

Project 2: Benchmarking, Profiling and Optimizing an Application for an ARM-based Embedded Device

Nombre: Ing. Andrés Gómez Jiménez		Carne: 200935203
Nombre: Ing. Randy Céspedes Deliyore		Carne: 201054417

1 Introduction

In this document, we present the results obtained while benchmarking, analyzing and optimizing a simple application for an embedded system. In section 2 we present the results of analyzing the RPI4 hardware with the existing CoreMark benchmarks. In section 3, we present the followed steps and results of prototyping the RGB to YUV application using OpenCV. In section 4, we present our initial implementation of a C application for the image conversion and the results of benchmarking it to obtain baseline results. In section 5, we present the results of using *perf* to analyze in more detail the hot-spots of our application. In sections 6.1, 6.2 and 6.3 we present the results of attempting to optimize our application with different methods: Neon intrinsics, pthreads and OpenMP respectively. Finally, in section 7, we present the result of combining different techniques to obtain an even higher speedup of our application.

2 CoreMark and CoreMark-Pro: Benchmarking the RPI4

2.1 CoreMark

We are going to start by answering the proposed questions on CoreMark.

1. Briefly describe the theory of operation of the benchmark algorithm. Make sure you add a short description of its three main algorithms.

CoreMark provides a performance indicator for embedded processors using basic data structures and algorithms that are common in real applications. The key algorithms utilized by CoreMark are described below [1]:

(a) Linked List: It consists of reversing, searching or sorting a list according to different parameters. Each list item can either contain a pre-computed value or a directive to call a specific algorithm to provide a value during sorting. The data space is partitioned into two blocks, one with the list itself and the other with the data items.

(b) Matrix Multiply: CoreMark performs multiplication on the input matrices with a constant, a vector, or another matrix. To validate that all results have been performed, CoreMark computes a CRC on the results from the matrix test.

(c) State Machine: CoreMark uses switch and if statements to exercise the CPU control structure. The state machine receives a string (stream of bytes) that is mapped to a number that can indicate a transition to 9 different states. The test is configured to ensure passing to all available states.

2. How does the CoreMark benchmark try to deal with compiler optimization to come up with a standardized result?

According to [2], CoreMark follows certain strategies to ensure the desired data for the benchmark is calculated at runtime (by the processor to benchmark) and not by the compiler. For this they would rely on system functions (like scanf), command line parameters or volatile variables, whose values cannot be pre-determined by the compiler. They will be used/invoked before the portion of code to time/benchmark. The values these methods would provide are standardized since the intent is not randomizing, but that they can't be determined in compile time.

3. What is the difference between the “core_portme” and the “core” files? Are we allowed to modify all of them?

According to [2], the “core_portme” files are intended to be used for cross-compilation. This can be modified to indicate the specific platform to be used before compiling. The “core” files are not expected to be modified since these are generic for the benchmark.

We have downloaded and compiled CoreMark to run on the RPI4. In figure 1 we can observe the results of running it where arguments have been provided to run 75,000 iterations for it to last longer than 20 seconds. We can observe that in this case the CoreMark given rate is of 3258.32, which is the same as iterations of the algorithm per second.

```
root@raspberrypi4-64:~# ./coremark.exe 0x0 0x0 0x66 75000 7 1 2000
2K performance run parameters for coremark.
CoreMark Size      : 666
Total ticks        : 23018
Total time (secs): 23.018000
Iterations/Sec     : 3258.319576
Iterations         : 75000
Compiler version   : GCC9.2.0
Compiler flags     : -O2 -DPERFORMANCE_RUN=1-lrt
Memory location    : Please put data memory location here
                    (e.g. code in flash, data on heap etc)
seedcrc           : 0xe9f5
[0]crclist        : 0xe714
[0]crcmatrix      : 0x1fd7
[0]crcstate       : 0x8e3a
[0]crcfinal       : 0x382f
Correct operation validated. See README.md for run and reporting rules.
CoreMark 1.0 : 3258.319576 / GCC9.2.0 -O2 -DPERFORMANCE_RUN=1-lrt / Heap
```

Figure 1: Running CoreMark on RPI4

2.1.1 Experiment 1: CoreMark Benchmark and Multi-Threading

We will review the behavior of the CoreMark benchmark while configuring it to use a different number of concurrent threads. The number of threads that CoreMark uses can be set by changing the `MULTITHREAD` variable on the `core_portme.h` file. We will configure CoreMark to use pthreads by enabling the variable `P_THREAD` on the same header file and compiling with the `-pthread` flag.

The results obtained are observed on figure 2. We can observe how the performance increases almost linearly from one to four threads where it seems to saturate and even degrade when adding more threads. The largest improvement is obtained when increasing from one to two threads, although this is very similar to the next two steps. Performance is not significantly improved after four threads since the RPI4 only has four cores, which means that beyond there threads actually compete for a resource. Since the Linux Completely Fair Scheduler attempts to assign resources equally, each of the threads would get similar chunks of time for running. One problem with adding more threads is that more overhead is added in context switching, which ends up degrading performance.

We also tested performance while using fork instead of pthreads for a couple of scenarios. The performance results were very similar and following the same pattern for the curve.

2.1.2 Experiment 2: Compiler Optimization of “non-optimizable” Code

We experience with different levels of compiler optimizations for the CoreMark benchmark: O0, O1, Os, O2, O3 and Ofast. According to the theory Ofast would yield the best results for the benchmark. For this experiments we keep the number of threads to one to only see the effect of the compiler optimization.

The results obtained are observed on figure 3. We can observe the biggest increase in performance when transitioning from O0 to O1. The worst performance is obtained when using Os, which optimizes for code size sacrificing performance. The best results are obtained when using the O3 optimization, although results are relatively similar to those obtained with O2 and Ofast. Considering the significant difference between greater or equal to O2 vs less than O2 optimizations, we can conclude that the benchmark is not independent from the compiler optimization.

