TUDELFT
RECONFIGURABLE COMPUTING DESIGN
LAB REPORT
2012-2013

LAB PARTICIPANTS:

1)Krommydas Kyriakos
ID: 4243161
Email: krommydas.k@gmail.com
Affiliation: TUDelft MSc Embedded Systems

2)Papadopoulos Charalampos
ID: 4192141
Email: xaris.papadopoulos@hotmail.com
Affiliation: TUDelft MSc Embedded Systems

Abstract

The project for Reconfigurable Computing course involves extracting a computationally intensive kernel from an application, and accelerating it on a coprocessor. In our case, we had to work on x264, which is a free software library for encoding video streams into the H.264 format. It is released under the GPL. H.264 is a standard for video compression, and is equivalent to MPEG-4 Part 10, or MPEG-4 AVC (for Advanced Video Coding).

STEPS

1) Profiling the application
2) Extracting the kernel (most computationally intensive function)
3) Test the correct behavior of the extracted kernel
4) Making a wrapper at x264 for transferring data to the rvex
5) Synchronize the kernel on rvex to work with the rest of x264 application at microblaze
6) Test the output of the whole x264 application with the rvex
7) Measure the speedup
8) Results and conclusions

Profiling the application

Initially we ran the software version of our program on our pc processor. After managing to do that correctly, we profiled the execution of this software version on our PC processor using the gprof tool. We found that the most time consuming function is the *x264_pixel_satd_8x4()* function. We suppose that this function is also going to be the most time consuming one on the microblaze. For that reason we implemented that function on the ρ-VEX coprocessor. In order to do that, we wrote a wrapper that transfers arguments to the ρ-VEX, then resets and starts the ρ-VEX and then waits until the ρ-VEX finishes and gets the results back.

Extracting the kernel (most computationally intensive function)

As it was mentioned before, the candidate-function to become the kernel was the *x264_pixel_satd_8x4().* In order to successfully implemented it on the rvex core though, we also had to transfer 2 macro's, one helper function and some type definitions, on which the function was depending on. After successfully compiling the kernel for running into the rvex, we had to test it's output, in order to see if it has the same functionality as within the x264 application.

Test the correct behavior of the extracted kernel

Testing the correct behavior of a function requires some sample data and the outcome which should provide with them. The *x264_pixel_satd_8x4()* function takes as arguments two pixel arrays and two integer strides and produces one integer result. It makes some calculations on four 'frames' of eight pixel each. The distance of each frame is given by the strides (different for each array). For example, it will start do calculations on the first eight pixels the 2 pixel array arguments point to. Then, it will move the working 'frame' on the next 8 pixels of each array by incrementing the two array pointers by the amount of the stride. The same thing will be repeated for two more times, making the total 'working frames' 4. Therefore, for one successful execution of the function we need two pixel arrays, with size equal to four times the amount of their stride. To get those, together with their strides, we placed some code inside the function

body in the x264 application. The behavior of the code was simple: copying the arguments with which the function is being called for the first time, together with it's result, to a separate file. The next step was to make a simple program in rvex which should call the kernel-function with those sample data. Then, we simulated it's execution on rvex with a simulator (modelSim). When we managed to get the same result as the one we draw from the execution of the same kernel on x264 with the same data, we were sure that we had the correct version of the kernel transferred. After wards, we had to implement a wrapper around the *x264_pixel_satd_8x4()* function on the x264 application which would feed the kernel on rvex with data and read it's result.

Making a wrapper at x264 for transferring data to the rvex

Before, starting implementing the wrapper we had to figure out the order in which the arguments and the result of the kernel-function will be laid on the memory. We used the following convention (starting at the start of rvex memory file):
First, the stride for the first pixel array, an integer of 4 bytes.
Second, the stride for the second pixel array, an integer of 4 bytes.
Third, the first pixel array, four times the first stride amount of pixels.
Fourth, the second pixel array, four times the second stride amount of pixels.
Fifth, the result of the kernel's execution.
The communication between rvex and microblaze (where x264 would run) is being made through two files. One of those files controls rvex and the other maps to it's memory. Therefore, the wrapper had to open the data file and transfer the function's arguments to it with the above mentioned order. In addition it had to reset and start the rvex core after tha transfer through the status file of rvex. Then, it had to wait until rvex core had finished execution of the kernel -function by continuously reading it's status. After that, it had to read the outcome – result of the kernel from the right place at the data file of rvex (with the help of lseek). However, because the result is an integer of 4 bytes and microblaze is a big endian processor, in contrast with the small endian rvex, we had to change endianness of the result. The final 'duties' of the wrapped was to return the result and close the two rvex files. The next step was to make the kernel-function at rvex, feed it's data and place it's result at the right place of rvex memory.

