Artículo de Investigación de Multiprocesadores

Filtro Sepia Aplicado a Imágenes

Mariana Pérez García (A01206747)
Tecnológico de Monterrey, Campus Querétaro
A01206747@itesm.mx

3 de diciembre de 2019

RESUMEN

Este documento tiene la finalidad de analizar y mostrar el desempeño de distintas tecnologías de multiprocesamiento, entre ellas Fork-Join, Threads, TBB y CUDA. El problema utilizado para poder comparar el tiempo de procesamiento de estas herramientas es la aplicación de un filtro color sepia a una imagen 4K cuyas dimensiones son de 1080 x 1920 y su peso es de 1.6MB.

1. INTRODUCCIÓN

Actualmente un programa que realiza una sola tarea no es considerado eficiente, un programa eficiente es aquel que permite ejecutar múltiples tareas dentro de sí mismo. Un sistema multiprocesador es aquel que tiene dos o más unidades de procesamiento central también conocidas como CPUs, estas unidades comparten todo el acceso a una memoria RAM común. El objetivo de los sistemas de multiprocesamiento es mejorar el rendimiento en cuanto a tiempo de ejecución [1]. Existen cuatro diferentes tipos de flujo de instrucciones que describen a todas las computadoras, y estas se dividen en: Single Instruction, Single Data (SISD), en esta categoría se encuentra la gran mayoría de los computadores existentes. Son equipos con un solo procesador que trabaja sobre un solo dato a la vez, es decir trabajan de manera secuencial. Single Instruction, Multiple Data (SIMD), es un arreglo de procesadores, cada procesador sigue el mismo conjunto de instrucciones, es decir diferentes elementos de información son asignados a cada procesador y utilizan memoria distribuida. Multiple Instrucition, Multiple Data (MIMD), este tipo de arquitectura establece que los procesadores pueden ejecutar la misma instrucción o múltiples instrucciones simultáneamente, esta es la arquitectura empleada por las diferentes tecnologías utilizadas para desarrollar programas multiprocesos. Finalmente se encuentra la Multiple Instructions, Single Data, la cual no son usadas comúnmente ni tan significativas como las anteriores [2]. Existen formas de desarrollar programas que pueden ejecutar múltiples tareas, estas se dividen en dos categorías, paralelismo y concurrencia. Estos dos conceptos son muy usados actualmente y muchas veces se emplean para definir lo mismo, sin embargo, son conceptos muy diferentes. El paralelismo se refiere a varios procesos que se ejecutan al mismo tiempo y realizan la misma tarea, mientras que la concurrencia son procesos que existen al mismo tiempo y que pueden o no realizar la misma tarea [3].

El procesamiento de imágenes es un problema muy común utilizado en el área de multiprocesamiento ya que es necesario aplicar un kernel a una imagen en un proceso llamado convolución. Este proceso tiende a ser lento dependiendo del tamaño de la imagen y el número de imágenes que van a pasar por el filtro, sin embargo, se han desarrollado formas de disminuir el tiempo de procesamiento utilizando más de un núcleo en los procesadores, esto ayuda en varias áreas de la computación como lo es machine learning o Deep learning donde se necesitan millones de imágenes pre-procesadas para poder entrar a una red neuronal creando un cuello de botella que aumenta el tiempo de espera para entrenar el modelo. En el actual documento se mostrarán diferentes formas de aplicar un filtro sepia a una imagen y se analizarán los tiempos de procesamiento y el uso del CPU de cada una de las tecnologías empleadas.

2. DESAROLLO

La imagen utilizada en todas las aplicaciones es una imagen 4K de 1080 x 1920 que pesa 1.6MB. Todas las pruebas fueron realizadas 10 veces para poder sacar el tiempo promedio que el programa tarda en procesar la información.

