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

Nombre: Ing. Luis Carlos Alvarez Mata
Carne: 200902144
Nombre: Ing. Verny Moreles Soto
Carne: 201116212

1. Respuesta de preguntas

1.1. Describir brevemente los siguientes términos sobre teoría de operación de los algoritmos de benchmark

Básicamente CoreMark es un benchmark que mide el rendimiento de los microcontroladores (MCUs) y la unidad central de procesamiento (CPU) en sistemas embebidos. Entre os algoritmos que utiliza CoreMark se tienen: list procesing (busca y clasificar), matrix manipulation (operaciones comunes de matrices), state machine (determina si un conjunto de entradas posee números válidos), y CRC (cyclic redundacy check por sus siglas en inglés). Está diseñado para correr en dispositivos de 8 a 64 bits. A continuación se da una mejor explicación de alguno de los algoritmos de benchmark.

- Linked List: en esta sección del benchmark se ejecuta una lista con una estructura pre-definida la cual se carga en memoria. Generalmente listas anidadas utilizan mallocs para hacer sus llamados, en sistemas embebidos generalmente son las aplicaciones las que controlan la memoria para llamados a pequeñas estructuras como arreglos o listas, por eso se considera un caso realista. La lista andidada será inicializada de tal manera que un cuarto de los punteros de la lista se asignan a áreas secuenciales de memorias, mientras que el resto se distribuyen de manera no secuencial. Una vez hecho esto, se comenzarán a ejecutar múltiples operaciones, la lista sera ordenada basado en un data16, y posteriormente derivará el valor de CRC el cual se convertirá en parte de la cadena de salida. Esta lista volverá a ser ordenada basada en el valor idx. Esto garantiza que la lista se retorne al estado inicial antes de dejar la función, asegurandose así que se llegue a los mismos resultados.
- Matrix Multiply: este algoritmo forma parte de la base de muchos otros algoritmos más complejos. Consta de un ciclo interno el cual se intenta optimizar al máximo. Cuenta con tres matrices de tamaño NxN (A, B, C). La matriz A tendrá valores pequeños, matriz B tendrá valores medianos, y matriz C se utiliza para los resultados. Para el benchmark se comienza multiplicando la matriz A por una constante y se guarda en C, posteriormente se multiplica la matriz A por una columna en específico de la matriz B y se guarda en C. Por último, se multiplica toda la matriz A por B, y se guarda en C.
- State Machine: esta parte del algoritmo se utiliza para probar casos de switch y condicionales. En efecto, se utiliza una máquina de estados de moore. Esta máquina va a identificar entradas de tipo strings como números y las va a clasificar acorde a su tipo. Este algoritmo realizará tareas realistas. Básicamente, se va a hacer un llamado de la máquina de estado en todas las entradas y va a recolectar estados finales y de transición. Por otra parte, modificara algunos estados adrede, para intentar colectar estos cambios, y después devolverá al sistema a su estado normal.

1.2. ¿Cúal es la diferencia entre coreportme y archivos core, se pueden modificar?

El archivo coreportme si se puede modificar, se puede cambiar como se implementa, cambiar variables de configuración y varios métodos de implementación.

1.3. Diferencias entre CoreMark y CoreMark-PRO

1. ¿Cómo difiere el algoritmo CoreMark-PRO de su original? ¿Qué se mejoró? Descripción breve de sus nuevos Workloads.
   Entre las principales diferencias radica que el nuevo algoritmo cuenta con cinco benchmarks nuevos que integran operaciones de enteros y números flotantes. Entre las pruebas que se incluye se tiene descompresión de imágenes y transformadas de Fourier FFT, entre otros. Entre otras mejores se tiene que CoreMark-PRO cuenta con mayor datos por contexto para cada algoritmo, 3MB en total, tiene como objetivo tanto la memoria del procesador como el de los subsistemas, y posee soporte multicore expandido.

2. ¿CoreMark se encuentra contenido en CoreMark-PRO?
   Sí.

3. ¿Cómo se encuentran combinados los workloads para resumir los resultados en un único Score?
   Se debe entender primero que los benchmarks contenidos en Coremark Pro se basan dos parámetros principales, contexto y procesadores, y su ejecución respectiva en el kernel. Es la combinación de estos tres factores que al final va a definir y consolidar un resultado basado en su desempéño individual, es decir se promedia el resultado de cada uno de los resultados. Una vez finalizados los resultados se colocan en un directorio temporal, cada uno consiste en una verificación simple seguido de 3 corridas de rendimiento. Básicamente se concatenn los mejores resultados de cada log.