Synchronize the kernel on rvex to work with the rest of x264 application at microblaze

At the simulation of rvex kernel at modelSim, we had it read it's sample data from a static place of rvex-memory. This time, however, we had to make it, get the data, in synchronization with the way the microblaze wrapper was sending them and accordingly return the result. As a point of reference, we were using an extern int (__DATA_START) which were pointing to the start of data of rvex. Then, we made it read data in the same order in which the microblaze-wrapper was sending it. The only difference was that we had to change the endianness of the two four bytes integer strides before call the kernel-function with them. The result was accordingly placed in the right position in memory. After that, the rvex-kernel returned with 0, thus, finishing it's execution. Having finished this part as well, we had to test the correct execution of the whole x264 application with the rvex acceleration.

Test the output of the whole x264 application with the rvex

The first thing we had to do, was to run the original x264 application (without the rvex) on the microblaz and get it's result mkv file, which would be our reference point for correct execution. Subsequently, we compiled the x264 application with the *x264_pixel_satd_8x4()* function's core replaced with the above mentioned wrapper and uploaded on the working platform. We also uploaded the kernel program and loaded it's instructions code on the rvex core. Then, we ran the x264 application and got it's new mkv result file. If this file was identical to the original one than we would have the same result as without the rvex and therefore we would have successfully accelerated the x264 application with the rvex. To compare those two files we used the *diff* command of unix. However, we also had to measure the new execution time of the x264 application, in order to find out the speed up we got.

Measure the speedup

In order to check that the ρVEX produces correct results, we produced the output for several different videos as inputs, using the microblaze only and then the combination of microblaze and ρ-VEX. We compared the outputs and they were the same. As a result, we are sure that our hardware works correctly.

After that we took some timing measurements concerning both the simple version that runs only on the microblaze and the version that runs on the combined version of ρ-VEX and microblaze. We present the results below:

For the simple version:

Total duration of execution	76.46 seconds
Average time of x264_pixel_satd_8x4() function	87 microseconds

For the combined version:

Total duration of execution	591.79 seconds
Average time of x264_pixel_satd_8x4() function	65622 microseconds

Analysis of the combined version timing (average times):

Open ρvex communication files	990 microseconds
Transfer data to ρvex	227 microseconds
Reset and Start ρvex	63542 microseconds
Wait for ρvex to finish	591 microseconds
Close files and return	272 microseconds

Speed down computation:

The total speed down can be calculated as following:

(76.46-591.79)/ 76.46 = 674% speed down

The speed down concerning the x264_pixel_satd_8x4() function:

(87-65622)/87 = 75327% speed down

Results and conclusions

What we get by measuring the timing requirements of the 'accelerated' version of the x264 (with the rvex addition) is that it is significant slower from the original one due to the overhead of the communication between the rvex and the microblaze. Apart from, writing and reading from a file – which is for sure slower from the same in memory – microblaze must also wait for rvex to finish before continuing it's execution. However, most of the time is 'wasted' on writing and reading from the status file of rvex, in other words when resetting and starting rvex from microblaze. Therefore, it is being proved that an acceleration of an application can not be done this way. Kernel code has to be implemented on hardware, in order to be significant faster and overcome any communication overheads, if any. That is the essence of reconfigurable computing. If we wanted to accelerate the 264 application implicitly on software (with another processor core) we would have to implement parallel computing instead.