Imagen 2.1 Imagen original (izquierda) y con el filtro sepia (derecha)

2.1 JAVA THREADS

Multithreading o multihilos es una forma de crear concurrencia en Java, esto permite la ejecución de uno o más hilos o procesos los cuales realizan una tarea en específico, puede ser la misma no. Este tipo de paradigma busca utilizar la mayor capacidad del CPU. Los hilos pueden ser creados por dos mecanismos, pueden ser extendidos de una clase pre-definida por Java llamada Thread,o pueden utilizar una interface llamada Runnable. La diferencia entre estos dos métodos es que si la clase creada se extiende de la clase Thread, ya que Java no permite múltiple herencia, solamente puede extenderse de la clase Thread pero no de otra clase padre, mientras que Runnable permite la herencia de una clase padre al mismo tiempo, sin embargo, la interface Runnable no tiene funciones definidas en la clase Thread como lo son los métodos yield() e interrupt() [4]. El código realizado para esta sección implementa la extensión de la clase Thread.

Algunas ventajas que permite el uso de multihilos es mejorar el rendimiento mediante la disminución del tiempo de desarrollo, simplificar el código, mejorar la responsividad en las interfaces gráficas, realizar diferentes tareas de manera simultánea, mejor uso del cache, y mejor uso de los recursos del CPU, sin embargo, como cualquier tecnología tiene sus desventajas las cuales son debuggeo y testing complejo, aumenta la posibilidad de un

deadlock, incrementa la dificultad al desarrollar el programa provocando que los resultados sean impredecibles debido a condiciones de carrera [5].

Forma de Desarrollo	Tiempo (ms)
Java Secuencial	31.10000
Java Threads	11.40000
Speedup	2.72807018

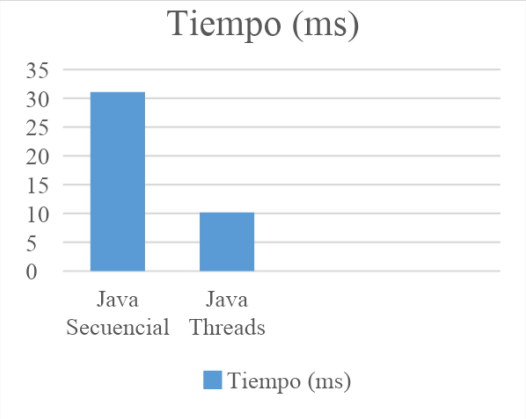


Figura 2.1.1. Tabla comparativa y representación visual entre java secuencial y el uso de threads junto con su speedup

El uso de threads en la aplicación de filtros permite una mejora de aproximadamente el 2.7% sobre el algoritmo secuencial.

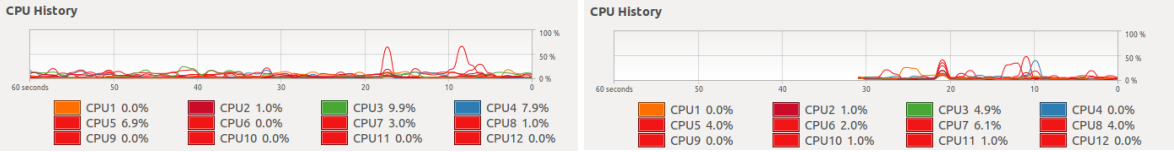


Figura 2.1.2 Uso del CPU en uso secuencial (izquierda) y usando Threads (derecha)

En cuanto al uso del procesador es posible observar que en la forma secuencial el uso de solo un núcleo aumentó, sin embargo, los demás núcleos se quedaron sin usar, cosa que no sucede en la versión de threads donde es posible que la mayoría de los núcleos fueron utilizados provocando un procesamiento más rápido.

2.2 JAVA FORK-JOIN

El framework fork-join fue presentado en Java7. Este framework proporciona herramientas que ayudan a acelerar los procesos en paralelo intentando utilizar todos los nucleos de los procesadores disponibles. La manera en la cual es implementado es mediante el uso del algoritmo Divide y Conquista. Es decir, esta tecnología lo primero que hace es hacer un “fork”, lo que significa romper la tarea recursivamente en tareas más pequeñas hasta que lo suficientemente sencillas para que puedan ser realizadas de manera asíncrona. Después se hace el “join” de los resultados, que es el esperar a que todas las sub-tareas se terminen de

