Modern Computer Architecture
Lab Report

Mick van Gelderen   Arian Stolwijk

4091566   4001079

November 2013

1   The kernel function

The concept of this lab is to take a piece of code that is executed frequently and run it on a reconfigurable co-processor. The extracted piece of code is called the kernel and the device is in our case the ρ-VEX: A reconfigurable and extensible VLIW processor.

The kernel needs to be compiled for the ρ-VEX so that we can inject the result, the bytecode, in to the co-processor. The rest of the code runs as usual on a regular processor.

2   Communication

We have access to several (but not enough during the lab) FPGAs which are configured as Microblaze processors and run Linux.

The ρ-VEX is a configured as a co-processor that can be controlled using a number of memory mapped files. The is a file for writing the instruction memory, one for reading and writing the data memory, one for reading the status and one for writing control commands.

We abstracted this away to a small interface with the following functionality:

rvexInit
> Attempts to open the files.

rvexDispose
> Closes all files opened by rvexInit.

rvexBytecode
> Allows you to write bytecode to the instruction memory.

rvexWrite
> Allows you to write to the data memory.

rvexRead

    Allows you to read from the data memory.

rvexSeek

    Allows you to jump to a given position in the data memory.

rvexGo

    Writes the clear and start commands to the control memory and blocks until the status reports that the operation was successful.

3 Profiling results

We compiled the unmodified x264 application with profiling support on a VM that allows us to compile microblaze and ρ-VEX applications. Unfortunately we cannot use gprof on the Microblaze machine itself so we ran gprof on the VM itself.

Using the input video $\sim$/Videos/inputs/eledream_640x360_8.y4m we obtained the following profiling results.

```
gprof x264 | head −n 10
Flat profile:

Each sample counts as 0.01 seconds.
  %    cumulative    self              self     total
 time    seconds    seconds    calls   ms/call   ms/call   name
13.70      0.10       0.10   1599044    0.00      0.00    x264_pixel_satd_8x4
13.70      0.20       0.10    570708    0.00      0.00    get_ref
 6.85      0.25       0.05     38770    0.00      0.00    x264_pixel_sad_x4_16x16
 5.48      0.29       0.04    460484    0.00      0.00    quant_4x4
 4.11      0.32       0.03    123076    0.00      0.00    sa8d_8x8
```

We see that the function x264_pixel_satd_8x4 is called 1.6 million times during the video conversion. These calls together take up about 14% of the total time. An equal amount of time is spend in x264_pixel_satd_8x4.

The video eledream_640x360_128.y4m makes the application spend about 18% of the runtime in the pixel processing function and 16% in get_ref.

The bigger our input video the more time we will spend processing and the more effect an optimization will have if it targets part of the processing code.

Judging from the profiling information there are two potential places where optimization will be effective: x264_pixel_satd_8x4 and get_ref.

When we looked at the source code of x264 we thought that the nature of x264_pixel_satd_8x4 was more suitable for optimization because it had some loops in it. The get_ref function was a lot more irregular and also harder to understand. On the flip side, the overhead caused by communication between the microblaze processor and the ρ-VEX would be smaller for get_ref because it is called three times less than x264_pixel_satd_8x4.

In the end we chose to try and put the computation of x264_pixel_satd_8x4 on the co-processor and let get_ref for what it was. The function x264_pixel_satd_8x4

was easier to understand and making it work was more important to us than choosing the best optimization area right away.

4 Endianess

Since we had to compile for the Microblaze using the flag -DWORD-BIGENDIAN we figured that the Microblaze would be a big endian machine. You can always test it by writing a multibyte value like 0xDEADBEEF to memory and read the individual bytes. If you read 0xDE 0xAD 0xBE 0xEF you will know that you have a big endian machine. If you get that sequence but in reverse you know its a little endian machine.

5 Results of using the ρ-VEX

We modified the x264 application to log its processing time computed with clock_gettime. This required linking the rt library.

Then we tried to find an FPGA that was not being used by anyone else and we ran several versions of the x264 application. All of them included the code from Listing 1 which allowed us to get an idea of the total runtime.

