Report Lab Assignment
Running x264 on Host Processor
Extracting and Accelerating Kernel on Co-processor

Imara Speek (1506374)
Aimee Ferouge (4014030)

10 november 2013

Date Performed:	October, 2013
Instructor:	Ir. A. Brandon

Samenvatting

10 lines of Abstract text

1 Wat moet er nog gebeuren

- debuggen van de rovex uitleggen met nep pixels en bewijs van werking
- Stride en manier van het schrijven naar het geheugen uitleggen
- Gebruik van lseek uitleggen
- uitleg van de kernel functie met plaatjes
- communicatie van pvex en microblaze
- big or little endiannes, en aanpak in code
- speedup & theoretical calculation speedup
- results of additional assignment and if more time which we would have chosen

2 Introduction

For the lab a computational intensive kernel has to be extracted from a `x264` software application, free software library for encoding video stream into H.236/MPEG-4 AVC format. The application is executed on a FPGA running a MicroBlaze host processor. By running the particular kernel on ρ-VEX co-processor, the execution time can be decreased by making use of hardware acceleration.

2.1 Getting Used to the Environment

The first lab is meant to get used to the project environment. A few input files are given to show the compile and run commands of the x264 application, by inserting a `.y4m` stream and creating a short `.mkv` movie. By adding the `gprof` flag to the compile command, a list is created of all functions ordered by their share of the total execution time (in percentage).

2.2 Detecting the Computationally Most Intensive Kernel

Profiling the x264 execution for the `.y4m` files that are provided by the lab leads to the ranking show in table **??**.

3 Approach

Our extracted kernel can be found in appendix **??**. In order to make the extracted kernel qualified for compilation and execution, a few things have to be altered. First, the new file has to be recognized by the `makefile` in the rovex-examples directory. By executing `make byte−` command two files are created:

- `bytecode`, containing all the intstructions to be executed by the ρ-VEX
- `bytedata`, containing the pixels of the input stream

Second, some type definitions have to be made. Originally, `pixel_satd_8x4` resides in the `pixel.c` file of the x264 application. When extracting this kernel, all prior knowledge is lost and has to be defined again. Then, in order to make the kernel compile and run on the ρ-VEX, the development board has to be reset and started. `Bytecode` has to be written to the instruction memory (`rex−imemory`) and `bytedata` to the data memory (`rvex−dmemory`). Finally, the calculated result should be returned to the host. Figure X shows the ρ-VEX memory layout for our kernel.

3.1 Adjusting the `makefile`

In order to make the makefile recognize the extracted kernel, the file can simply be added to the `EXECUTABLES`. Other files are already in the `makefile` (e.g. `adpcm`), but these can be removed since we will not need them for our application. An important issue is the difference between logical memory and physical memory. While the first address of a logical memory is obviously zero, the physical memory can have the first part of the register being occupied by the operating system. For the ρ-VEX, the first address which is allowed to be (over)written is 120. Thus, when writing for example the result of the kernel to the logical address '0', it actually should be written to the physical address '120' of the data memory. This can be done using `__DATA_START` to indicate the start address.
Despite the fact that this is a rather simplistic operation, it took us some struggling to have this action confirmed as correct. For example, we were told to remove the `AUTOINLINE` flag which resulted in a lot of wrong hexdumps. Also, half way the lab a fix has been made to eliminate the issue of the logical and physical addresses. All references to `__DATA_START` had to be removed again, which felt like we had wasted lots of time. Unfortunately, we still had major problems getting the application to run

```c
// Open the imem and fill it with bytecode
// Write the bytecode into the instruction memor
int instr = open("/dev/rvex-imemory.0", O_WRONLY
if (instr == -1){
    printf("\ninstr is already open or error has
    return -1;
}
write(instr, codebuffer, 2000);
close(instr);
printf("\nafter writing bytecode");
```

(a) Bytecode

```c
// Open the data register and check whether it has oper
// point data mem to 120 on the rovex, address 0 on the
// write pixel data to memory written in a sequential m
// the data memory and start at 0
int data = open("/dev/rvex-dmemory.0", O_RDWR);
if (data == -1){
    printf("\ndata is open or error has occured\n");
    return -1;
}
if (lseek(data, 0, SEEK_SET) == -1)
{
    printf("\nset data location to start data memory fa
    return 0xff;
}
for (i = 0; i < 4; i++, pix1 += i_pix1, pix2 += i_pix2)
{
    printf("\nenter pix loop");
    write(data, pix1, stride);
    write(data, pix2, stride);
}
```