ejecutar para unir el resultado. La principal diferencia entre el fork-join y la clase Thread o la interfaz Runnable es que fork-join utiliza un pool llamado ForkJoinPool el cual es un conjunto de hilos, y cada hilo tiene una fila de tareas por realizar, en caso de que un hilo este vacío y los demás hilos tengan todavía tareas pendientes por realizar el hilo sin tarea puede obtener una de las tareas en la fila para ejecutarla haciéndolo más eficiente que un thread normal. Este framework consiste en dos tipos de acciones, el RecursiveAction el cual es una acción que no regresa nada, y RecursiveTask, que es cuando se regresa algún valor al final de la ejecución. Para este caso se utilizó RecursiveAction ya que sólo se llenó la matriz resultante con respecto a la matriz original.

Forma de Desarrollo	Tiempo (ms)
Java Secuencial	31.10000
Java Fork-Join	10.90000
Speedup	2.85321101

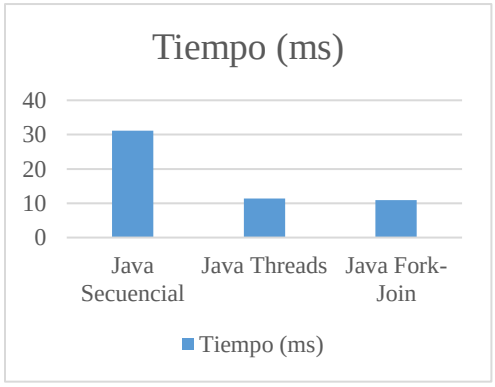


Figura 2.2.1 Tabla comparativa y representación visual entre el tiempo de procesamiento de la forma secuencial threads, y fork-join

El utilizar fork-join es posible obtener hasta un 2.8% de mejora sobre el algoritmo secuencial y un 1.04% de mejora sobre el uso de threads normales, esto es debido al algoritmo implementado en el fork-join de robo de trabajo.

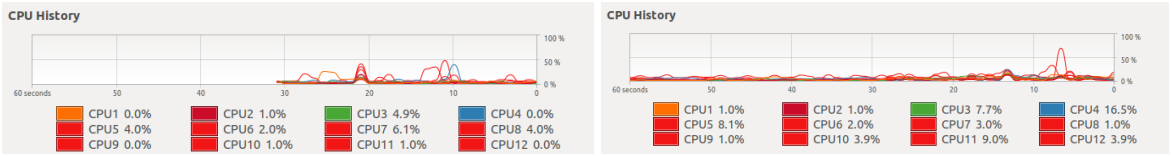


Figura 2.2.2 Uso de CPU utilizando threads (izquierda) y fork-join (derecha)

En cuanto al uso del procesador es posible observar que se hicieron varios hilos que trabajaron al mismo tiempo para procesar la imagen, en comparación con el uso del CPU en uso de threads también es posible notar que todos los núcleos tuvieron la misma carga, a diferencia de threads donde se puede observar que hubo núcleos que tuvieron más trabajo que otros, haciendo el fork-join más eficiente que el uso convencional de threads.

2.3 CUDA

En el 2003 un grupo de investigadores dirigidos por Ian Buck creó Brook, un programa extendido de C que permitía la programación en paralelo, después de un tiempo Buck se unió a Nvidia y en el 2006 lanzaron CUDA, la primera solución comercial cuyo propósito principal era mejorar el procesamiento utilizando GPUs. CUDA es un modelo desarrollado por la empresa de GPUs Nvidia, esta diseñado para un paradigma paralelo en donde todos los procesos creados trabajan en la misma tarea al mismo tiempo. CUDA permite desarrollar aplicaciones intensivas utilizando el poder de las tarjetas de video dedicadas. Actualmente CUDA es una de las herramientas más utilizadas debido a su alto nivel de procesamiento, principalmente en las áreas de computación financiera, modelado del clima, data science, Deep learning y machine learning, investigación, procesamiento de imágenes y gráficas computacionales, entre otras [7]. En el caso de procesamiento de imágenes, como es el usado en este reporte, CUDA utiliza un vector de enteros llamados dim3, que permite pasar el tamaño de la matriz, así como el bloque que esta siendo ejecutado por el kernel, esto permite que en una imagen se le asigne a cada thread creado por el bloque una cantidad específica de pixeles, provocando que el tiempo de procesamiento sea mucho menor que el de otras tecnologías.

