Assignment 3b: Loop unrolling and program optimization, the easy way?

Martijn Koedam

2016

Organization

In Assignment 3a we tried to optimize the application by hand. This is often a lot of work for very little gain. In this assignment we are going to start from scratch again and we are going to 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, but it also requires effort from the developer to write the code in a way that the compiler knows it can do these optimizations.

The goal of this assignment is to learn how to fully exploit the optimization capabilities of the compiler. So that in the future when you write code, you write it in a way that gets you the best performance for the least amount of effort.

1 Obtaining and running the application

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

Exercise 1

First we are going to clone the application:

```
$ git clone /home/pcp16/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 `co9.ics.ele.tue.nl` or on `co10.ics.ele.tue.nl`.

1.1 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 2

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
```

to

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

Did the compiler managed to speed up the application significantly?

1.2 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 3

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.

1.3 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 4

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 5

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?

1.4 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 6

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.

2 Polly

The exercise about polly[1] will follow later.

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