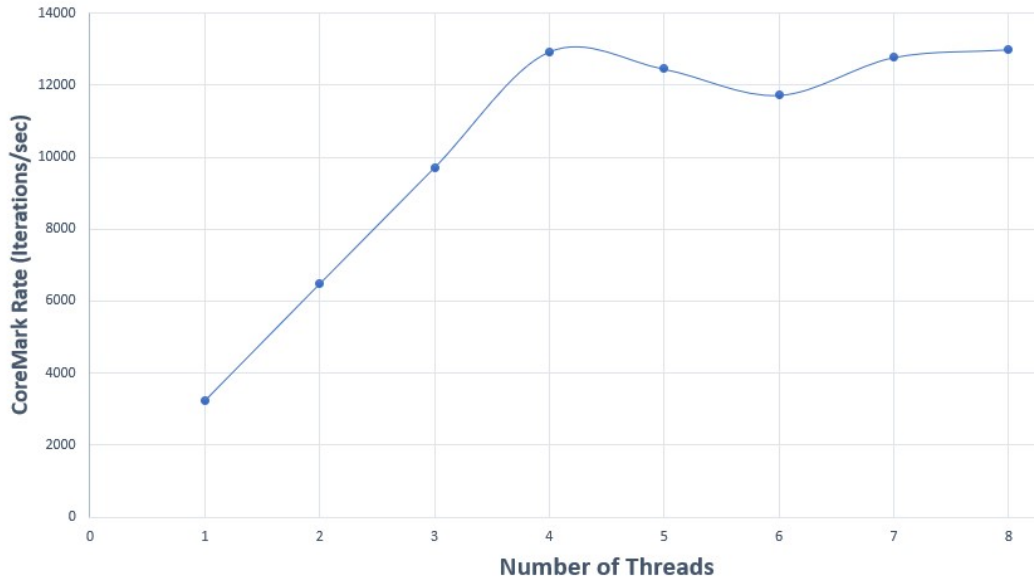


Figure 2: CoreMark Benchmark Performance Versus Number of Threads

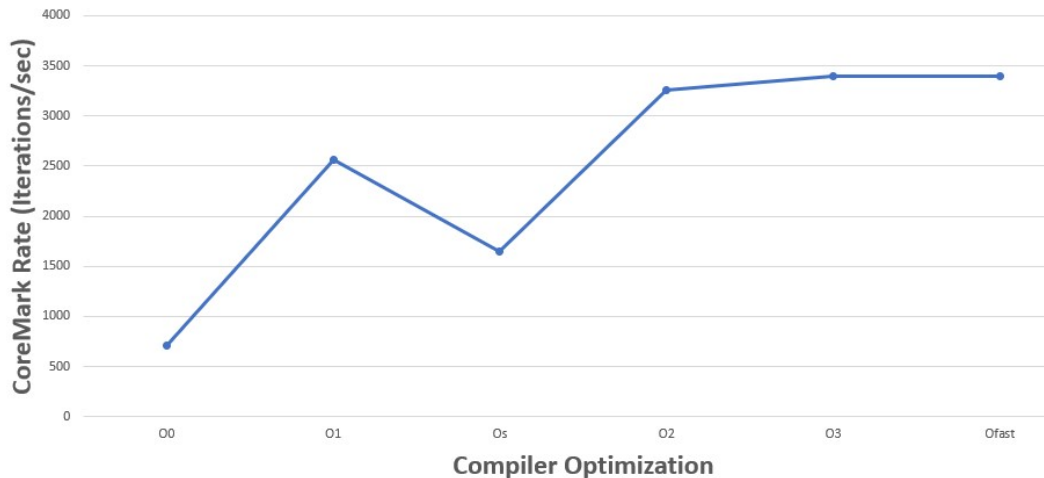


Figure 3: CoreMark Benchmark Performance Versus Compiler Optimization

2.1.3 Best Combination

Based on the results obtained on the previous sections, we conduct a few more experiments to determine the best achievable result for the CoreMark benchmark using our platform. We summarize our results on Table 1. The best result of 13547.99 was obtained with 8 threads and using Fork for parallelization, although the rest of results on the table are relatively similar. Something interesting and somehow unexpected is that the official rate given on the EEMBC site is of 33100 for the BCM2711 used on the RPI4, which they document is obtained with four threads using fork. The only major difference is that their test is conducted using Raspbian GNU/Linux, although the difference is significative to only be attributed to this factor.

2.2 CoreMark-Pro

In the same manner, we are going to start by answering the proposed questions on CoreMark-Pro.

Table 1: Best Results Obtained for CoreMark BenchMark

Compiler	Optimization	Number of Threads	Parallelization Type	CoreMark (iterations/second)
	O3	4	PTHREADS	13485.57
	O3	8	PTHREADS	13539.13
	O3	4	FORK	13505.60
	O3	8	FORK	13547.99

1. How does the algorithm differ from the original one? What has improved?

According to [3], while CoreMark stresses the CPU pipeline, CoreMark-Pro tests the entire processor, adding comprehensive support for multicore technology, a combination of integer and floating-point workloads, and data sets for utilizing larger memory subsystems.

2. Overview its integer and floating-point workloads without explaining in detail the 24 FORTRAN kernels.

According to [3], the integer workloads include JPEG compression, ZIP compression, an XML parser, the SHA-256 Secure Hash Algorithm, and a more memory-intensive version of the original CoreMark. The floating-point workloads include a fast Fourier transform (FFT), a linear algebra routine derived from LINPACK, a greatly improved version of the Livermore loops benchmark, and a neural net algorithm to evaluate patterns.

3. How are the multiple workloads combined to summarize results in one single score?

According to [4], the CoreMark-PRO score is a weighted geometric mean of each workload that runs in the benchmark.

2.2.1 Running CoreMark-Pro on the RPI4

For the CoreMark-Pro, we decide to compile the source directly on the RPI4 for simplicity and also to experience with a different approach than cross-compilation. The code compiles and runs just fine when following this approach. We will run the benchmark using different number of contexts, which ends up being translated as varying the number of threads. We compile and run the benchmark with the command indicated below, where *cX* configures the number of contexts to use.

```
$ make TARGET=linux64 XCMD='-cX' certify -all
```

The results obtained can be observed on figure 4. Similarly as CoreMark, performance increases almost linearly until reaching four contexts. From there performance has a small decrease and settles around a constant value. Given its characteristics and the scenarios covered by CoreMark-Pro, it offers a more robust benchmark mechanism for our embedded system than the standard CoreMark.

3 Prototyping RGB to YUV Transformation using OpenCV

To prototype the RGB to YUV transformation using openCV the program named *Prototyping_OpenCV* was created. This code can be seen below. The logic is fairly simple and it uses the default openCV function called *cvtColor* with the *COLOR_BGR2YUV* parameter, which allow the system to covert and RGB image into a YUV one.

```
// Function to convert RGV to YUV using OpenCV
int main(int argc, char **argv) {
    Mat rgb_image;
    Mat yuv_image;

    printf("Loading original RGB image ...\n");
    rgb_image = imread("imagejpg.jpg", IMREAD_UNCHANGED);
    // Confirm image was loaded correctly
    if(!rgb_image.data)
    {
        printf("There was a problem loading the RBG image. Aborting!\n");
    }
}
```

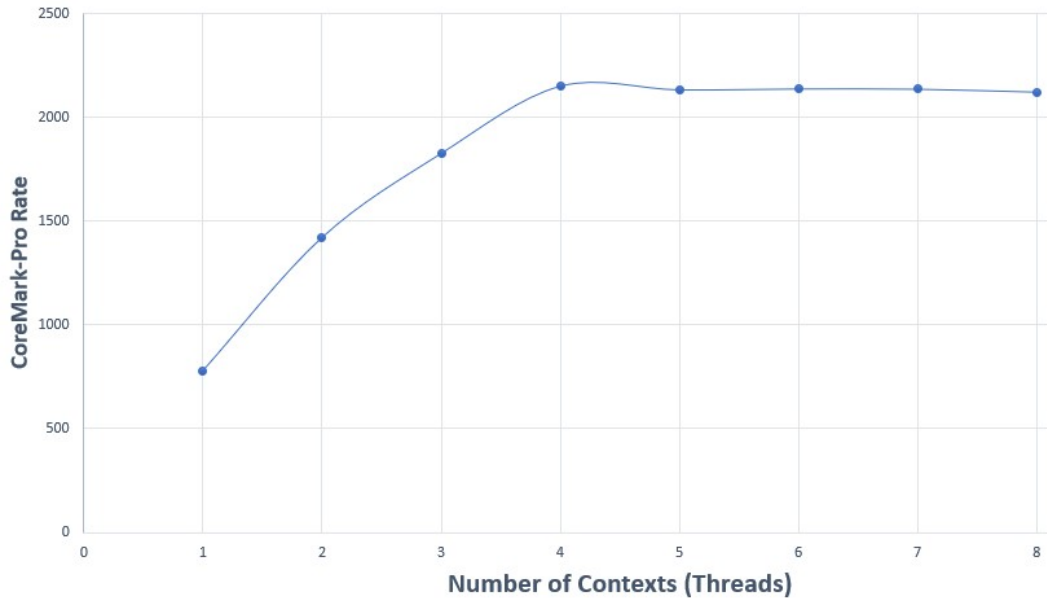


Figure 4: CoreMark-Pro Benchmark Performance Versus Number of Contexts

```

    return -1;
}
int img_height = rgb_image.size[0];
int img_width = rgb_image.size[1];
printf("Image loaded. Height %d. Width %d\n", img_height, img_width);

printf("Converting image to YUV ..\n");
yuv_image = rgb_image.clone();
cvtColor(rgb_image, yuv_image, COLOR_BGR2YUV);

printf("Saving converted image to new file ...\n");
FILE * output_file = fopen("imageyuv.yuv", "wb");
size_t bytes_written = fwrite(yuv_image.data, 1, 3*img_height*img_width, output_file);
printf("File imageyuv.yuv written with %ld bytes\n", bytes_written);

return 0;
}

```

In order to be able to run the code in the RPI4 the OpenCV libraries had to be included in the Yocto File System. The first step to do this was to add OpenCV version 4.1.0 using the open-embedded meta layer which is available in the community. This meta layer was downloaded and placed within the poky folder as it can be seen in figure 5. Then the *bblayers* file was modified to include the open-embedded meta layer as it can be seen on figure 6.

Finally, before recompiling the Yocto file system, the *local configuration* file was modified to include the OpenCV libraries as it can be seen in figure 7. This is achieved by including the flags *-opencv*, *-libopencv-core-dev*, *-libopencv-imgproc-dev*, and *opencv-dev* in the *CORE_IMAGE_EXTRA_INSTALL* variable. After this, the Yocto file system was successfully recompiled including the additional meta layer as it can be seen in figure 8.