APPENDIX A

x264_pixel_satd_8x4() function's wrapper code, which runs at microblaze

```
/* This function changes endianness of an unsigned 32 bytes value */

void changeEndianessInt(uint32_t *val)

{

     *val = (*val >> 24) |

        ((*val<<8) & 0x00FF0000) |

        ((*val>>8) & 0x0000FF00) |

        (*val<24);



}


static NOINLINE int x264_pixel_satd_8x4( pixel *pix1, intptr_t i_pix1, pixel *pix2,
intptr_t i_pix2 )

{

   /* Initilizations */

   int result = 0;

   char status = '0';

   char init = '2';

   char start = '1';

   /* Open rvex communication files */

   int mem = open("/dev/rvex-dmemory.0", O_RDWR);

   int ctl = open("/sys/devices/virtual/rvex_core/rvex-smemory.0/rvex_core_ctl",
O_WRONLY);

   int statusFile = open("/sys/devices/virtual/rvex_core/rvex-
smemory.0/rvex_core_status", O_RDONLY);
```

```
/* Transfer data to rvex with the following order:
   1)Stride 1 int, 4 bytes
   2)Stride 2 int, 4 bytes
   3)Pixel array 1, 4 * Stride 1 bytes
   4)Pixel array 2, 4 * Stride 2 bytes
*/
write(mem, &i_pix1, 4);
write(mem, &i_pix2, 4);
write(mem, pix1, 4 * i_pix1);
write(mem, pix2, 4 * i_pix2);
/* Reset and Start rvex */
write(ctl, &init, 1);
write(ctl, &start, 1);
/* Read rvex status */
read(statusFile, &status, 1);
/* Until rvex is not finished */
while( status != '3')
   read(status, &status, 1);
/* The result will be after the pixel array 2 in rvex memory.
   Seek from the start of rvex-memory and skip as many data as we transfered*/
lseek(mem,  (4 * i_pix1) + (4 * i_pix2) + 8, 0);
read(mem, &result, 4);
changeEndianessInt(&result);       //make result Big Endian
/* Close files and return */
close(mem);
close(ctl);
close(statusFile);
return result; }
```

APPENDIX B

x264_pixel_satd_8x4() kernel's code, which runs at rvex

```
#include <sys/_inttypes.h>

typedef uint8_t pixel;

typedef uint16_t sum_t;

typedef uint32_t sum2_t;

typedef int intptr_t;


#define BITS_PER_SUM (8 * sizeof(sum_t))

#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;\

}


static inline 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( pixel *pix1, intptr_t  i_pix1, pixel *pix2, intptr_t  i_pix2 )
{
    sum2_t tmp[4][4];
    sum2_t a0, a1, a2, a3;
    sum2_t sum = 0;
    int i;
    for( i = 0; i < 4; i++, pix1 += i_pix1, pix2 += i_pix2 )
    {
        a0 = (pix1[0] - pix2[0]) + ((sum2_t)(pix1[4] - pix2[4]) << BITS_PER_SUM);
        a1 = (pix1[1] - pix2[1]) + ((sum2_t)(pix1[5] - pix2[5]) << BITS_PER_SUM);
        a2 = (pix1[2] - pix2[2]) + ((sum2_t)(pix1[6] - pix2[6]) << BITS_PER_SUM);
        a3 = (pix1[3] - pix2[3]) + ((sum2_t)(pix1[7] - pix2[7]) << 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;
}

/* This function changes endianness of an unsigned 32 bytes value */
void changeEndianessInt(uint32_t *val)
{
     *val = (*val >> 24) |
        ((*val<<8) & 0x00FF0000) |
```

```
        ((*val>>8) & 0x0000FF00) |
        (*val<24);

}

/* The adress of the following integer points to the start of the rvex data. */
extern int __DATA_START;

int main(void)
{
   /* Initializations. */
   pixel *pixel1, *pixel2;
   int *result;
   /* Read the transferred data from microblaze with the following order:
      1)Stride 1 int, where __DATA_START points originally.
      2)Stride 2 int, it will be after Stride 1, which is an integer, therefore
         we add 1 to __DATA_START.
      3)Pixel array 1, it will be after Stride 2, which is an integer, therefore
         we add 2 to __DATA_START.
      4)Pixel array 2, it will be after Pixel Array 1, which is 4 * Stride 1 bytes,
         therefore we had to add 2 to __DATA_START, plus the previous amount divided
         by 4 (integer pointer).
       5)Result, it will be after Pixel Array 2, which is 4 * Stride 2 bytes,
         therefore we had to add 2 to __DATA_START, plus the previous amount divided
         by 4 (integer pointer), plus the Pixel Array's 1 integer's amount of bytes.
   */
   intptr_t *ptr1 = &__DATA_START;
   intptr_t *ptr2 = &__DATA_START + 1;
```

```
    /* Before doing any calculations with the two strides, we have to change their
       endianness because they are integers with 4 bytes each and rvex has a different
       endianness from microblaze, from which we get those 2 values. */
    changeEndianessInt(ptr1);
    changeEndianessInt(ptr2);
    pixel1 = &__DATA_START + 2;
    pixel2 = &__DATA_START + 2 + *ptr1;
    result = &__DATA_START + 2 + *ptr1 + *ptr2;

    /* Execute the function and write it's result at the right memory place, so as
       microblaze will read it afterwards */
    *result = x264_pixel_satd_8x4((pixel*)pixel1, *ptr1, (pixel*)pixel2, *ptr2);

    /* Finished, return */
    return 0;
}
```