1.4. Corriendo CoreMark-Pro en la Raspberry Pi 4

1.4.1. Propuesta escogido para compilar y correr los algoritmos

Para la compilación del binario de coremark-pro, y la generación del build y sus respectivos workloads, se escoge copiar todos los archivos del benchmark a la Raspberry pi 4 y realizar la compilación nativa. La principal razón por la que se escogió de esta manera es que el enfoque del trabajo se encuentra en el análisis de los resultados y realizar análisis sensitivos de algunos parámetros posibles de cambiar. Por otro lado, realizar una cross compilación resulta mucho más tedioso y laborioso, puesto que hay que buscar los parámetros correctos, banderas, y compiladores necesarios para su ejecución

1.4.2. Qué parámetros se pueden modificar en CoreMark-Pro

Entre los parámetros que se pueden modificar se tiene el número de iteraciones, el TOOLCHAIN, lso builds, que pueden ser de diversos destinos dependiendo de si se desea hacer una compilación cruzada o no. El porteo de los archivos es otra cosa que se puede cambiar, y esto es algo que va de la mano con la compilacón cruzada. También se pueden modificar los Makefile para una específica perfilación de los workloads, como por ejemplo la cantidad de contextos y cantidad de núcleos a utilizar para correr el benchmark.

Prueba práctica modificando parámetros medidos desde la Raspberri pi 4

Para la prueba de parámetros se van a utilizar los siguientes escenarios, Multi Mark en donde únicamente se modifica la cantidad de contextos que se utilizan mientras que los núcleos se mantienen en 1. Se va a correr el Parallel Mark, en donde la cantidad de núcleos varía y el contexto se mantiene fijo en 1. Por último se corrió Mix Mark, el posibilita modificar tanto el contexto como la cantidad de núcleos. A continuación se muestran los resultados:

2. Resultados

2.1. Ejecución de los benchmark de CoreMark en la Raspberry Pi + Análisis

1. A continuación se va a explicar el estudio al cambiar la variable DMULTITHREAD. Se identifica primeramente que la cantidad máxima de threads que puede manejar la Raspberrypi son 4. Por esto el análisis se realiza de 1 a 4 threads, con incrementos unitarios. A continuación se presenta el comportamiento de los resultados vs cantidad de threads. Como se puede obsevar en la figura 1 al aumentar los números de threads, se mejora el Score de manera lineal. Se puede deducir además que el incremento es proporcional al aumento de threads.

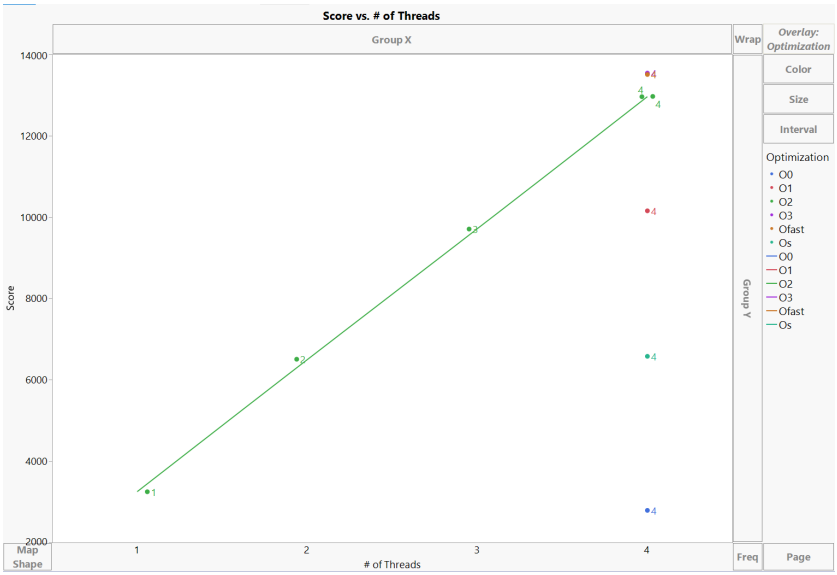

Figura 1: Experimento 1. Análisis sensitivo de cantidad de threads vs Score

Según lo muestran estos resultados, si pudieramos incrementar de esta manaera los threads, y manteniendo el punto de optimización fijo, el socre seguirá aumentando conforme aumenten los hilos.De la misma manera se hace una gráfica dándole vuelta a los parámatros, se utiliza la misma optimización disminuyendo el número de hilos, es muy interesante observar como en el punto donde se utilizan 2 y un 1 hilo, se obtienen tiempos de ejecución menores, pero esto lo que ejemplicfica es que esta optimización esta enfocada a mas de 2 threads activos, por lo que estas corridas no se consideran de fiar. Es por esto que los siguientes estudios se realizan con los 4 threads fijos.