The C++ code had to be cross-compiled to run in the RPI4 new file system. After the Yocto compilation, an SDK was generated and used in Eclipse IDE (through the Yocto *oxygen 2.6.1* plugin). The *CMakeList.txt* file needed to be modified to include the OpenCV libraries and allow the autohandled compilation using CMake, as it can be seen in figure 9.

Once the OpenCV application was moved to the RPI4, it was tested using the image named *imagejpg.jpg*, and the results can be seen in figure 10. The result was later evaluated using rawpixels.net and it is shown in figure 11.

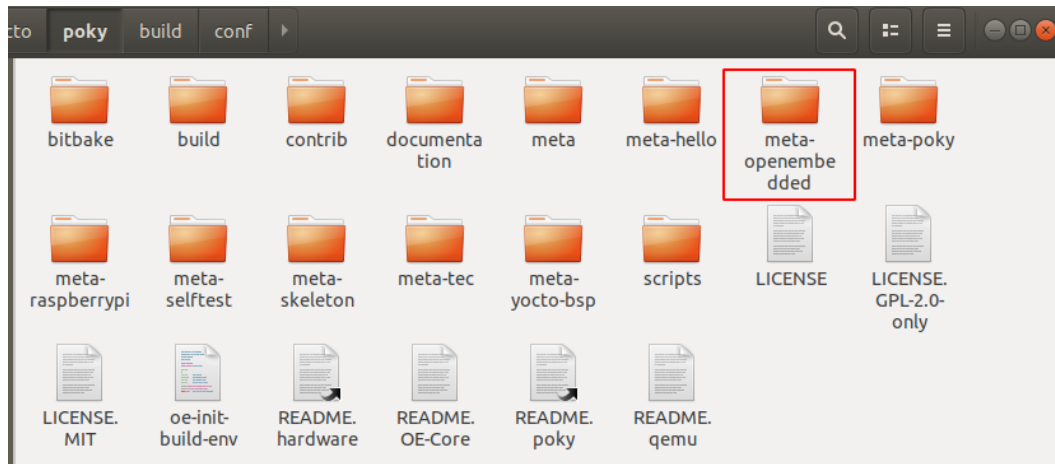


Figure 5: Open-Embedded meta layer including opencv

```
BBLAYERS ?= " \
/home/project2/Yocto/poky/meta \
/home/project2/Yocto/poky/meta-poky \
/home/project2/Yocto/poky/meta-yocto-bsp \
/home/project2/Yocto/poky/meta-raspberrypi \
/home/project2/Yocto/poky/meta-hello \
/home/project2/Yocto/poky/meta-tec \
/home/project2/Yocto/poky/meta-openembedded/meta-oe \
"
```

Figure 6: Open-Embedded meta layer included in bblayers configuration file

```
IMAGE_INSTALL_append = " libgomp libgomp-dev libgomp-staticdev glibc-staticdev"
CORE_IMAGE_EXTRA_INSTALL += "opencv libopencv-core-dev libopencv-imgproc-dev opencv-dev |"
SERIAL_CONSOLES = "115200;ttyAMA0"
```

Plain Text ▾ Tab Width: 8 ▾

Figure 7: Local Yocto configuration file including OpenCV

4 C Application for RGB to YUV Transformation

We implemented a C application that performs a direct conversion of an input binary RGB image onto a binary YUV image. The code uses *getopt* to receive the input and output file as arguments. The actual conversion is performed on a function called *rgb2yuv* whose source code is shown below. The function receives a pointer to the buffer containing the input image and an already allocated buffer for the output image, as well as the number of bytes to process. It then iterates through the image pixel by pixel, which means that in each iteration the bytes counter is increased by three. Y, U and V are calculated and saved to the output image.

```
// Function to convert RGB to YUV
void rgb2yuv(unsigned char *input_image, unsigned char *output_image,
             uint32_t total_bytes){
    // Prepare variables
    uint32_t bytes_counter = 0;
    uint8_t R, G, B;
    uint16_t Y_tmp, U_tmp, V_tmp;
    // Run through bytes in RGB image. They are increased by 3 every iteration
    for(bytes_counter=0; bytes_counter<total_bytes; bytes_counter += 3){
        // Extract R, G and B
        R = input_image[bytes_counter];
        G = input_image[bytes_counter + 1];
```



```

project2@project2-VirtualBox: ~/Yocto/poky/build
File Edit View Search Terminal Help
DISTRO = "poky"
DISTRO_VERSION = "3.0.3"
TUNE_FEATURES = "aarch64 cortexa72 crc crypto"
TARGET_FPU = ""
meta
meta-poky
meta-yocto-bsp = "zeus:ca9dd4b8eab400f736a4f522b1383c21bf47351a"
meta-raspberrypi = "zeus:0e05098853eea77032bff9cf81955679edd2f35d"
meta-hello
meta-tec = "zeus:ca9dd4b8eab400f736a4f522b1383c21bf47351a"
meta-oe = "zeus:9e60d30669a2ad0598e9abf0cd15ee06b523986b"

Initialising tasks: 100% [#####] Time: 0:00:05
Sstate summary: Wanted 348 Found 0 Missed 348 Current 1451 (0% match, 80% complete)
NOTE: Executing Tasks
NOTE: Setscene tasks completed
NOTE: Tasks Summary: Attempted 5154 tasks of which 4105 didn't need to be rerun and all succeeded.

Summary: There were 4 WARNING messages shown.
project2@project2-VirtualBox:~/Yocto/poky/build$

```

Figure 8: Local Yocto configuration file including OpenCV

```

B = input_image[bytes_counter + 2];
// Calculate Y
Y_tmp = ((66*R + 129*G + 25*B) + 128) >> 8;
output_image[bytes_counter] = (uint8_t) CLIP_uint16(Y_tmp + 16);
// Calculate U
U_tmp = ((-38*R - 74*G + 112*B) + 128) >> 8;
output_image[bytes_counter + 1] = (uint8_t) CLIP_int16(U_tmp + 128);
// Calculate V
V_tmp = ((112*R - 94*G - 18*B) + 128) >> 8;
output_image[bytes_counter + 2] = (uint8_t) CLIP_int16(V_tmp + 128);
}
}

```

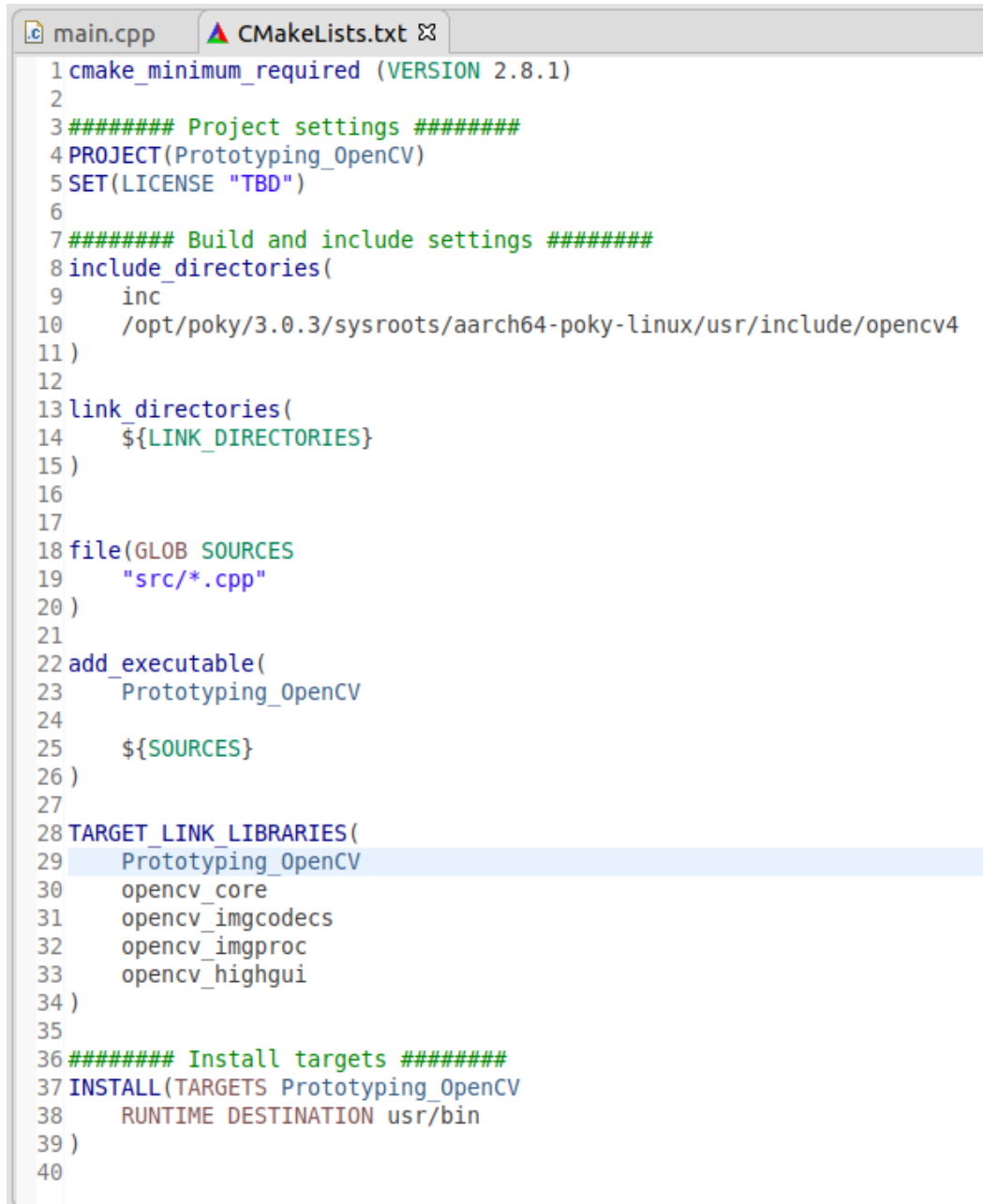
We use the `clock()` function to benchmark the CPU time used by the `rgb2yuv` function and `gettimeofday` to obtain the wall time elapsed while it runs. We will continue using this approach to benchmark code presented in latter sections.

```

// Capture times before running function
gettimeofday(&t_wall_start, 0);
t_clock_start = clock();
// Do conversion
rgb2yuv(input_image, output_image, bytes_read);
// Capture times after running and print results
t_clock_end = clock();
gettimeofday(&t_wall_end, 0);