Forma de Desarrollo	Tiempo (ms)
CUDA Secuencial (C)	104.82820
CUDA Paralizado	0.00520
Speedup	20159.2692

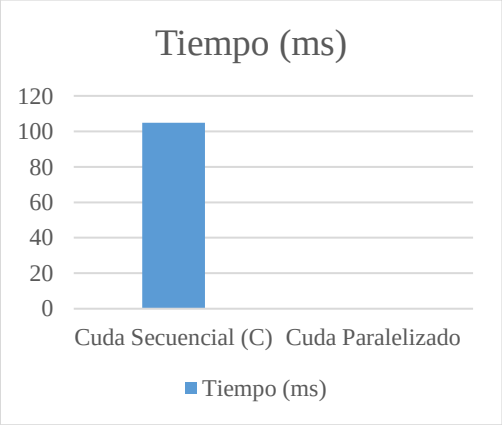


Figura 2.3.1 Tabla comparativa y representación visual entre el tiempo de procesamiento de la forma secuencial de CUDA (C) y la forma paralelizada

La forma secuencial del programa fue desarrolla utilizando C plano y funciones que se realizan en el CPU (Host), mientras que la forma paralelizada utiliza una tarjeta gráfica Nvidia GTX 1050 (device). La mejora del procesamiento en paralelo tiene un speedup de 20,159%, esto muestra la eficiencia del paradigma en paralelo.

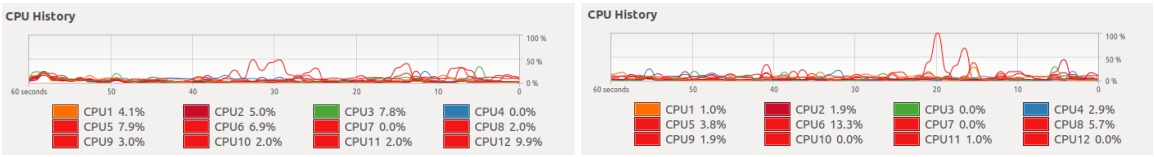


Figura 2.3.2 Uso del CPU en utilizando procesamiento secuencia (izquierda) y paralelización (derecha)

Con respecto al uso del CPU no es posible comparar como tal esta tecnología ya que no hace uso del CPU hasta que la información ya procesada es pasada al CPU o que haya previo algún tipo de pre-procesamiento en el CPU, pero todo el procesamiento es hecho en el GPU, por lo tanto, no es posible observar su modo de trabajo con el historial del CPU, es por eso que en la forma secuencial se puede observar el uso continuo de un núcleo, mientras que en la paralización el uso del CPU es mínimo.

2.4 TBB

Thread Building Blocks o TBB por sus siglas en inglés es una librería para C++ que permite programación en paralelo y memoria compartida, esta librería fue desarrolla por Intel y provee diferentes herramientas como lo son algoritmos genéricos de paralelización, contenedores concurrentes, asignadores escalables de memoria, robo de trabajo, y sincronización de bajo nivel de primitivos [8].

Forma de Desarrollo	Tiempo (ms)
C++ Secuencial	104.82820
C++ con TBB	13.21060
Speedup	7.93515813

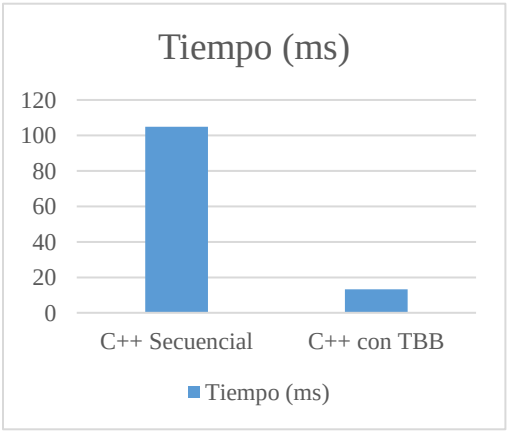


Figura 2.3.1 Tabla comparativa y representación visual entre el tiempo de procesamiento de la forma secuencial en C++ y la forma paralelizada usando TBB

