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

Imara Speek (1506374)
Aimee Ferouge (4014030)

11 november 2013

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

Samenvatting

During this lab, we are speeding up the `x264` application by extracting the `pixel_satd_8x4` and the `pixel_satd_4x4` kernel for parallel processing on a ρ-VEX co-processor. This involves running the `x264` application on a FPGA which runs a MicroBlaze operating system. The required instructions and data for the extracted kernels is being written to the instruction- and data memory of the ρ-VEX. This lab report describes in detail our chosen approach, problems we encountered along the way, test results measuring the execution time before and after extracting the kernels and an overal evaluation.

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 the `x264` software application: a free software library for encoding video stream into H.236/MPEG-4 AVC format. It is able to use Periodic Intra Refresh instead of keyframes as used by h.264, refreshing the image of the video by moving a column of intra blocks from one side of the screen to the other. This hides the refreshing effect from the user while the frame loads. The `x264` application is executed on a FPGA running a MicroBlaze host processor. By running the particular kernel on the ρ-VEX co-processor, the execution time can be decreased by making use of parallel processing.

3 Exploring the x264 application

Duing the first lab we got familiar with the x264 application, a powerfull coding tool to remove temporal redundancy by using motion estimation. In Section 3.1 we will discuss the various computational intenstive kernels within the x264 algorithm. The choosen kernel will be explained in Section 3.2, the provided memory layout of the FPGA architecture in Section 3.3 and the actual implementation will be discussed in Section 3.4.

3.1 Detecting the Computationally Most Intensive Kernel

A few input files are provided in the first lab to demonstrate the compile and run commands of the x264 application. The `.y4m` input files are decoded to create a short `.mkv` movie using this command in the terminal: `./x264 eledream_64x32_3.y4m -o testframe.mkv`. 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). The number of function calls is also shown, as well as the total execution time for each input file. These five input `.y4m` files, vary in resolution (either 64x32 or 640x320 pixels) and amount of frames (either 1, 3, 8, 32 or 128). Profiling the x264 execution for the `.y4m` files that are provided by the lab leads to the ranking show in table 1.

Input file	Time (sec)	Share (%)	Calls	Kernel name
eledream_32x18_1.y4m	0.02	100.00	71	x264_analyse_init_costs
		0.00	1646	x264_free
		0.00	784	x264_cabac_encode_desicion_c
eledream_64x32_3.y4m	0.03	66.67	71	x264_analyse_init_costs
		33.33	1971	x264_pixel_satd_4x4
		0.00	4830	x264_pixel_satd_8x4
eledream_640x320_8.y4m	1.61	14.29	1599044	x264_pixel_satd_8x4
		11.80	570708	x264_get_ref
		4.97	38770	x264_pixel_satd_x4_16x16
eledream_640x320_32.y4m	8.54	20.61	7228633	get_ref
		13.23	15386501	x264_pixel_satd_8x4
		4.57	1009690	x264_pixel_satd_x4_8x8
eledream_640x320_128.y4m	29.86	17.48	21956292	get_ref
		14.17	49862831	x264_pixel_satd_8x4
		6.56	1023315	x264_pixel_satd_x4_16x16