```

We compile our code using an `O2` level of optimization. The result obtained after running our C code can be observed on figure 12, where we identify that the time taken by our baseline conversion function is of approximately `10.26ms`. Finally, we use `rawpixels.net` to validate our output image has the expected format, which can be observed on figure 13.



```

1 cmake_minimum_required (VERSION 2.8.1)
2
3 ##### Project settings #####
4 PROJECT(Prototyping_OpenCV)
5 SET(LICENSE "TBD")
6
7 ##### Build and include settings #####
8 include_directories(
9     inc
10    /opt/poky/3.0.3/sysroots/aarch64-poky-linux/usr/include/opencv4
11 )
12
13 link_directories(
14     ${LINK_DIRECTORIES}
15 )
16
17
18 file(GLOB SOURCES
19     "src/*.cpp"
20 )
21
22 add_executable(
23     Prototyping_OpenCV
24
25     ${SOURCES}
26 )
27
28 TARGET_LINK_LIBRARIES(
29     Prototyping_OpenCV
30     opencv_core
31     opencv_imgcodecs
32     opencv_imgproc
33     opencv_highgui
34 )
35
36 ##### Install targets #####
37 INSTALL(TARGETS Prototyping_OpenCV
38     RUNTIME DESTINATION usr/bin
39 )
40

```

Figure 9: CMake List file modification in Eclipse to compile RGB to YUV using OpenCV

4.1 Scheduling Priority Test

We run some tests with our baseline application by modifying the niceness (to increase its priority) and by changing the real-time scheduling attributes of our process. To set our process with the minimum niceness (and then maximum priority) we run it with the next command:

```
$ nice --20 ./rgb2yuv -i image.rgb -o YUV.c.yuv
```

To change the scheduling attributes with *chrt*, we run our process with the next command:


```
taylorcespedes@taylorcespedes-desktop: ~
root@raspberrypi4-64:~# ./Prototyping_OpenCV
Loading original RGB image ...
Image loaded. Height 480. Width 640
Converting image to YUV ..
Saving converted image to new file ...
File imageyuv.yuv written with 921600 bytes
root@raspberrypi4-64:~#
```

Figure 10: Results of running RGB to YUV conversion function using OpenCV in RPI4

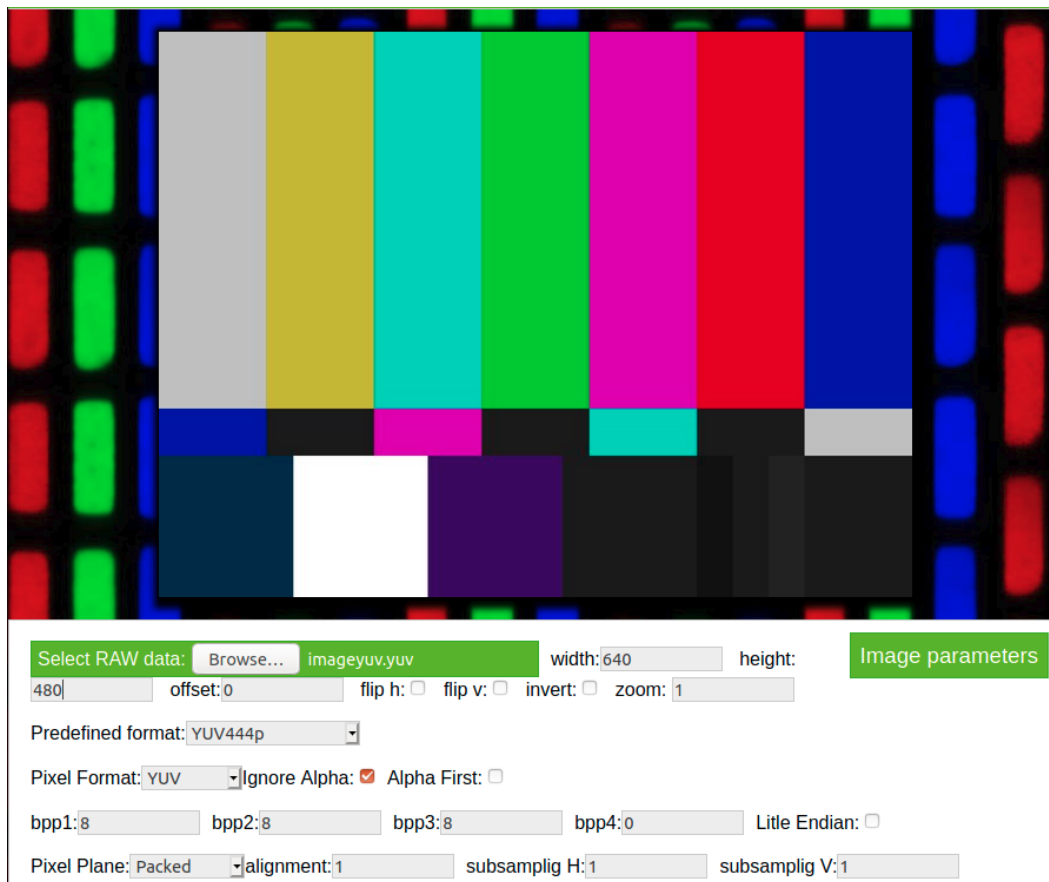


Figure 11: Evaluating results from figure 10 in rawpixels.net

```
$ chrt -f 99 ./rgb2yuv -i image.rgb -o YUV.c.yuv
```

The results obtained are presented on table 2. The results presented represent the average of running on each mode five consecutive times. As it can be observed, there was not a significant difference with changing the priority settings for our process versus running normally. The main reason for this is that most of the time our program is actually not competing with other tasks to get access to the CPU resources. Thus, increasing priority does not yield significantly faster execution time for a situation like this one.

```
root@raspberrypi4-64:~# ./rgb2yuv-c -i image.rgb -o YUV_c.yuv
>> Loading binary RGB file 'image.rgb'
Bytes read from file 921600

>> Converting RGB to YUV
Clock ticks spent 10252 (0.010252 seconds).
Wall Time Elapsed in 10268 us

>> Saving converted image to new file
File 'YUV_c.yuv' written with 921600 bytes
root@raspberrypi4-64:~#
```