(b) Bytedata

```c
/* ---------------------reset and run the code-------------------- */
char temp;
// Write the reset command to rvex
temp = '2';
write(ctrl, &temp, 1);

// Write the start command to rvex
temp = '1';
write(ctrl, &temp, 1);

// Clean temp
temp = 0;
printf("\nafter running code o the rovex");

// Wait for the rvex to finish by reading the status
do
{
    read(status, &temp, 1);
    printf("\nstatus loop");
} while (temp != '3');

printf("\nrovex is finished");
```

(c) Reset, set and get the status of the ρ-VEX

Figuur 1: Commands for reading from and writing to the ρ-VEX

correctly. As it turned out, the fix had only affected the 'home' directory. In order to avoid conflicting files, we had created a separate folder named 'lab2' from where we executed the application. Due to this, the fix did not reach our code and thus did not eliminate the start address issue.

3.2 Type Definition

As stated earlier, an extracted kernel is unable to obtain information from previous code. Consequently, all parameters have to be pre-defined. Table **??** shows all required type definitions.

3.3 Constraints of the ρ-VEX

*** Beperkingen van de RoVEX uitleggen ***

3.4 Communication between the MicroBlaze and the ρ-VEX

In order to delegate the `satd_8x4` kernel from the MicroBlaze to the ρ-VEX, both the environments need to communicate with each other. When executing the `x264` application, the MicroBlaze has to load the instructions of the extracted kernel into the instruction memory of the ρ-VEX and the data (for which the SATD has to be evaluated) into the data memory. This is when the source code of `x264` becomes involved. The `pixel_satd_8x4 kernel`, which is residing in the `pixel.c` file of the application, needs to be overwritten. Instead of calculating the SATD, it should send the instructions and input to the ρ-VEX. The commands **??** and **??** are therefore added to the `pixel.c` file of the `x264` application. These pixels are written at physical address 120, the first address on the ρ-VEX that is not set apart for communication between the driver and the ρ-VEX. Also, an empty pixel is written after `bytedata` in order to make space for writing the result.
To control the ρ-VEX from the x264 application, the control memory should also be written. By first setting the control to `'2'`, the ρ-VEX is being reset. It can then be

started by writing `'1'` to control, telling the ρ-VEX to start calculating the satd of the two pixels. While calculating, the status of the ρ-VEX can be checked in a `while`-loop by reading the status variable of the status memory (`rvex−smemory`). When the `satd` kernel is finished, the result can be read from the data memory. See also **??**.

3.5 Result Hyphothesis

By having the computational intensive kernel being run concurrently on a co-processor, one could expect an overal speedup of the application. A problem is, however, that the `bytecode` and `bytedata` have to be sent to the ρ-VEX every time the kernel is being called. `Bytedata` contains two new pixels of which the SATD has to be calculated, `bytecode` the kernel instructions. Because of this constant data traffic between the MicroBlaze and the ρ-VEX we don't think the intended speed up is achieved.
The reason that `bytecode` has to be sent every time the kernel is called, is that the three FPGAs are shared among a lot of students. When running their application concurrently, the instruction memory is constantly overwritten by another group. If there was a one-to-one setup, `bytecode` could have to been placed in `main.c`, written tot the `imemory` of the ρ-VEX once when starting the application.

4 Implementation

The approach described in the previous section wasn't implemented in a day, obviously. This section will cover some difficulties that came along during the lab.

4.1 The Fix

Halfway the lab, a fix was introduced to take care of the stride issue. The stride values were now set and did not have to be passed on using `__DATA_START`. In order to make the ρ-VEX read the pixels in the data memory correct a pointer to the first address would be enough.
Unfortunately, this caused us problems for weeks. Since we wanted to let the original x264 file unaltered, we copied the folder and created a new application. The folder was called 'lab2', with the `pixel.c` file called `microlab2.c`. The `lab2.c` file containing the extracted kernel was in the rovex-examples folder because of the `makefile`, that was also residing there. Now, the fix was only applicable to the original x264 folder, so we still had to deal with the stride issue.
When we found out about this fix, we adjusted the original x264 folder. We commented out the source code of `pixel_satd_8x4` and put our MicroBlaze kernel code instead. To check whether the pixels were sent to ρ-VEX data memory correctly, we pre-defined two pixels in `pixel.c` to be written to `rvex−dmemory`. This way, we could have certain expectations when doing a `hexdump` to examine the registers. Our testpixels are shows in figure **??**.