La función implementada en TBB para el desarrollo de este código fue el método parallel_for, esta función es similar a la función RecuersiveAction de java en la cual no se regresa nada al final de la ejecución. TBB en comparación con el algoritmo secuencial desarrollado en C++ obtuvo un speedup de 7.93%.

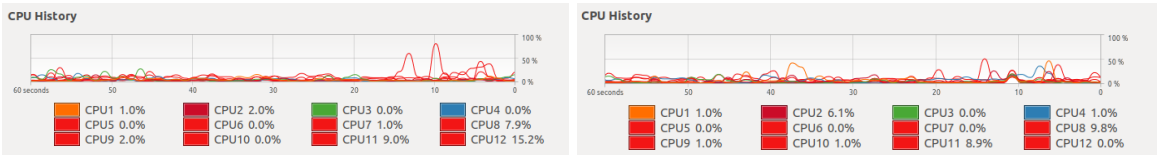


Figura 2.4.2 Uso del CPU en utilizando procesamiento secuencia (izquierda) y paralelización (derecha)

En cuanto al uso del procesador es posible observar que, en la ejecución en secuencia, solamente un núcleo realizó toda la tarea llegando a casi un 100% de uso y tardándose 104ms, mientras que en la forma paralelizada es posible observar cómo los núcleos trabajan con casi la misma carga provocando que la ejecución sea mas rápida.

2.5 OVERVIEW

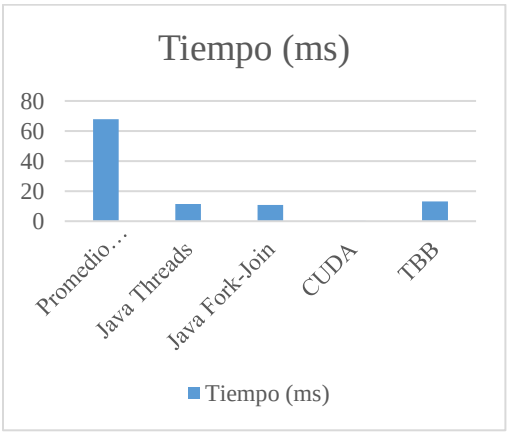


Figura 2.5.1 Representación gráfica del tiempo de procesamiento de las diferentes tecnologías

Con este análisis es posible observar que CUDA es la herramienta de procesamiento más rápido, esto es debido el GPU es un componente que se especializa en procesar rapidamente tareas específicas, como lo serían multiplicación de matrices o en este caso el procesamiento de imágenes. Después de CUDA sigue el procesamiento de java con fu framework fork-join, seguido del uso de threads. Fork-Join terminó con un mejor tiempo debido a su metodología de robo de trabajo en los threads, finalmente la tecnología TBB terminó con el tiempo más alto sin embargo no es mucha la diferencia que hay entre java e Intel.

Conclusiones

CUDA es la mejor herramienta para paralelizar programas, sin embargo, para poder utilizar esta herramienta es necesario tener una tarjeta de video dedicada de la marca Nvidia, cosa que no muchas computadoras tienen debido a su alto precio en el mercado, además de que aunque los GPUs son potentes no tienen la misma versatilidad que los CPUs, es decir, los GPUs no pueden hacer tareas concurrentes mientras que los CPUs pueden ejecutar ambas, además de que cualquier computadora tiene un CPU y puede tener acceso a las entradas y salidas de la computadora, es por eso que las aplicaciones que son usadas constantemente no son en paralelo, ya que es necesario realizar diferentes tareas al mismo tiempo, como lo es escribir y escuchar música al mismo tiempo, cosa que el paradigma en paralelo no permite. Es necesario identificar las funcionalidades del programa a desarrollar para poder encontrar el mejor paradigma para volverlo más eficiente y que funcione de manera correcta.