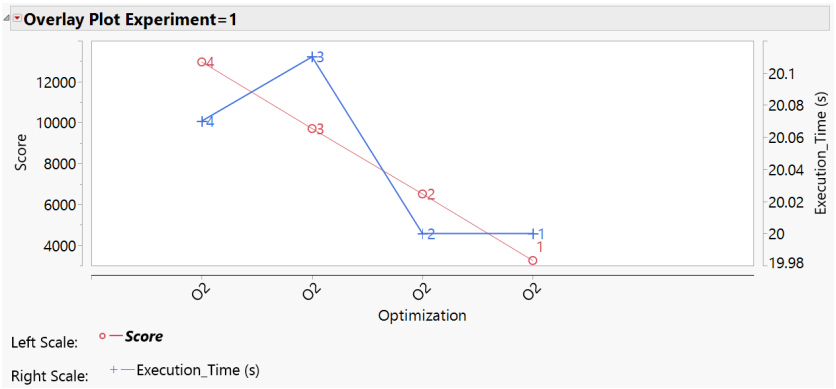

Figura 2: Experimento 1. Análisis sensitivo de Cantidad de procesadores.

2. Es importante entender el rol que tiene el calendarizador de Linux en un ejemplo como este. Básicamente el calendariza sus tareas basados en prioridades estáticas y dinámicas. Cada vez que el calendarizador es llamado, hace un checkeo de todas las tareas en cola y las analiza para calcular el goodness value. La tarea con el valor más alto se va a ejecutar, este valor se calcula combinando las prioridades estáticas como dinámicas.

3. El siguiente experimento realizado fue un estudio en la optimización de como ejecutar Coremark. Primero se debe entender que sin ningún parámetro de optimización no se podrá reducir el costo computacional para compilar y poder debuggear de manera correcta los resultados. Dependiendo de la optimización escogida así va

a durar más el benchmark, por ejemplo -0 -01, toman mucho más tiempo en ejecutarse ya que utilizan mucho más espacio de memoria para ejecutar sus funciones. -02 ejecuta casi que todos los tipos de optimización en todas sus funciones, por lo que se espera que su ejecución sea mucho más corto. Basados en la teoría se espera que la optimización -03, -0fast, -02 presenten los mejores resultados.
Como se ovbserva en la figura 3, se realiza una gráfica donde se muestra como varia el tiempo de ejecución así como el resultado del benchmark dependiendo de la optimización escogida. Se puede observar que la mejor optimización para las raspberrypy4, fue la opción -0fast. Recordemos que esta optimización habilita tanto optimizaciones que se encuentran en -03, así como otras que no siempre son compatibles con todos los sistemas, es por esto que estas dos se asemejan mucho. Era de esperarse que -04 tuvieras los peores resultados ya que utiliza muchos recursos de memoria ya que no cuenta muchas banderas de optimización activas. Es posible mejorar Coremark por medio del compilador, como se muestra enlos resultados el score es muy sensitivo a cualquier cambio en banderas de optimización, por lo que para algunos casos si se puede considerar como un benchmark independiente del compilador.

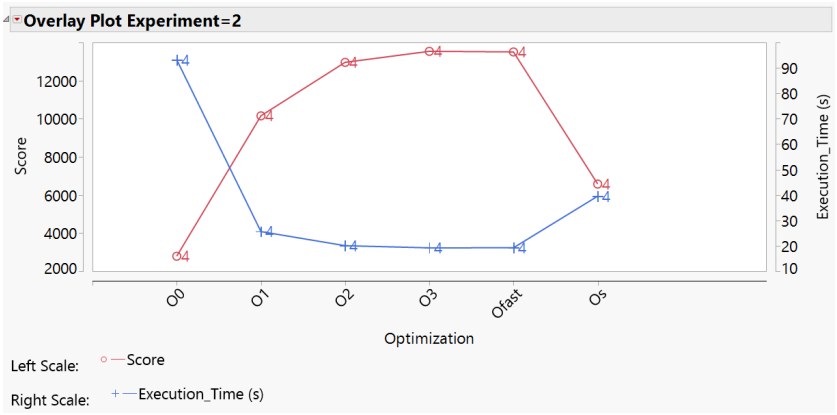

Figura 3: Experimento 2. Distintos tipo de optimizaciones CoreMark.

