Modern Computer Architecture
Lab Report

Mick van Gelderen
4091566

Arian Stolwijk
4001079

November 2013

Abstract

For the practical assignment of the Modern Computer Architecture course we've changed the x264 program. We've profiled x264 and extracted the `x264_pixel_satd_8x4` as kernel function. To get a performance improvement we've moved the execution of the kernel function to the ρ-VEX, a co-processor of the MicroBlaze. A requirement for this is to load the program into the instruction memory and write and read the data to and from the data memory. We've found that our implementation did not generate a performance increase, but rather a decrease. We think this is due to the communication overhead.

1 Introduction

The assignment for this report was basically, find a kernel function from an application and run it on the ρ-VEX. To do this there are a few things we need to know first:

Platform The platform the program will run on is the ERA platform. It consists out of the host processor, the MicroBlaze, accelerated with a VLIW co-processor, the ρ-VEX. For the co-processor computationally intensive kernels can be extracted to achieve a performance increase.

Kernel function The extracted piece of code is called the kernel. The kernel needs to be compiled for the ρ-VEX so that we can inject the result, the byte-code, in to the co-processor. The rest of the code runs as usual on a regular processor.

x264 The x264 program is the application we will try to improve using the ρ-VEX. x264 is a software library for encoding video streams in the H.264 compression format.

The goal for this report is to extract the correct kernel, using profiling and compile this for the ρ-VEX. To execute the kernel on the ρ-VEX there should be communication between the processor and the co-processor.

2 Profiling

To know what we are going to optimize, we need to profile the application first. Just taking a random function isn't a good idea. Fortunately we can compile the x264 program with the gprof profiling flags enabled. Compiling the program and running it on the Virtual Machine with the ~/Videos/inputs/eledream_640x360_8.y4m as input video we get 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 spent 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 and arithmetic 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.

3 Data Layout

4 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. You can also specify the bytecode which will be written to the instruction memory.

rvexDispose
: Closes all files opened by rvexInit.

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.

5 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.

6 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 a 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
```

Table 1 summarizes the results found by manual testing and the listed script.

Table 1: Execution times

	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 terrific 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-VEXwas extremely tightly integrated into the MicroBlaze which allowed very good 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.

Because the ρ-VEXsimple 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 rvex simple. This means that $\frac{1200-0.86 \cdot 75}{75-0.86 \cdot 75} = 108$ is the relative speed of the vanilla function in comparison with the rvex version. It actually runs 108 times faster. In other words, the rvex version runs 108 times slower which amounts to a speedup of -10700% obtained by introducing rvex.

7 Additional assignment

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

Figure 1: The Universe

8 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.