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

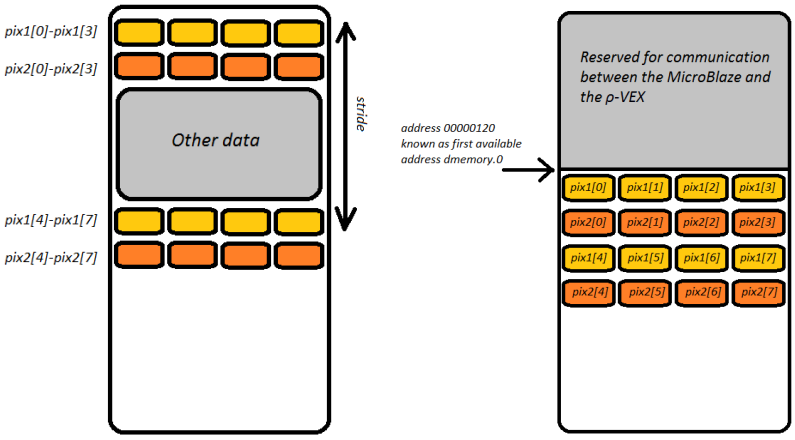
3.2 The `pixel_satd_8x4` kernel

Given these statistics, we decide to extract the `x264_pixel_satd_8x4 kernel`, since its share in execution time increases as the files become larger. This kernel evaluates the Sum of Transformed Differences (SATD) between a 8x4 pixel block from the input stream and reference blocks. The SATD is a metric used for video compression where the differences between the pixels are taken and put into a frequency transform, usually a Hadamard transform. These blocks can have various sizes depending on the inputs and constraint of the system varying from 4x4 pixel blocks to 8x4 blocks and even 16x16 blocks. The reference block with the lowest SATD value can be used to estimate motion in video coding to remove temporal redundancy.

3.3 Memory layout of the MicroBlaze

When writing pixels to the ρ-VEX for SATD calculation, these pixels need to be picked from the MicroBlaze memory. However, these pixels are not written as one block of contiguous bytes. Rows of four bytes

(MicroBlaze has 32-bit registers) are interrupted by a fixed amount of memory called a stride. This value is often given in bytes to be skipped in order to continue reading the particular data file. Figure 1a shows the concept of strides for two pixels from the MicroBlaze memory, which are splitted into two rows of 4 bytes since a pixels consists of eight bytes. When writing data to the ρ-VEX memory we use a stride of four bytes, resulting in a configuration as shown in 1b



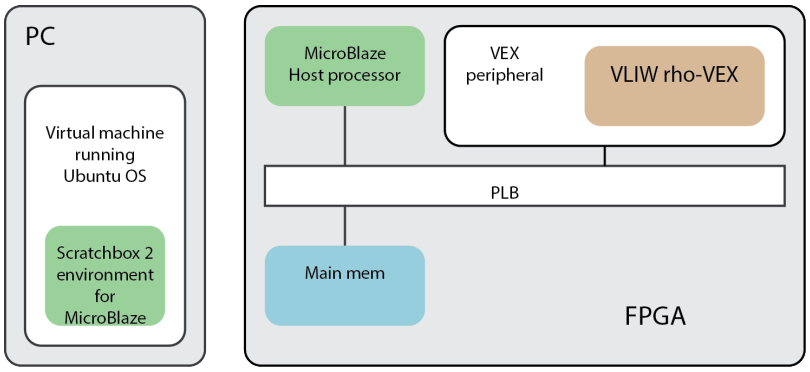
(a) Large unknown stride

(b) Stride of four bytes

Figuur 1: MicroBlaze memory using strides to split up data files

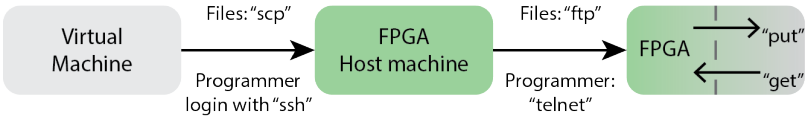
3.4 Executing an Application on the Development Board and ρ-VEX

When executing `./configure`, a file is created for configuration of the application. However, the resulting `config.mak` file is made for applications running on the guest (Ubuntu). In order to configure for MicroBlaze, the `config.mak` file has to altered. All references to `m32` have to be removed and the `--DWORDS_BIGENDIAN` flag has to be added to the `CFLAGS` variable. This has to be done everytime when configurating the application for MicroBlaze.
After doing this, the application now can be 'made' for MicroBlaze by first moving to the Scratchbox 2 environment for MicroBlaze and then execute the `make` command. Fig. 2 shows a block diagram of both the platforms.



Figuur 2: Block diagram of the Virtual Machine and the ERA platform

In order to run the application on the MicroBlaze host processor, the `x264` executable must be compressed along with an input stream in a `tar.gz` file. This file can be put, via the FPGA host machine, on one of the three FPGAs using the `scp`, `ftp` and `put` command. For the programmer to connect to the development board, one must use the `ssh` and `telnet` command. See also fig 3.