4. Se utiliza ahora de nuestro procesador BCM2711 con los resultados obtenidos de la página de CoreMark. Los resultados acá muestran un resultado de 33100, mientras que nuestro mejor resultado fue de 13000, esto utilizando las mismas banderas de ejecución. Sin embargo, existe un parámetro que no fue utilizado por nuestro experimento, y es el Fork, el cual habilita la ejecución en paralelo, debido a esto es que notamos esa diferencia entre los resultados, ya que las velocidades de procesador y capacidad de memoria son las mismas.

2.2. Ejecución de Coremark-Pro en la RaspberryPi4 + Análisis

a) Cómo ya se exploró en la sección anterior, se observa como Coremar Pro es sensitivo al número de contextos como a la cantidad de workers utilizados, es por esto que se toman esas variables para hacer el análisis sensitivo. Se corren los 3 principales escenarios del benchmark como lo son Multi Mark, Parallel Mark, y Mix Mark. En la figura 4 se puede observar como se dan variaciones, además se dividen los resultados en un multicore y un single core. Para el escenario de Mix Mark, es en el que se da una mayor variabilidad, lo cual es esperado ya que se modican tanto el contexto como los workers. En este caso, obtenemos el mejor resultado al tener C4 y W4. Al intentar colocar C5, se observa como el resultado es 0, esto es debido a que si el contexto es mayor al número de workers el benchmark va a fallar. Ahora, para el escenario Multi Mark, en donde únicamente se modifica el contexo y se mantiene W1, se observa como a mayor cantidad de contexto se obtiene un mayor score, lo esperado ya que se ejecutan más funciones en paralelo en el kernel. Se intentó correr C4 pero el procesador no lo permitió, al igual que en el caso anterior. Para parallel mark no se obtuvo mucha variación al modificar la cantidad de workers y manteniendo un único contexto.

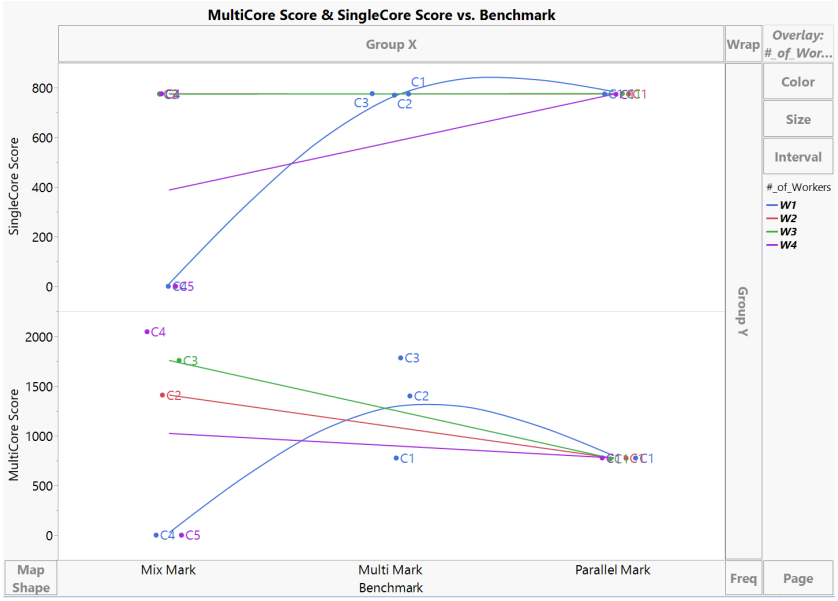

Figura 4: Estudio experimental de de contextos vs de workers.

2.2.1. Prototipo de la transformación de color RGB/YUV con OpenCV

Se crea un prototipo utilizando OpenCV en el lenguaje *C++*, con el cual utiliza bibliotecas de alto nivel dándonos una abstracción del problema y haciendo el desarrollo fácil y rápido. Para esto se utiliza la función *cvtColor* la cual realiza la transformación de RGB a YUV por medio del parámetro *COLOR_BGR2YUV*. La fracción del código donde se aprecia la implementación de esta función se puede ver a continuación:

```
void rgb2yuv (char *input_image, char *output_image){

   Mat matRGB, matYUV;

   printf("Opening the image..\n");
   matRGB = imread(input_image, IMREAD_COLOR);

   printf("Converting RGB to YUV..\n");
   cvtColor(matRGB, matYUV, COLOR_BGR2YUV);

   saveYUV(matYUV, output_image);
}
```