Referencias

[1] GeeksForGeeks. (s.f.). Introduction of Multiprocessor and Multicomputer. Recuperado de https://www.geeksforgeeks.org/introduction-of-multiprocessor-and-multicomputer/
[2] Galeon. (s.f.). SISD, SIMD, MISD, MIMD. Recuperado de http://rubmarin.galeon.com/sisd.htm
[3] Gil M. (s.f.). Concurrencia y paralelismo. Recuperado de http://people.ac.upc.edu/marisa/miso/concurrencia.pdf
[4] Narang M. (s.f.). Multithreading in Java. Recuperado de https://www.geeksforgeeks.org/multithreading-in-java/
[5] Multisoft Virtual Academy. (2015). Common Advantages and Disadvantages of Multithreading in Java. Recuperado de https://www.multisoftvirtualacademy.com/blog/common-advantages-and-disadvantages-of-multithreading-in-java/
[6] Baeldung. (2019). Guide to Fork/Join Framework in Java. Recuperado de https://www.baeldung.com/java-fork-join
[7] Heller M. (2018). What is CUDA? Parallel Programming for GPUs. Recuperado de https://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html
[8] Intel (s.f.). Intel Threading Building Blocks. Recuperado de https://software.intel.com › en-us › tbb

Apéndices

Java Secuencial

```
import java.awt.image.BufferedImage;
import java.io.File;
import javax.imageio.ImageIO;


public class SepiaSeq {
  private int[] src;
  private int[] dest;
  private int width;
  private int height;


  public SepiaSeq(int src[], int dest[], int width, int
height) {
    this.src = src;
    this.dest = dest;
    this.width = width;
    this.height = height;
  }


  private void RGBToSepia() {
```

```
    int i, size;
    int ren, col;
    size = width * height;
    for(i = 0; i < size; i++) {
      ren = i / width;
      col = i % width;
      setPixel(ren, col);
    }
  }

  private void setPixel(int ren, int col) {
    int pixel = src[(ren * width) + col];

    float r = (float) ((pixel & 0x00ff0000) >> 16);
    float g = (float) ((pixel & 0x0000ff00) >> 8);
    float b = (float) ((pixel & 0x000000ff) >> 0);

    float tr = (float)(0.393 * r + 0.769 * g + 0.189 * b);
    float tg = (float)(0.349 * r + 0.686 * g + 0.168 * b);
    float tb = (float)(0.272 * r + 0.534 * g + 0.131 * b);

    tr = (tr > 255) ? 255 : tr;
    tg = (tg > 255) ? 255 : tg;
    tb = (tb > 255) ? 255 : tb;

    int color = (0xff000000)
    | ((int)(tr) << 16)
    | ((int)(tg) << 8)
    | ((int)(tb));

    dest[(ren * width) + col] = color;
  }

  public static void main(String args[]) throws Exception {
    long startTime, stopTime;
    double acum = 0;
    if(args.length != 1) {
      System.out.println("Usage: java SepiaSeq
[image_file]");
      System.exit(-1);
    }

    final String filename = args[0];
```

```
    File srcFile = new File(filename);
    final BufferedImage source = ImageIO.read(srcFile);
    int w = source.getWidth();
    int h = source.getHeight();
    int src[] = source.getRGB(0, 0, w, h, null, 0, w);
    int dest[] = new int[src.length];


    SepiaSeq obj = new SepiaSeq(src, dest, w, h);
    for(int i = 0; i < 10; i++) {
      startTime = System.currentTimeMillis();
      obj.RGBToSepia();
      stopTime = System.currentTimeMillis();
      acum += (stopTime - startTime);
    }
    System.out.printf("avg time = %.5f\n", (acum / 10));
          final BufferedImage destination = new
BufferedImage(w, h, BufferedImage.TYPE_INT_ARGB);
          destination.setRGB(0, 0, w, h, dest, 0, w);


          javax.swing.SwingUtilities.invokeLater(new
Runnable() {
            public void run() {
               ImageFrame.showImage("Original - " + filename,
source);
            }
    });


          javax.swing.SwingUtilities.invokeLater(new
Runnable() {
            public void run() {
               ImageFrame.showImage("Sepia - " + filename,
destination);
            }
    });
  }
}
```