Figuur 3: Ubuntu commands for putting files on and navigating to the FPGAs

For the first lab, the `satd_8x4` kernel has to be:

- extracted into a separate file
- supplemented to a proper .c file which can be compiled and run using the ρ-VEX
- included in the makefile that creates a bytecode and bytedata file of the kernel
- debugged

Looking back, the debugging part has been chasing us all the way through the lab. Not only did we suffer from bugs in our source code, the ρ-VEX itself did also have shortcomings that had to be circumvented by downloading several fixes, competing for FPGA availability and break downs of the entire host machine due to unsufficient capacity for the amount of students.

4 Approach

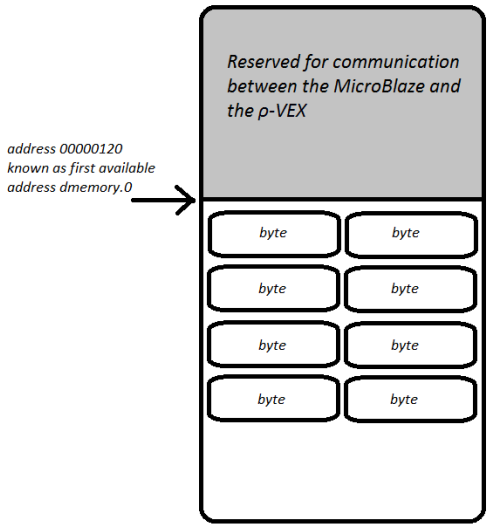
Our extracted kernel can be found in appendix A. In order to make the extracted kernel qualified for compilation and execution, a few things have to be altered. First, the new file has to be recognized by the `makefile` in the rovex-examples directory. By executing `make byte-` command two files are 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 4 shows the ρ-VEX memory layout for our kernel.

4.1 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 written to is 120. Thus, when writing the result of the kernel to the logical address '0', it actually should be written to the physical address '120' of the data memory.



Figuur 4: Memory layout of the ρ-VEX

Despite the fact that this is a rather simplistic operation, it took us some struggling to have this action confirmed as correctly. 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 had been made to eliminate the issue of the logical and physical addresses. Since the purpose of `__DATA_START` was now unclear we decided to remove all references to it, which felt like we had wasted lots of time. *Klopt dit wel?* Unfortunately, we still had major problems getting the application to run 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.

4.2 Constraints of the ρ-VEX

Using the ρ-VEX as a co-processor comes with some limitations. First, the ρ-VEX does not understand C commands, so using `printf` statements to find bugs in the extracted kernel file is not possible. Everytime the application is being run concurrently on the ρ-VEX, the status is being inspected by placing `printf` statements in the `pixel.c` file. Also, the ρ-VEX is very strict in the order of variable declarations and initialization. One of our bugs involved `int i` being declared and initialized, followed by a declaration of `int result`. This is not supported by the ρ-VEX, since it wants to have `int i, result;` both to be declared first, before initializing `i = 4;`.
When running an extracted kernel on the ρ-VEX, the kernel is unable to obtain information from previous code. Consequently, all parameters have to be re-defined. Table 2 shows all required type definitions.

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.

4.3 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 adjusted. Instead of calculating the SATD, it should send the instructions and input to the ρ-VEX. The commands 5a and 5b 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.

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

(a) Bytecode

```
// Open the data register and check whether it has opened correctly
// point data mem to 120 on the rvex, address 0 on the microblaze,
// write pixel data to memory written in a sequential manner to
// 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 failed");
    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

```
/* ----------------------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 rvex");

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

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

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

Figuur 5: Commands for reading from and writing to the ρ-VEX

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 5c.

4.4 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 new data have to be sent to the ρ-VEX every time the kernel is being called. The data 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.