2.2.2. Implementación de la transformación de color RGB/YUV con C

Para la implementación en *C* es necesario realizar operaciones matemáticas donde se extraen de cada pixel los componentes R, G y B por medio de corrimientos y se operan contra coeficientes definidos por medio de sumas, restas y multiplicación para obtener los componentes Y, U y V. . A continuación se presenta un fragmento del código donde se realizo esto:

```
int rgb2yuvPixel (int R, int G, int B){

   int Y, V, U;
   unsigned int pixel32;
   unsigned char *pixel = (unsigned char *)&pixel32;
```

```
   Y = (0.257 * R) + (0.504 * G) + (0.098 * B) + 16;
   V =  (0.439 * R) - (0.368 * G) - (0.071 * B) + 128;
   U = -(0.148 * R) - (0.291 * G) + (0.439 * B) + 128;

   pixel[0] = Y;
      pixel[1] = U;
   pixel[2] = V;
      pixel[3] = 0;
   return pixel32;
}

void rgb2yuv(char *input_image, char *output_image){
    ...
   size = IMAGE_WIDTH*IMAGE_HEIGHT;
   for(i=0; i<size; i++){
      fread(&pixelRGB, 3, 1, in);
      R  = ((pixelRGB & 0x000000ff));
      G  = ((pixelRGB & 0x0000ff00)>>8);
      B  = ((pixelRGB & 0x00ff0000)>>16);

      pixel32 = rgb2yuvPixel(R, G, B);
      pixelYUV[0] = (pixel32 & 0x000000ff);
      pixelYUV[1] = (pixel32 & 0x0000ff00) >> 8;
      pixelYUV[2] = (pixel32 & 0x00ff0000) >> 16;

      fwrite(pixelYUV, 3, 1, out);
   }
   fclose(in);
   fclose(out);
}
```

2.3. Implementación en C con NEON Intrinsics

Para este escenario se realizó una implementación utilizando las instrucciones *NEON Instrinsics* donde re maximizó el rendimiento para la tecnología de datos múltiples de instrucción única (SIMD, por sus siglas en ingles), la cual es una extensión de la arquitectura de los procesadores ARM Cortex-A y Cortex-R. Para realizar esto se utilizan los enlaces y ejemplos proveídos por el profesor y el instructivo donde se utilizaron instrucciones básicas para esta tecnología. Se puede observar en el extracto del código a continuación, que la abstracción para esta implementación es mucho menor y la complejidad del código se incremento considerablemente al punto de tener que analizar todo el set de instrucciones disponible. El extracto de código muestra cómo se obtienen las componentes R, G y B y como se genera la componente Y.

```
void rgb2yuv(char *input_image, char *output_image){
    ...
   //Loading the buffer
   fread(buffer_in,totalBytes,1,in);

   //go over the image every 16 pixels
   for(int i = 0; i<totalPixels/16; i++)

      //loading 16 pixels (R, G, B components)
      rgb_x3x16_tmp = vld3q_u8(rgb_ptr+i*16*3);

      //Dividing the data of the vector for each color component
```

```
    //Red
    uint8x8_t high_r = vget_high_u8(rgb_x3x16_tmp.val[2]); //Duplicate vector element to vector
    uint8x8_t low_r = vget_low_u8(rgb_x3x16_tmp.val[2]); //Duplicate vector element to vector
    int16x8_t signed_high_r = vreinterpretq_s16_u16(vaddl_u8(high_r, vdup_n_u8(0))); //Vector
        reinterpret cast operation
    int16x8_t signed_low_r = vreinterpretq_s16_u16(vaddl_u8(low_r, vdup_n_u8(0))); //Vector reinterpret
        cast operation

    //Green
    uint8x8_t high_g = vget_high_u8(rgb_x3x16_tmp.val[1]); //Duplicate vector element to vector
    uint8x8_t low_g = vget_low_u8(rgb_x3x16_tmp.val[1]); //Duplicate vector element to vector
    int16x8_t signed_high_g = vreinterpretq_s16_u16(vaddl_u8(high_g, vdup_n_u8(0))); //Vector
        reinterpret cast operation
    int16x8_t signed_low_g = vreinterpretq_s16_u16(vaddl_u8(low_g, vdup_n_u8(0))); //Vector reinterpret
        cast operation

    //Blue
    uint8x8_t high_b = vget_high_u8(rgb_x3x16_tmp.val[0]); //Duplicate vector element to vector
    uint8x8_t low_b = vget_low_u8(rgb_x3x16_tmp.val[0]); //Duplicate vector element to vector
    int16x8_t signed_high_b = vreinterpretq_s16_u16(vaddl_u8(high_b, vdup_n_u8(0))); //Vector
        reinterpret cast operation
    int16x8_t signed_low_b = vreinterpretq_s16_u16(vaddl_u8(low_b, vdup_n_u8(0))); //Vector reinterpret
        cast operation

    //Y component
    //Duplicate vector element to scalar
    uint8x8_t coef1 = vdup_n_u8(66);
      uint8x8_t coef2 = vdup_n_u8(129);
      uint8x8_t coef3 = vdup_n_u8(25);

    uint16x8_t high_y = vmull_u8(high_r, coef1); //Unsigned Multiply long
    high_y = vmlal_u8(high_y, high_g, coef2); //Unsigned Multiply-Add Long
    high_y = vmlal_u8(high_y, high_b, coef3); //Unsigned Multiply-Add Long
    uint16x8_t low_y = vmull_u8(low_r, coef1); //Unsigned Multiply long
    low_y = vmlal_u8(low_y, low_g, coef2); //Unsigned Multiply-Add Long
    low_y = vmlal_u8(low_y, low_b, coef3); //Unsigned Multiply-Add Long
    high_y = vaddq_u16(high_y, vdupq_n_u16(128)); //Add
    low_y = vaddq_u16(low_y, vdupq_n_u16(128)); //Add
    //Join two smaller vectors into a single larger vector
    //Unsigned saturating shift right narrow
    uint8x16_t y = vcombine_u8(vqshrn_n_u16(low_y, 8), vqshrn_n_u16(high_y, 8));
    yuv_x3x16_tmp.val[0] = y; //-> Component Y

    ...

    //Joining the 3 components
    vst3q_u8(yuv_ptr, yuv_x3x16_tmp); //Store multiple 3-element structures from three registers
    yuv_ptr += 3*16; //Move to the next block of registers
  }
  //write the file YUV
   fwrite(buffer_out,totalBytes,1,write_ptr);

  //Close files
  fclose(in);
  fclose(out);
}
```