Java Threads

```
import java.awt.image.BufferedImage;
import java.io.File;
import javax.imageio.ImageIO;


public class SepiaThreads extends Thread {
  private int src[];
```

```
  private int dest[];
  private int width;
  private int height;
  private int start;
  private int end;


  public SepiaThreads(int start, int end, int src[], int
dest[], int width, int height) {
    this.start = start;
    this.end = end;
    this.src = src;
    this.dest = dest;
    this.width = width;
    this.height = height;
  }


  public void run() {
    int i;
    int ren, col;
    for(i = start; i < end; i++) {
      ren = i / width;
      col = i % width;
      setPixel(ren, col);
    }
  }


  private void setPixel(int ren, int col) {
    int pixel = src[(ren * width) + col];


    float r = (float) ((pixel & 0x00ff0000) >> 16);
    float g = (float) ((pixel & 0x0000ff00) >> 8);
    float b = (float) ((pixel & 0x000000ff) >> 0);


    float tr = (float)(0.393 * r + 0.769 * g + 0.189 * b);
    float tg = (float)(0.349 * r + 0.686 * g + 0.168 * b);
    float tb = (float)(0.272 * r + 0.534 * g + 0.131 * b);


    tr = (tr > 255) ? 255 : tr;
    tg = (tg > 255) ? 255 : tg;
    tb = (tb > 255) ? 255 : tb;


    int color = (0xff000000)
    | ((int)(tr) << 16)
```

```
    | ((int)(tg) << 8)
    | ((int)(tb));


    dest[(ren * width) + col] = color;
  }


  public static void main(String args[]) throws Exception {
    long startTime, stopTime;
    double acum = 0;
    if(args.length != 1) {
      System.out.println("Usage: java SepiaSeq
[image_file]");
      System.exit(-1);
    }


    final String filename = args[0];
    File srcFile = new File(filename);
    final BufferedImage source = ImageIO.read(srcFile);
    int w = source.getWidth();
    int h = source.getHeight();
    int src[] = source.getRGB(0, 0, w, h, null, 0, w);
    int dest[] = new int[src.length];


    int block = (src.length /
Runtime.getRuntime().availableProcessors());
    SepiaThreads[] threads = new
SepiaThreads[Runtime.getRuntime().availableProcessors()];


    for(int j = 0; j < 10; j++) {
      for(int i = 0; i < threads.length; i++) {
        if(i != threads.length - 1) {
          threads[i] = new SepiaThreads((i * block), ((i + 1)
* block), src, dest, w, h);
        } else {
          threads[i] = new SepiaThreads((i * block),
src.length, src, dest, w, h);
        }
      }
      startTime = System.currentTimeMillis();
      for(int i = 0; i < threads.length; i++) {
        threads[i].start();
      }
      for(int i = 0; i < threads.length; i++) {
        try {
```

```
          threads[i].join();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
      stopTime = System.currentTimeMillis();
      acum += (stopTime - startTime);
    }
    System.out.printf("avg time = %.5f\n", (acum / 10));
    final BufferedImage destination = new BufferedImage(w, h,
BufferedImage.TYPE_INT_ARGB);
    destination.setRGB(0, 0, w, h, dest, 0, w);


    javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
               ImageFrame.showImage("Original - " + filename,
source);
            }
    });


    javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
               ImageFrame.showImage("Blur - " + filename,
destination);
            }
    });
  }


}
```