Figure 12: Running the C Application rgb2yuc-c

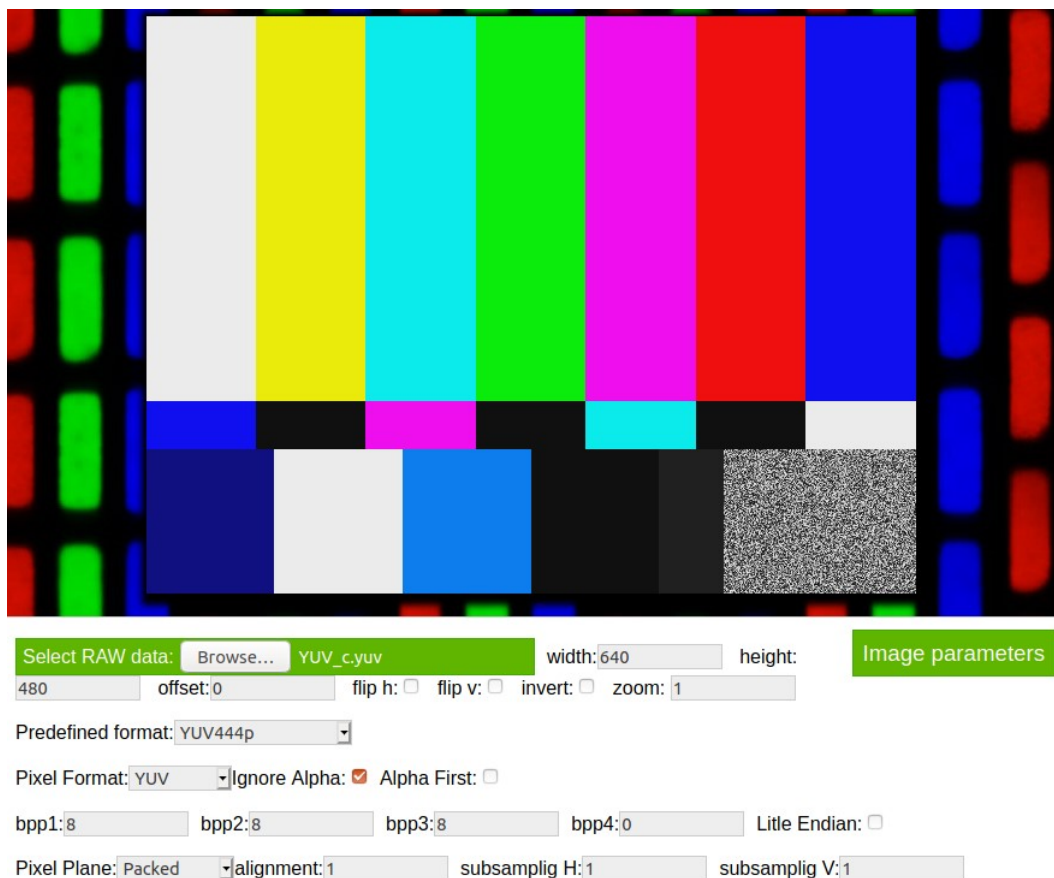


Figure 13: Validating Output Image of rgb2yuc-c

4.2 Disabling Kernel Frequency Scaling Test

We disable the kernel frequency scaling by modifying the `cpu0/cpufreq/scaling_min_freq` to 1500000 instead of its default value of 600000. When running our application on the CPU core 0 we obtain the results observed on table 3. We can observe how this change brought a significant improvement in our execution time with a **speedup** of 2.36. This is an expected result since now we are guaranteeing that the cores will always be running at the maximum

Table 2: Scheduling Priority Tests Results

Priority Policy	Average Wall Time (μs)	Average Clock Ticks
Standard	10152	10105
Minimum Niceness	10234	10175
RT Attributes	10147	10130

frequency, although with the drawback of having greater power consumption.

Table 3: Kernel Frequency Scaling Test Results

Scaling Min Freq (MHz)	Average Wall Time (μs)	Average Clock Ticks	Speedup
600	10152	10105	1.00
1500	4293	4289	2.36

4.3 Compiler Optimizations

We experiment with different compiler optimizations and evaluate the impact on the execution time of our conversion function. The results are presented on table 4, where again the average represents the result of running five consecutive times on each mode. Surprisingly, the best results were obtained when compiling with *O1* and *Os* (slightly better for *Os*). The *O2* and greater seem to be including something that actually degrades performance around 10%. More investigation would need to be done on the optimizations done and potentially on the generated assembly code to determine the root cause.

Table 4: Compiler Optimization Tests Results

Compiler Optimization	Average Wall Time (μs)	Average Clock Ticks
O0	35740	35613
O1	9189	9160
Os	9115	9089
O2	10152	10105
O3	10189	10159
Ofast	10357	10327

5 Profiling Application with perf

The *rgb2yuv-c* and the *Prototyping_OpenCV* applications were profiled using the *perf* utility installed in the RPI4 as it can be seen in figures 14 and 15. We want to start this section by discussed the proposed questions below.

Proposed Questions

- What are the performance bottlenecks of your application? Can you identify the critical functions in your code?

Based on the flame graph generated for the *rgb2yuv-c* code in figure 17, it can be seen that the code the biggest bottle neck is the function *rgb2yuv*. After investigating the rest of the functions it was seen that they belong to generic Linux libraries, which are commonly outside of our control using a high level programming language like C.

- How does the profiling data of your own implementation differ from the prototype implementation? Are bottlenecks somewhere else?

The *Prototyping_OpenCV* code have two bottlenecks the functions named *_dl_map_object* and *_do_lookup_x* which are part of the GNU Library (glibc) based on [5] and [6].

```
taylorcespedes@taylorcespedes-desktop: ~
root@raspberrypi4-64:~# perf record -F 10000 -a -g -- ./rgb2yuv -i image.rgb -o
imageyuv2.yuv
>> Loading binary RGB file 'image.rgb'
Bytes read from file 921600

>> Converting RGB to YUV
Clock ticks spent 11862 (0.011862 seconds).
Wall Time Elapsed in 11905 us

>> Saving converted image to new file
File 'imageyuv2.yuv' written with 921600 bytes
[ perf record: Woken up 2 times to write data ]
[ perf record: Captured and wrote 0.658 MB perf.data (3718 samples) ]
root@raspberrypi4-64:~# ls -alh
total 29M
drwx----- 3 root root 4.0K Jul 29 2020 .
drwxr-xr-x 3 root root 4.0K Jul 27 08:19 ..
-rw----- 1 root root 513 Jul 28 2020 .bash_history
drwxr-xr-x 9 root root 4.0K Jul 28 2020 .debug
-rwxr-xr-x 2 root root 584K Jul 27 2020 Prototyping_OpenCV
-rw-r--r-- 1 root root 900K Jul 29 2020 image.rgb
-rw-r--r-- 1 root root 14K Jul 28 2020 imagejpg.jpg
----r-xr-x 1 root root 900K Jul 29 2020 imageyuv.yuv
-rw-r--r-- 1 root root 900K Jul 29 2020 imageyuv2.yuv
-rw----- 1 root root 678K Jul 29 2020 perf.data
-rw----- 1 root root 4.2M Jul 29 2020 perf.data.old
-rw-r--r-- 1 root root 21M Jul 29 2020 perf.script
-rwxr-xr-x 2 root root 40K Jul 29 2020 rgb2yuv
root@raspberrypi4-64:~# perf script > perf2.script
```

Figure 14: Profiling the C Application rgb2yuc-c with perf

```
root@raspberrypi4-64:~# perf record -F 10000 -a -g -- ./Prototyping_OpenCV
Loading original RGB image ...
Image loaded. Height 480. Width 640
Converting image to YUV ..
Saving converted image to new file ...
File imageyuv.yuv written with 921600 bytes
[ perf record: Woken up 10 times to write data ]
Warning:
Processed 25661 events and lost 4 chunks!

Check IO/CPU overload!

[ perf record: Captured and wrote 3.945 MB perf.data (25133 samples) ]
root@raspberrypi4-64:~#
```

Figure 15: Profiling the C++ Application Prototyping_OpenCV with perf

- Design a draft strategy to optimize your application based on your profiling data. What would you do and how to speed up your code?

The *rgb2yuv-c* code can be easily parallelized since each one of the groups of three pixels of the image are converted individually. By the time the covert function is called, the image is already loaded into memory and consequentially the execution time can be accelerated by having multiple loops indexing and processing groups of pixels in parallel. In order for this approach to work is important to have each of these loops running on an individual thread. Also, it is a good idea to divide the image of chunks of the same size, since we want to keep the same number of material operations in each one of the loops/threads.