2.4. Optimizando nuestra implementación en C con Multi-Threading

Conversión RGB-YUV usando Pthread

Tomando en cuenta que el Raspberry Pi 4 tiene 4 núcleos disponibles, se optimizó el código para que pudiera realizar una ejecución en paralelo de 4 hilos. Donde el recorrido de los pixeles se dividió en 4 sectores dando a cada hilo un sector a ejecutar. Al crear cada hilo, este recibe un parámetro el cual le indica que sector de la imagen va a ejecutar, así realizando como $\frac{1}{4}$ del recorrido en comparación con la implementación original.

```
void *rgb2yuvPixel(void *pOption){
   ...
   if(*option == 1){
      limitU = 0;
      limitL = size;
   }else if(*option == 2){
      limitU = size;
      limitL = size*2;
   }else if(*option == 3){
      limitU = size*2;
      limitL = size*3;
   }else if(*option == 4){
      limitU = size*3;
      limitL = size*4;
   }

   for(i=limitU; i<limitL; i++){

      R  = ((pixelRGB[i] & 0x000000ff));
      G  = ((pixelRGB[i] & 0x0000ff00)>>8);
      B  = ((pixelRGB[i] & 0x00ff0000)>>16);

      Y = (0.257 * R) + (0.504 * G) + (0.098 * B) + 16;
      V =  (0.439 * R) - (0.368 * G) - (0.071 * B) + 128;
      U = -(0.148 * R) - (0.291 * G) + (0.439 * B) + 128;

      pixel[0] = Y;
      pixel[1] = U;
      pixel[2] = V;
      pixel[3] = 0;

      pixelYUV[i][0] = (pixel32 & 0x000000ff);
      pixelYUV[i][1] = (pixel32 & 0x0000ff00) >> 8;
      pixelYUV[i][2] = (pixel32 & 0x00ff0000) >> 16;
   }
}

void rgb2yuv (char *input_image, char *output_image){
    ...
   for(i=0; i<size; i++){
      fread(&pixelRGB[i], 3, 1, in);
   }

   iret1 = pthread_create( &thread1, NULL, rgb2yuvPixel, &option1);
    if(iret1) {
       fprintf(stderr,"Error - pthread_create() return code: %d\n",iret1);
       exit(EXIT_FAILURE);
    }
```

```
    iret2 = pthread_create( &thread2, NULL, rgb2yuvPixel, &option2);
    if(iret2) {
       fprintf(stderr,"Error - pthread_create() return code: %d\n",iret2);
       exit(EXIT_FAILURE);
    }

    iret3 = pthread_create( &thread3, NULL, rgb2yuvPixel, &option3);
    if(iret3) {
       fprintf(stderr,"Error - pthread_create() return code: %d\n",iret3);
       exit(EXIT_FAILURE);
    }

    iret4 = pthread_create( &thread4, NULL, rgb2yuvPixel, &option4);
    if(iret4) {
       fprintf(stderr,"Error - pthread_create() return code: %d\n",iret4);
       exit(EXIT_FAILURE);
    }

     pthread_join(thread1, NULL);
     pthread_join(thread2, NULL);
     pthread_join(thread3, NULL);
     pthread_join(thread4, NULL);
   ...
}
```