4.2 Order of Variable Initialization

After moving our code to the original x264 code and creating test pixels to be used for SATD calculation, a dump of the ρ-VEX data memory can been seen in figure **??**. As you can see, the first two byte are not matching. At first, we thought this

```c
pixel imagepix[pixsize] = { 50, 47, 45, 42, 40, 37, 34, 32,
                49, 46, 44, 41, 39, 36, 33, 31,
                48, 45, 43, 40, 37, 35, 32, 30,
                47, 44, 42, 39, 36, 34, 31, 29  };

pixel imagepix2[pixsize] = {    48, 50, 47, 33, 27, 26, 30, 29,
                48, 50, 47, 33, 27, 26, 30, 29,
                48, 50, 47, 33, 27, 26, 30, 29,
                48, 50, 47, 33, 27, 26, 30, 29  };
```

(a) Test pixels defined in `pixel.c`

```
00000000  00 b0 2d 2a 28 25 22 20  31 2e 2c 29 27 24 21 1f  |..-*(%" 1.,)'$!.|
00000010  30 2d 2b 28 25 23 20 1e  2f 2c 2a 27 24 22 1f 1d  |0-+(%# ./,*'$"..|
00000020  30 32 2f 21 1b 1a 1e 1d  30 32 2f 21 1b 1a 1e 1d  |02/!....02/!....|
*
00000040  80 80 01 43 80 80 80 80  80 80 80 80 80 80 80 80  |...C............|
00000050  80 80 80 80 80 80 80 80  80 80 80 80 80 80 80 80  |................|
*
```

(b) Hexdump of the testpixels (first two bytes incorrect)

Figuur 2: Commands for reading from and writing to the ρ-VEX

could be because of the fact that another group was running their application at the same FPGA concurrently. However, the value of these bytes remained the same during several runs.
This unwanted write was caused by the initialization of the `sum` variable, that has type `short int` and was stored in the data memory before the pixel. When `pixel_satd_8x4` tried to find the pixels, it found `sum` instead of the pixels, causing the program to get stuck in a loop. When changes `sum` to not being initiazlied, `datamem` containing the pixels had the first spot at address 120.