Having multiple threads will require to have a mechanism that allow each of these threads to convey after all the data is processed, so it can be used to build the new yuv image. This logic comparison between the existing (a) and the proposed methods (b) can be seen in figure ??.

- Generate .svg profiling data of your prototype and low-level implementations and deep analysis of your profiling information and draft of your optimization strategy.

The flame graph on figure 17 was build with the results from figure 14 for the *rgb2yuv-c*code and the *Prototyping_OpenCV* code respectively.

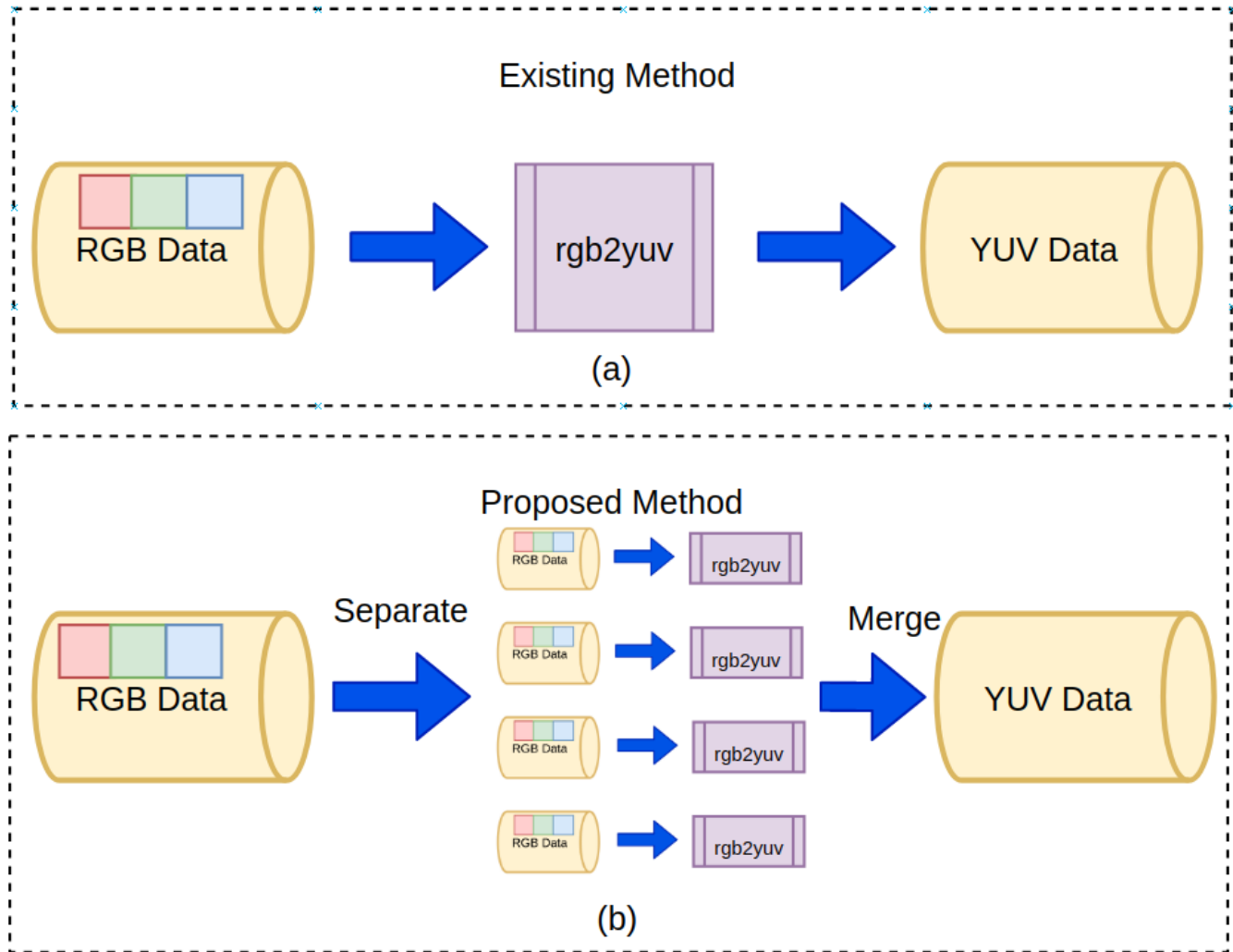


Figure 16: (A) Existing C Application *rgb2yuc-c* method vs (b) proposed one

6 Application Optimizations

In this section, we present three different approaches intended to optimize the execution time of our baseline *rgb2yuv-c* application. The main results are summarized on table 5 where the average represents the result of running each technique five consecutive times. We can observe how *OpenMP* provided the fastest execution, providing a speedup

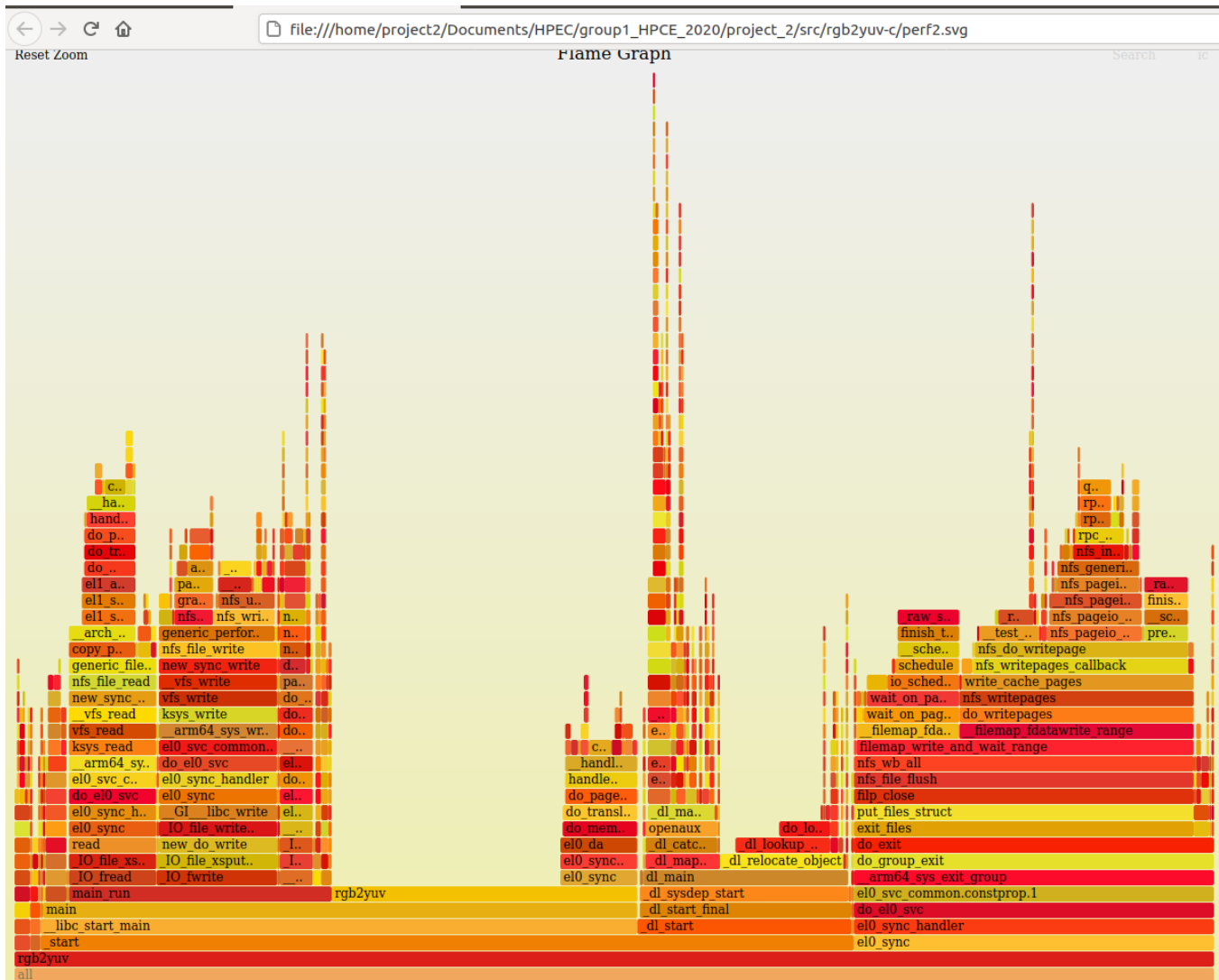


Figure 17: Frame graph of the C Application rgb2yuc-c

over the baseline of 3.05. *Neon Intrinsics* and *Pthreads* also provided a significant improvement of 2.52 and 2.67 respectively. We can observe how performance does not necessarily increase linearly when adding more processing elements that run in parallel. This is likely the case because there are implicit overheads on the different mechanisms used to enable the parallelism. This overhead is more relevant as the amount of work to be done by each processing unit decreases. We will present more details on each technique on the next sections.