Conversión RGB-YUV usando OpenMP

OpenMP es un API que se puede utilizar para dirigir explícitamente el paralelismo de memoria compartida multiproceso. Para ello, se agregaron las directivas en el bucle principal, donde se recorre la imagen. Se le indico cuales variables serian compartidas y cuales, privadas, además de indicarla el segmento especifico del bucle.

Código 1: case1.c

```
int rgb2yuvPixel (int R, int G, int B){
    ...
   Y = (0.257 * R) + (0.504 * G) + (0.098 * B) + 16;
   V =  (0.439 * R) - (0.368 * G) - (0.071 * B) + 128;
   U = -(0.148 * R) - (0.291 * G) + (0.439 * B) + 128;

   pixel[0] = Y;
      pixel[1] = U;
   pixel[2] = V;
      pixel[3] = 0;
   return pixel32;
}

void rgb2yuv (char *input_image, char *output_image){
    ...
   #pragma omp parallel shared(pixelYUV, pixelRGB, size) private(i, R, G, B, pixel32)
   {
      #pragma omp for
      for(i=0; i<size; i++){

         R  = ((pixelRGB[i] & 0x000000ff));
         G  = ((pixelRGB[i] & 0x0000ff00)>>8);
         B  = ((pixelRGB[i] & 0x00ff0000)>>16);

         pixel32 = rgb2yuvPixel(R, G, B);
```

```
        pixelYUV[i][0] = (pixel32 & 0x000000ff);
        pixelYUV[i][1] = (pixel32 & 0x0000ff00) >> 8;
        pixelYUV[i][2] = (pixel32 & 0x00ff0000) >> 16;

      }
    }
    ...
}
```

2.4.1. Comparación de los resultados RGB2YUV .C vs openmp vs pthread

Para comoparar el comportamiento de los distintos cófigos se hace un gráfico comparativo en la figura 5. Se pueden observar los casos de estudio, el RGB2YUV el cual es un código en C sin nigún tipo de optimización ni paralelismo, por otro lado se utilizan dos tipos de paralelismo, uno con la función de openmp, y otro por medio de hilos. Como se puede observar el tiempo de ejecución (eje y) disminuye conforme se cambian los escenarios. Se realizan tres corridas por cada escenario, y se busca su promedio para encontrar un valor que se estadísticamente creible y fiable. Como era de esperarse el código sin ninguna optimizacón es el que más tarda de convertir la imagen RGB a YUV 0.062s, mientras que el código con la implementación de los pthreads es el que mejor se comportó, convirtiendo las imágenes en un tiempo promedio de 0.04s.