4.3 Endianness

*** byteswap toelichten ***

A Extracted kernels `pixel_satd_8x4` and `pixel_satd_4x4`

```c
// Defining types for the expected input arguments
// changed intptr_t to int instead of unsigned int and sum2 from us long to us int
typedef unsigned int intptr_t;
typedef unsigned char pixel;
typedef unsigned short sum_t;
typedef unsigned long sum2_t;

// Define an inputarray that points to the start of the data memory
pixel datamem[128];

// BIT_DEPTH is 8 always, so if else statement is removed
#define BIT_DEPTH 8

// #define BITS_PER_SUM (8 * sizeof(sum_t))
#define BITS_PER_SUM (8 * sizeof(sum_t))

// HADAMARD4 function exported from the original pixel.c file
#define HADAMARD4(d0, d1, d2, d3, s0, s1, s2, s3) {\
    sum2_t t0 = s0 + s1;\
```

```
    sum2_t t1 = s0 - s1;\
    sum2_t t2 = s2 + s3;\
    sum2_t t3 = s2 - s3;\
    d0 = t0 + t2;\
    d2 = t0 - t2;\
    d1 = t1 + t3;\
    d3 = t1 - t3;\
}

// abs function exported from the original pixel.c file
static sum2_t abs2(sum2_t a);

// x264_pixel_satd_8x4 exported from the original pixel.c file
int x264_pixel_satd_8x4();

// x264_pixel_satd_4x4 exported from the original pixel.c file
int x264_pixel_satd_4x4();

char main()
{
    // run the x264_pixel_satd_8x4 by passing on the pixel values.
    int result, i;
    result = 0;
    i = 0;

    // calculate the result
    if (datamem[0x40] == 0x44)
    {
        result = x264_pixel_satd_4x4();
    }
    else if (datamem[0x40] == 0x84)
    {
        result = x264_pixel_satd_8x4();
    }
    else result = 0xdead;


    // clean result array
    for (i = 0; i < 4; i++)
    {
        datamem[i] = 0x00;
    }

    // write result in reverse endianess
    datamem[0x00] = (result >> 24) & 0xFF;
    datamem[0x01] = (result >> 16) & 0xFF;
    datamem[0x02] = (result >> 8) & 0xFF;
    datamem[0x03] = result & 0xFF;

    // DEBUG
    // datamem[0x40] = 0xfe;
    // datamem[0x41] = 0xed;

    return 0;
}

static sum2_t abs2(sum2_t a)
```

```
{
    sum2_t s = ((a>>(BITS_PER_SUM-1))&(((sum2_t)1<<BITS_PER_SUM)+1))*((sum_t)-1);
    return (a+s)^s;
}

int x264_pixel_satd_8x4()
{
        sum2_t tmp[4][4];
        sum2_t a0, a1, a2, a3;
    sum2_t sum = 0;
    int i = 0;

    // Strides are set values and do not have to be given as input arguments
    // Stride is 16 to compute 2 rows of 8 bytes of pixel array elements
    intptr_t stride = 16;

    // Pointer to data memory
    pixel* datapoint = datamem;

    // Adjust the for loop so it can fetch data from data memory
    for(i = 0; i < 4; i++, datapoint += stride)
    {
        a0 = (datapoint[0] - datapoint[8]) + ((sum2_t)(datapoint[4] - datapoint[12]) << BITS_PER_SUM
        a1 = (datapoint[1] - datapoint[9]) + ((sum2_t)(datapoint[5] - datapoint[13]) << BITS_PER_SUM
        a2 = (datapoint[2] - datapoint[10]) + ((sum2_t)(datapoint[6] - datapoint[14]) << BITS_PER_SU
        a3 = (datapoint[3] - datapoint[11]) + ((sum2_t)(datapoint[7] - datapoint[15]) << BITS_PER_SU
        HADAMARD4( tmp[i][0], tmp[i][1], tmp[i][2], tmp[i][3], a0,a1,a2,a3 );
    }
    for(i = 0; i < 4; i++ )
    {
        HADAMARD4( a0, a1, a2, a3, tmp[0][i], tmp[1][i], tmp[2][i], tmp[3][i] );
        sum += abs2(a0) + abs2(a1) + abs2(a2) + abs2(a3);
    }
    return (((sum_t)sum) + (sum>>BITS_PER_SUM)) >> 1;;
}

int x264_pixel_satd_4x4()
{
    sum2_t tmp[4][2];
    sum2_t a0, a1, a2, a3, b0, b1;
    sum2_t sum = 0;
    int i = 0;

        // Strides are set values and do not have to be given as input arguments
    // Stride is 8 to compute 2 rows of 4 bytes of pixel array elements
    intptr_t stride = 8;

    // Pointer to data memory
    pixel* datapoint = datamem;

    // instead of 8x4 method, it writes per 4 bytes
    for( i = 0; i < 4; i++, datapoint +=stride )
    {
        a0 = datapoint[0] - datapoint[4];
        a1 = datapoint[1] - datapoint[5];
        b0 = (a0+a1) + ((a0-a1)<<BITS_PER_SUM);
        a2 = datapoint[2] - datapoint[6];
```

```
        a3 = datapoint[3] - datapoint[7];
        b1 = (a2+a3) + ((a2-a3)<<BITS_PER_SUM);
        tmp[i][0] = b0 + b1;
        tmp[i][1] = b0 - b1;
    }
    for( i = 0; i < 2; i++ )
    {
        HADAMARD4( a0, a1, a2, a3, tmp[0][i], tmp[1][i], tmp[2][i], tmp[3][i] );
        a0 = abs2(a0) + abs2(a1) + abs2(a2) + abs2(a3);
        sum += ((sum_t)a0) + (a0>>BITS_PER_SUM);
    }
    return sum >> 1;
}
```

Input file Kernel name	Time (sec)	Share (%)	Calls
stream_32x18_1.y4m	0.02		
x264_analyse_init_costs		100.00	71
x264_free		0.00	1646
x264_cabac_encode_desicion_c		0.00	784
stream_64x32_3.y4m	0.03		
x264_analyse_init_costs		66.67	71
x264_pixel_satd_4x4		33.33	1971
x264_pixel_satd_8x4		0.00	4830
stream_640x320_8.y4m	1.61		
x264_pixel_satd_8x4		14.29	1599044
x264_get_ref		11.80	570708
x264_pixel_satd_x4_16x16		4.97	38770
stream_640x320_32.y4m	8.54		
get_ref		20.61	7228633
x264_pixel_satd_8x4		13.23	15386501
x264_pixel_satd_x4_8x8		4.57	1009690
stream_640x320_128.y4m	29.86		
get_ref		17.48	21956292
x264_pixel_satd_8x4		14.17	49862831
x264_pixel_satd_x4_16x16		6.56	1023315

Tabel 1: Chart with computationally most intensive kernels for each input stream.

Parameter	Type definition	Motivation
`intptr_t`	`unsigned int`	This parameter represents the stride, which cannot be <0
`pixel`	`unsigned char`	Pixels are made out of bytes, which have 8 bits (like a `char`)
`sum_t`	`short int`	Same type definition as in the source code (16 bits)
`sum2_t`	`long int`	Same type definition as in the source code (32 bits)
`BIT_DEPTH`	defined as 8	Also in the source code, plus the kernel handles pixels (bytes)
`BIT_PER_SUM`	(8 * `sizeof(sum_t)`)	Also predefined as in the source code, being 8 as well

Tabel 2: Type definitions in the extracted kernel file.