6.1 Application Optimization Using Neon Intrinsics

We start by taking advantage of the Neon unit on the ARM processors of the RPI4 by using the vectorial intrinsics instructions. Since the Neon unit has *128-bit* registers, we decide to process the image with the next logic in each iteration of our *for-loop*:

- De-interleave R, G and B into three separate registers of *128-bits* each. Since each pixel uses 1-byte for each color, we are loading 16 pixels per iteration. The instruction used to load these registers is shown below:

```
// Extract R, G and B onto triple register. Each register has 16, 8-bits registers
```

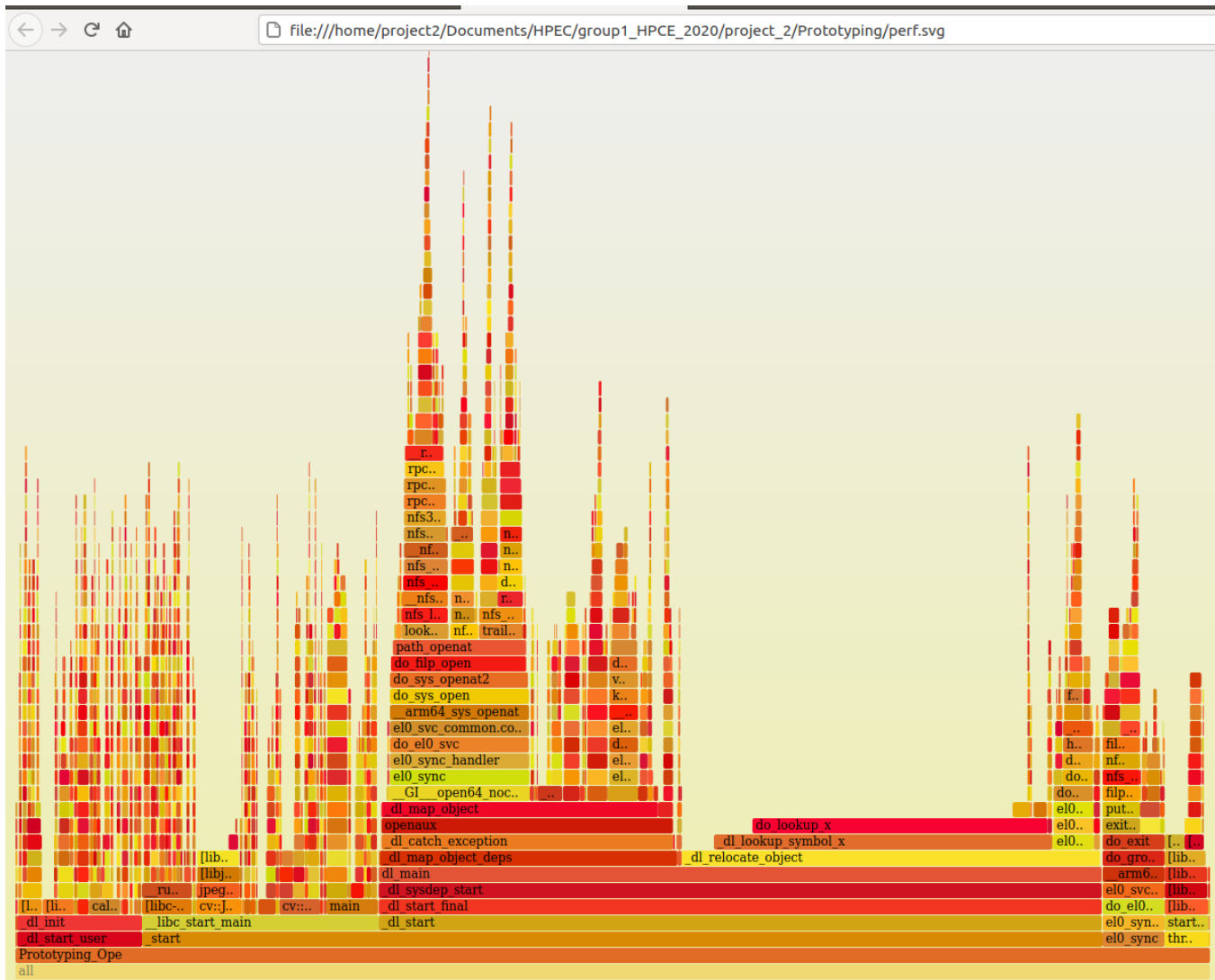



Figure 18: Flame graph from results in figure 15

```
uint8x16x3_t rgb = vld3q_u8(input_image + bytes_count);
```

- The variable *bytes_count* counts the number of bytes processed. It increases 48 bytes per iteration, which is the equivalent of 16 pixels.
- We split each of the R, G and B registers into two registers of *64-bits*. This is necessary since they will be multiplied by other registers, with the result being extended to registers of *16-bits* to avoid overflows.

```
// We are going to process each color in two chunks.
uint8x8_t R_high_u = vget_high_u8(rgb.val[0]);
uint8x8_t R_low_u = vget_low_u8(rgb.val[0]);
uint8x8_t G_high_u = vget_high_u8(rgb.val[1]);
uint8x8_t G_low_u = vget_low_u8(rgb.val[1]);
uint8x8_t B_high_u = vget_high_u8(rgb.val[2]);
uint8x8_t B_low_u = vget_low_u8(rgb.val[2]);
```

Table 5: Application Optimization Results Comparison

Optimization Technique	Average Wall Time (μs)	Speedup
Baseline	10152	1.00
Neon Intrinsics	4036	2.52
Pthreads	3798	2.67
OpenMP	3330	3.05

- Then, we apply the required mathematical manipulations to obtain the desired result. The code below exemplifies the required calculations to obtain the Y component of the output image.

```
// ***** Calculating Y *****
//Y_tmp = ((66*R + 129*G + 25*B) + 128) >> 8;
//output_image[bytes_counter] = (uint8_t) CLIP_uint16(Y_tmp + 16);
// Perform multiplications by coefficients and accumulate for each of the 2 chunks
tmp_high_u = vmull_u8(R_high_u, vdup_n_u8(66));
tmp_low_u = vmull_u8(R_low_u, vdup_n_u8(66));

tmp_high_u = vmlal_u8(tmp_high_u, G_high_u, vdup_n_u8(129));
tmp_low_u = vmlal_u8(tmp_low_u, G_low_u, vdup_n_u8(129));

tmp_high_u = vmlal_u8(tmp_high_u, B_high_u, vdup_n_u8(25));
tmp_low_u = vmlal_u8(tmp_low_u, B_low_u, vdup_n_u8(25));

// Add 128 constant
tmp_high_u = vaddq_u16(tmp_high_u, vdupq_n_u16(128));
tmp_low_u = vaddq_u16(tmp_low_u, vdupq_n_u16(128));

// Now shift 8 bits to the right
tmp_high_u = vshrq_n_u16(tmp_high_u, 8);
tmp_low_u = vshrq_n_u16(tmp_low_u, 8);

// Add 16 constant
tmp_high_u = vaddq_u16(tmp_high_u, vdupq_n_u16(16));
tmp_low_u = vaddq_u16(tmp_low_u, vdupq_n_u16(16));
```

- Since we now have two separate registers storing 16-bit values, they are both saturated back to the 8-bits range and combined again to build the $16, 8\text{-bits}$ results for each output. For example for Y :

```
// Now saturate and convert back to 16, 8-bits registers which is final result for Y
yuv.val[0] = vcombine_u8(vqmovn_u16(tmp_low_u), vqmovn_u16(tmp_high_u));
```

- Similar calculations are applied for U and V (although they used signed operations since some of their coefficients are negative). Finally, the triple yuv register is interleaved and saved to the output image buffer with the next instruction:

```
// Store yuv results interleaved to output image
vst3q_u8(output_image + bytes_count, yuv);
```

This optimized applications using Neon intrinsics is compiled with the next flags:

```
$ -O2 -mtune=cortex-a72 -ftree-vectorize -DNDEBUG
```

As it was presented on table 5, this technique enabled a **speedup** of 2.52 over the baseline implementation.

