization.

3 Experimenting with the polyhedral model

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.

Polyhedral modeling and optimization in LLVM is performed through the Polly framework[5]. To enable Polly we need to add it to the LLVM project which means downloading its source into the `tools/polly` subdirectory of the LLVM repository. LLVM will then detect this during configuration and automatically load the Polly extensions into the compiler.

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.

Exercise 13

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.

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

Exercise 14

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`.

3.1 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[6]? 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 15

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 16

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.

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

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[7]. Sounds familiar?

Exercise 17

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

3.2 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`[9]. 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 18

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?

[7] 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.

[8] Or at least, it used to in LLVM 3.9 which was available when we started with this assignment. The current LLVM 5.0.1 has a regression though and still fails to analyze the code. As a work-around you can pass `-mllvm -simplifycfg-sink-common=false` to Clang to disable the optimization that is suspected of messing with our code structure. Doing so indeed helps and gets Polly to properly analyze things again. Quite annoying but something to live with for this assignment.

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

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 19

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

Exercise 20

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[10] 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 21

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.

[10] But only if the loop range is actually larger than this tile size.

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[11] you a whole lot more possibilities[12]. Still, most of 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[13]. Some work on this was started recently by Roman Gareev[14] as part of a Google Summer of Code project for adding pattern based tiling optimizations into Polly. The final report of his project contains some info on how to use these transformations.

Exercise 22

Based on the tiling optimizations presented in a Transactions on Optimization of Mathematical Software[a] these added optimization target a specific algorithm structure (matrix multiplication) and apply tiling optimizations accordingly in the Polyhedral model.

These additions to Polly are still experimental but can be enabled using the Polly flag `-polly-pattern-matching-based-opts`. The optimizations can then be controlled by specifying the processor memory hierarchy and execution model in more detail so that the right trade-off can be selected. The following options can be found in `opt`'s help output that control these transformations. Try to see if these can help you optimize your code further.

```
-polly-target-1st-cache-level-associativity=<int>
-polly-target-1st-cache-level-size=<int>
-polly-target-2nd-cache-level-associativity=<int>
-polly-target-2nd-cache-level-size=<int>
-polly-target-first-level-cache-line-size=<int>
-polly-target-latency-vector-fma=<int>
-polly-target-throughput-vector-fma=<int>
-polly-target-vector-register-bitwidth=<int>
```

[a] *Analytical Modeling is Enough for High Performance BLIS*, available at `http://www.cs.utexas.edu/users/flame/pubs/TOMS-BLIS-Analytical.pdf`

As you can see, 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.

[11] 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.

[12] Although not all of them are working equally well at this moment as the project is heavily under development.

[13] This might make a nice graduation project for someone.

[14] `http://romangareev.blogspot.nl/`