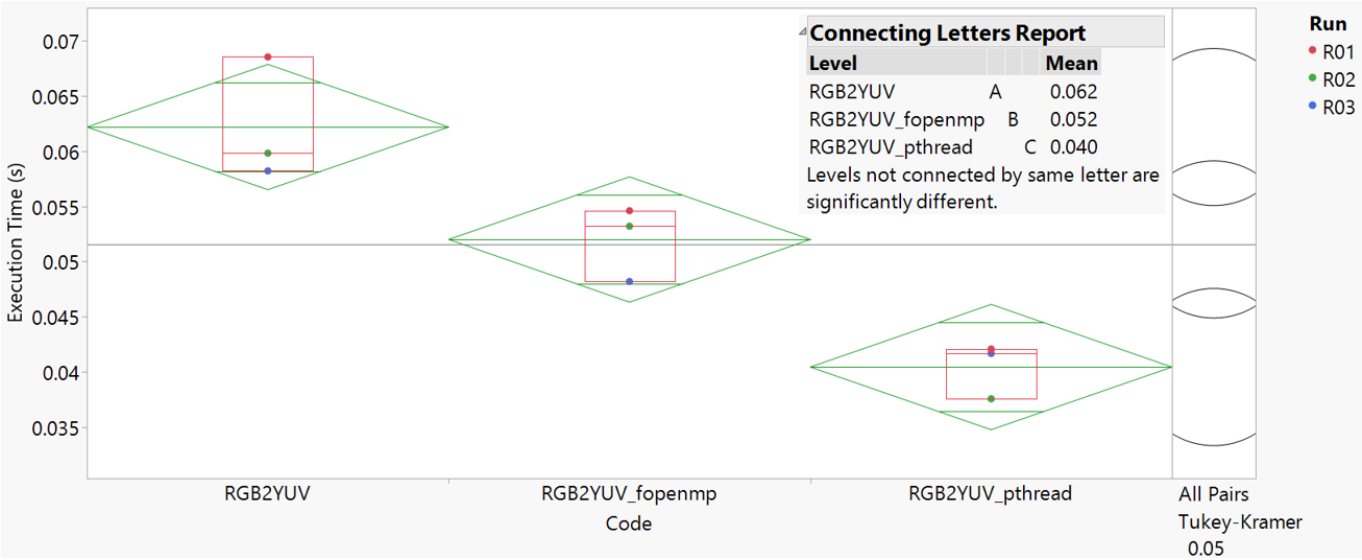

Figura 5: Comparación estadística de la conversión de la imagen RGB a YUV para cada caso de estudio.

Referencias

[1]B. Gregg, brendangregg/FlameGraph. 2020.
[2]B. Cornianu, "Change swap size in Ubuntu 18.04 or newer," Bogdan Cornianu, Aug. 11, 2018. https://bogdancornianu.com/c swap-size-in-ubuntu/ (accessed Aug. 02, 2020).
[3]B. Cornianu, "Change swap size in Ubuntu 18.04 or newer," Bogdan Cornianu, Aug. 11, 2018. https://bogdancornianu.com/c swap-size-in-ubuntu/ (accessed Aug. 02, 2020).
[4]"clock - C++ Reference." https://www.cplusplus.com/reference/ctime/clock/ (accessed Aug. 02, 2020).
[5]E. M. B. Consortium, eembc/coremark. 2020.

[6]E. M. B. Consortium, eembc/coremark-pro. 2020.
[7]"Getopt (The GNU C Library)." https://www.gnu.org/software/libc/manual/html$_n$ode/Getopt.htmlGetopt(accessedAug,0
[8]"How to Compile and Run An OpenMP Program." https://www.dartmouth.edu/ rc/classes/intro$_o$penmp/compile$_r$un.html
[9]"IHI0073A$_a$rm$_n$eon$_i$ntrinsics$_r$ef.pdf." Accessed : Aug,02, 2020.[Online].Available : http : //www.ie.tec.ac.cr/sarriola/H1
[10]"Linux Tutorial: POSIX Threads." http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html (accessed Aug. 02, 2020).
[11]"Maximizing SIMD performance - Raspberry Pi Forums." https://www.raspberrypi.org/forums/viewtopic.php?p=1538193 (accessed Aug. 02, 2020).
[12]"OpenCV: Install OpenCV-Python in Ubuntu." https://docs.opencv.org/master/d2/de6/tutorial$_p$y$_s$etup$_i$n$_u$buntu.html(ac
[13]"Optimize Options (Using the GNU Compiler Collection (GCC))." https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html (accessed Aug. 02, 2020).
[14]"RAW Pixels." http://rawpixels.net/ (accessed Aug. 02, 2020).
[15]A. Ltd, "SIMD ISAs — Introducing Neon for Armv8-A," Arm Developer. https://developer.arm.com/architectures/instruct sets/simd-isas/neon/neon-programmers-guide-for-armv8-a/introducing-neon-for-armv8-a (accessed Aug. 02, 2020).
[16]A. Ltd, "SIMD ISAs — Neon Intrinsics Reference," Arm Developer. https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics (accessed Aug. 02, 2020).
[17]"Using OpenMP with C — Research Computing University of Colorado Boulder documentation." https://curc.readthedocs. C.html (accessed Aug. 02, 2020).
[18]"Using your C compiler to exploit NEON," p. 13.
[19]"Using your C compiler to exploit NEON.pdf." Accessed: Aug. 02, 2020. [Online]. Available: https://www.doulos.com/know
[20]"Yocto Project Profiling and Tracing Manual." https://www.yoctoproject.org/docs/2.1/profile-manual/profile-manual.html (accessed Aug. 02, 2020).
[21]"YUV," Wikipedia. Jun. 17, 2020, Accessed: Aug. 02, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title
[22]"YUV to RGB Conversion." http://www.fourcc.org/fccyvrgb.php (accessed Aug. 02, 2020).