6.2 Application Optimization Using Pthreads

We use *POSIX Threads (pthreads)* to attempt to distribute the load of our application among the four CPU-cores available on the RPI4. Considering this, we divide the image to process into four blocks of equal size and attempt to have four workers running in parallel. To achieve this, we launch three asynchronous *pthreads* to process the first three blocks and process the last one on the main process thread. The code described is shown below:

```
// Function to convert RGB to YUV
void rgb2yuv(unsigned char *input_image, unsigned char *output_image, uint32_t total_bytes){
    // Since we will have 4 workers, each of them has 1/4 bytes yo process
    uint32_t bytes_per_thread = total_bytes >> 2;
    pthread_t thread1, thread2, thread3;

    // Launch first thread
    struct pthread_data pt1_data;
    pt1_data.bytes_to_process = bytes_per_thread;
    pt1_data.input_image = input_image;
    pt1_data.output_image = output_image;
    pthread_create(&thread1, NULL, rgb2yuv_async_thread, (void*) &pt1_data);

    // Launch second thread with offset on data to process
    struct pthread_data pt2_data;
    pt2_data.bytes_to_process = bytes_per_thread;
    pt2_data.input_image = pt1_data.input_image + bytes_per_thread;
    pt2_data.output_image = pt1_data.output_image + bytes_per_thread;
    pthread_create(&thread2, NULL, rgb2yuv_async_thread, (void*) &pt2_data);

    // Launch third thread with offset on data to process
    struct pthread_data pt3_data;
    pt3_data.bytes_to_process = bytes_per_thread;
    pt3_data.input_image = pt2_data.input_image + bytes_per_thread;
    pt3_data.output_image = pt2_data.output_image + bytes_per_thread;
    pthread_create(&thread3, NULL, rgb2yuv_async_thread, (void*) &pt3_data);

    // Fourth thread is actually current thread
    rgb2yuv_main_thread(pt3_data.input_image + bytes_per_thread, pt3_data.output_image +
        bytes_per_thread, bytes_per_thread);

    // Wait for async threads to return
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);
}
```

The function invoked by each of the *pthreads* is shown below:

```
// Separate thread worker for rgb to yuv conversion
void* rgb2yuv_async_thread(void * data_ptr){
    // Convert input generic ptr to our pthread data pointer
    struct pthread_data * pt_data = (struct pthread_data *)data_ptr;

    // Prepare variables
    uint32_t bytes_counter = 0;
    uint8_t R, G, B;
    uint16_t Y_tmp;
    int16_t U_tmp, V_tmp;
    // Run through bytes in RGB image. They are increased by 3 every iteration
    for(bytes_counter=0; bytes_counter < pt_data->bytes_to_process; bytes_counter += 3){
        // Extract R, G and B
        R = pt_data->input_image[bytes_counter];
        G = pt_data->input_image[bytes_counter + 1];
        B = pt_data->input_image[bytes_counter + 2];
        // Calculate Y
        Y_tmp = ((66*R + 129*G + 25*B) + 128) >> 8;
        pt_data->output_image[bytes_counter] = (uint8_t) CLIP_uint16(Y_tmp + 16);
        // Calculate U
        U_tmp = ((-38*R - 74*G + 112*B) + 128) >> 8;
```

```
pt_data->output_image[bytes_counter + 1] = (uint8_t) CLIP_int16(U_tmp + 128);
// Calculate V
V_tmp = ((112*R - 94*G - 18*B) + 128) >> 8;
pt_data->output_image[bytes_counter + 2] = (uint8_t) CLIP_int16(V_tmp + 128);
}
return NULL;
}
```

This optimized applications using *pthread*s is compiled with the next flags:

```
$ -O2 -mcpu=cortex-a72+crc+crypto -DNDEBUG
```

As it was presented on table 5, this technique enabled a **speedup** of 2.67 over the baseline implementation.

6.3 Application Optimization Using OpenMP

Similarly as it was done with *pthread*s, we attempt to balance the load of our program among the four CPU-cores available on the system. However, in this case we use a more automated approach based on *OpenMP*. The code stays very simple since we let the tool determine the mechanism to parallelize our loop by using the *pragma omp parallel for* statement. We also use the function *omp_set_num_threads* to configure the number of threads to use based on an argument provided by the user. The code using this approach is shown below:

```
// Function to convert RGB to YUV
void rgb2yuv(unsigned char *input_image, unsigned char *output_image, uint32_t total_bytes,
             uint16_t threads_number){
// Prepare variables
uint32_t bytes_counter = 0;
uint8_t R, G, B;
uint16_t Y_tmp;
int16_t U_tmp, V_tmp;
// Make for loop parallel with openmp pragma
omp_set_num_threads(threads_number);
#pragma omp parallel for private(R, G, B, Y_tmp, U_tmp, V_tmp)
// Run through bytes in RGB image. They are increased by 3 every iteration
for(bytes_counter=0; bytes_counter<total_bytes; bytes_counter += 3){
// Extract R, G and B
R = input_image[bytes_counter];
G = input_image[bytes_counter + 1];
B = input_image[bytes_counter + 2];
// Calculate Y
Y_tmp = ((66*R + 129*G + 25*B) + 128) >> 8;
output_image[bytes_counter] = (uint8_t) CLIP_uint16(Y_tmp + 16);
// Calculate U
U_tmp = ((-38*R - 74*G + 112*B) + 128) >> 8;
output_image[bytes_counter + 1] = (uint8_t) CLIP_int16(U_tmp + 128);
// Calculate V
V_tmp = ((112*R - 94*G - 18*B) + 128) >> 8;
output_image[bytes_counter + 2] = (uint8_t) CLIP_int16(V_tmp + 128);
}
}
```

This optimized applications using *openmp* is compiled with the next flags:

```
$ -O2 -fopenmp -mcpu=cortex-a72+crc+crypto -DNDEBUG
```

As it was presented on table 5, this technique enabled a **speedup** of 3.05 over the baseline implementation, which represents the best result obtained at this point.

7 Combining Optimizations

With the desire of further increasing performance, we proceed to combine some of the techniques reviewed on the previous sections. We implement two different experiments described below:

- Experiment 1: *Neon intrinsics* + disabled frequency scaling + *OpenMP*.
- Experiment 2: *Neon intrinsics* + disabled frequency scaling + *pthread*s

The obtained results are presented on table 6. We can observe how these combinations yield a speedup close to a factor of 8. In this case the best result is obtained when using *pthread*s for load distribution among cores.

Table 6: Combining Optimizations Results Comparison

Experiment	Average Wall Time (μs)	Speedup
Baseline	10152	1.00
Experiment 1	1307	7.76
Experiment 2	1282	7.92

References

- [1] S. Gal-On and M. Levy, *Exploring coremarkTM – a benchmark maximizing simplicity and efficacy*, Online; accessed 12-July-2020. [Online]. Available: <https://www.eembc.org/techlit/articles/coremark-whitepaper.pdf>.
- [2] EEMBC, *Coremark git-hub readme page*, Online; accessed 12-July-2020. [Online]. Available: <https://github.com/eembc/coremark>.
- [3] —, *Coremark-pro: An eembc benchmark*, Online; accessed 12-July-2020. [Online]. Available: <https://www.eembc.org/coremark-pro/>.
- [4] —, *Coremark=pro git-hub readme page*, Online; accessed 13-July-2020. [Online]. Available: <https://github.com/eembc/coremark-pro>.
- [5] GoogleGit, *Dl-load.c*, Online; accessed 1-August-2020. [Online]. Available: https://chromium.googlesource.com/chromiumos/third_party/glibc/+/cvs/libc-960927/elf/dl-load.c.
- [6] E. Cormier, *What is dllookup_{symbol}? – c + +profiling*, Online; accessed 1-August-2020. [Online]. Available: <https://stackoverflow.com/questions/11768919/what-is-dl-lookup-symbol-x-c-profiling>.