Listing 1: Capturing runtime with the monotonic clock

```
struct timespec tss, tse, tsd; // start, end and diff
clock_gettime(CLOCK_MONOTONIC, &tss);

if( !ret )
    ret = encode( &param, &opt );

clock_gettime(CLOCK_MONOTONIC, &tse);
if (tse.tv_nsec > tss.tv_nsec) {
    tsd.tv_sec = tse.tv_sec - tss.tv_sec;
    tsd.tv_nsec = tse.tv_nsec - tss.tv_nsec;
} else {
    tsd.tv_sec = tse.tv_sec - tss.tv_sec - 1;
    tsd.tv_nsec = tse.tv_nsec - tss.tv_nsec + 1000000000;
}
printf("Took %lu.%09lu sec\n", tsd.tv_sec, tsd.tv_nsec);
```

The versions that we created were the following

vanilla
> The original x264 implementation.

rvex simple
> Simple implementation where memory mapped files are opened once for stability during execution and only one ρ-VEX is supported at a time.

rvex inline
> Based on simple, uses an object to store the file descriptors so allows multiple ρ-VEX processors to be used and the time critical functions are placed in a header file with the inline annotation.

We ran the script from Listing 2 to get an idea of the runtimes of the different versions. Note that we already did manual test runs to get an idea of how long the runs would take. We noticed that running the same test multiple times produced very similar run times. We did not find it necessary to do multiple runs for the purposes of this lab because the error would be very small in comparison with the run times for different versions of the application:

```
Took  75.280923348  sec
Took  75.350194581  sec
Took  75.221714038  sec
Took  75.374499116  sec
```

Listing 2: Test script

```
echo 'eledream 64x36 3 frames with timing' >> group11.log
./x264-timing-sb2 eledream_64x36_3.y4m -o out.mkv | grep Took >> group11.log

echo 'eledream 640x360 8 frames with timing' >> group11.log
./x264-timing-sb2 eledream_640x360_8.y4m -o out.mkv | grep Took >> group11.log

echo 'eledream 64x36 3 frames with rvex interface simple' >> group11.log
./x264-rvex-sb2 eledream_64x36_3.y4m -o out.mkv | grep Took >> group11.log

echo 'eledream 64x36 3 frames with rvex interface struct and inline' >> group11.log
./x264-rvex-struct-inline-sb2 eledream_64x36_3.y4m -o out.mkv | grep Took >> group11.
```

Listing 3: Test output

```
/ # cat group11.log
eledream 64x36 3 frames with timing
Took 75.699494266 sec
eledream 640x360 8 frames with timing
Took 317.733194410 sec
eledream 64x36 3 frames with rvex interface simple
Took 442.169817935 sec
eledream 64x36 3 frames with rvex interface struct and inline
Took 440.886382661 sec
```

Summarizing the manual and script results we get the following table:

	64x36 3f	640x360 8f
vanilla	75.7s	317s
rvex simple	442s	>20m
rvex inline	441s	untested

Most importantly, we see that our optimization actually made the application run several times slower. Also, we can conclude that the gcc compiler does

a teriffic job at inlining functions when -O3 is enabled, even if you do not ask it to. We couldnt wrap our heads around doing so much communication related tasks in a function that was basicly a relatively small fixed number of basic operations (*, +, -, shifts, comparisons) and achieving a faster result by that. We assumed that the ρ-VEX was extremely tightly integrated into the microblaze which allowed insane communication speeds but the test results show that this is not the case.

The speedup that we obtained can be calculated by looking at the time spent in the kernel function. This should be about 14% for the input video eledream_640x360_8.y4m judging from the profiling results obtained on the VM. We cannot say exactly how much time is spent there because we cannot profile on the Microblaze.

But because the ρ-VEX simple version took longer than 20 minutes which caused the ssh session to hang we cannot use the exact run time. Let us see what the speedup would be if the run time was 20 minutes exactly.

The vanilla version spends 14% of the 75 seconds that it runs in `x264_pixel_satd_8x4`. So 86% is spent in the rest of the functions. Since we have not changed anything beside the kernel we can assume that the time spent in that part will be equal for vanilla and for ρ-VEX simple. This means that $\frac{1200 - .86*75}{75 - .86*75} = 108$ is the relative speed of the vanilla function in comparison with the ρ-VEX version. It actually runs 108 times faster. In other words, the ρ-VEX version runs 108 times slower which amounts to a speedup of -10700% obtained by introducing ρ-VEX.

6   Additional assignment

- What were the results of the additional assignment? How did it affect the speed of the application?

Figure 1: The Universe

7   Conclusion

"I always thought something was fundamentally wrong with the universe" [1]

References

[1] D. Adams. *The Hitchhiker's Guide to the Galaxy.* San Val, 1995.