5 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.

```
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 6: Commands for reading from and writing to the ρ-VEX

5.1 Using Test Pixels for Debugging

With all these logical and physical addresses, it would be nice to know what is written on which address. Unfortunately, the only way to find out is by doing `hexdumps`, resulting in an array with hexadecimal numbers. In order to quickly check where the ρ-VEX is writing the data, a debugging file `Debugging_pixel.c` is created. When using this debugging file, `bytecode` is manually put into the instruction memory using `scp`, `ftp` and `put`. The data that is being sent from the MicroBlaze are two pre-defined pixels as seen in figure 6a.This way, we can have certain expectations when doing a `hexdump` to examine the registers.

5.2 The Fix

Halfway the lab, a fix was introduced. Setting the physical start addres to 0x120 was now done 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 logical address 0 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 used `Debugging_pixel.c` to write the test pixels to `rvex-dmemory`.

5.3 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 6b. As you can see, the first two byte are not matching. At first, we thought this 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.

5.4 Endianness

The MicroBlaze and the ρ-VEX understand basic operations like `open()`, `close()`, `read()` and `write()`. However, they differ in Endiannes. ρ-VEX operates in Little Endian, meaning the bytes are read from right to left, as seen in figure 7a. MicroBlaze, on the other hand, operates in Big Endian and thus reads the bytes from left to right (see figure 7b). This becomes an issue when reading the `result` variable from the data memory, as we will get the bytes in reversed order. This problem is solved by manually reversing the order of the `result` bytes, realized by the piece of code in figure 7c.

Byte[3] | Byte[2] | Byte[1] | Byte[0]

(a) Little Endian

Byte[0] | Byte[1] | Byte[2] | Byte[3]

(b) Big Endian

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

(c) Reversing the order of bytes

Figuur 7: Different reading direction due to Endianness

5.5 Setting pointer to address 0 using `lseek`

When writing pixels to the data memory of the ρ-VEX, it is important to know exactly where they end up. This situation can be solved by using `lseek`, a system call that is used to change the location of the read/write pointer of a file descriptor. When ordering `lseek(data, 0, SEEK_SET)`, the pointer in the ρ-VEX moves to the address that is the start address from his point of view (thus, physical address 0x120). We first made a mistake by putting 0x120 as `lseek` parameter, which gives an error since the MicroBlaze can not see into the physical address of the ρ-VEX. Translating the logical address 0 to the physical address 0x120 is done by the linker and is not of our concern.

5.6 Using Test Pixels for Debugging

With all these logical and physical addresses, it would be nice to know what is written on which address. Unfortunately, the only way to find out is by doing `hexdumps`, resulting in an array with hexadecimal numbers. In order to quickly check where the ρ-VEX is writing the data, a debugging file `Debugging_pixel.c` is created. When using this debugging file, `bytecode` is manually put into the instruction memory using `scp`, `ftp` and `put`. The data that is being sent from the MicroBlaze are two pre-defined pixels as seen in figure 6a.

6 Extracting a Second Kernel

For lab 3, a second kernel has to be extracted in order to achieve an even higher speed up. For this assignment, we wanted to choose our kernel more wisely. Given the constant competition for the use of the FPGAs, `bytecode` has to be written to the instruction memory everytime the kernel is being called. In combination with the data to be processed by the kernel, this leads to huge data traffic which is actually slowing down the application. This is why we wanted, at first, to pick a kernel that is not being called too often but requires a lot of heavy computation. A candidate for this would be the `halfpel` kernel, which has quite its share in execution time but not that many functions calls (*halfpel zit helaas niet in de top 10 kernels, dus kan harde getallen noemen.* Unfortunately, there is no time for writing an entire new file for the extraction of the `halfpel` kernel. Instead, we will pick a kernel similar to `pixel_satd_8x4` which also have a large share in the execution time: `pixel_satd_4x4`.

6.1 Extract `pixel_satd_4x4`

*** uitleg aanvulling rovex code ***

7 Evaluation

- How much speedup did you obtain?
- Is this what you had expected?
- Give a theoretical calculation for the speedup you should have expected and compare it to the practical result
- What were the results of the additional assignment? How did it affect the speed of the application?

A Extracted kernels `pixel_satd_8x4` and `pixel_satd_4x4`

```
// 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_SUM);
        a3 = (datapoint[3] - datapoint[11]) + ((sum2_t)(datapoint[7] - datapoint[15]) << BITS_PER_SUM);
        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;
}
```