Java Fork-Join

```
import java.awt.image.BufferedImage;
import java.io.File;
import javax.imageio.ImageIO;
import java.util.concurrent.RecursiveAction;
import java.util.concurrent.ForkJoinPool;


public class SepiaJoin extends RecursiveAction {
  private static final int MIN = 10_000;
  private int[] src;
  private int[] dest;
  private int start, end, width, height;
```

```
  public SepiaJoin(int start, int end, int[] src, int[] dest,
int width, int height) {
    this.start = start;
    this.end = end;
    this.src = src;
    this.dest = dest;
    this.width = width;
    this.height = height;
  }


  protected void computeDirectly() {
    int i;
    int ren, col;
    for (i = start; i < end; i++) {
      ren = i / width;
      col = i % width;
      setPixel(ren, col);
    }
  }


  private void setPixel(int ren, int col) {
    int pixel = src[(ren * width) + col];


    float r = (float) ((pixel & 0x00ff0000) >> 16);
    float g = (float) ((pixel & 0x0000ff00) >> 8);
    float b = (float) ((pixel & 0x000000ff) >> 0);


    float tr = (float)(0.393 * r + 0.769 * g + 0.189 * b);
    float tg = (float)(0.349 * r + 0.686 * g + 0.168 * b);
    float tb = (float)(0.272 * r + 0.534 * g + 0.131 * b);


    tr = (tr > 255) ? 255 : tr;
    tg = (tg > 255) ? 255 : tg;
    tb = (tb > 255) ? 255 : tb;


    int color = (0xff000000)
    | ((int)(tr) << 16)
    | ((int)(tg) << 8)
    | ((int)(tb));


    dest[(ren * width) + col] = color;
  }
```

```
  @Override
  protected void compute() {
    if((end - start) <= MIN) {
      computeDirectly();
    } else {
      int mid = start + ((end - start) / 2);
      invokeAll(
        new SepiaJoin(start, mid, src, dest, width, height),
        new SepiaJoin(mid, end, src, dest, width, height)
      );
    }
  }


  public static void main(String args[]) throws Exception {
    long startTime, stopTime;
    double acum = 0;
    if(args.length != 1) {
      System.out.println("Usage: java SepiaSeq
[image_file]");
      System.exit(-1);
    }


    final String filename = args[0];
    File srcFile = new File(filename);
    final BufferedImage source = ImageIO.read(srcFile);
    int w = source.getWidth();
    int h = source.getHeight();
    int src[] = source.getRGB(0, 0, w, h, null, 0, w);
    int dest[] = new int[src.length];


    ForkJoinPool pool;
    for(int i = 0; i < 10; i++) {
      startTime = System.currentTimeMillis();
      pool = new
ForkJoinPool(Runtime.getRuntime().availableProcessors());
      pool.invoke(new SepiaJoin(0, w * h, src, dest, w, h));
      stopTime = System.currentTimeMillis();
      acum += (stopTime - startTime);
    }
    System.out.printf("avg time = %.5f\n", (acum / 10));
          final BufferedImage destination = new
BufferedImage(w, h, BufferedImage.TYPE_INT_ARGB);
          destination.setRGB(0, 0, w, h, dest, 0, w);
```

```
          javax.swing.SwingUtilities.invokeLater(new
Runnable() {
             public void run() {
                ImageFrame.showImage("Original - " + filename,
source);
             }
    });


          javax.swing.SwingUtilities.invokeLater(new
Runnable() {
             public void run() {
                ImageFrame.showImage("Blur - " + filename,
destination);
             }
    });
  }
}
```

CUDA Secuencial

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv/highgui.h>
#include "cppheader.h"


void set_pixel(IplImage * src, IplImage *dest, int ren, int
col) {
  int pos;
  int step = src->widthStep / sizeof(uchar);
  pos = (ren * step) + (col * src->nChannels);
  unsigned char r = (float)src->imageData[pos + 2];
  unsigned char g = (float)src->imageData[pos + 1];
  unsigned char b = (float)src->imageData[pos + 0];
  dest->imageData[pos + 2] = (unsigned char) (((0.393f * r +
0.769f * g + 0.189f * b) > 255) ? 255 : (0.393f * r + 0.769f
* g + 0.189f * b));
  dest->imageData[pos + 1] = (unsigned char) (((0.349f * r +
0.686f * g + 0.168f * b) > 255) ? 255 : (0.349f * r + 0.686f
* g + 0.168f * b));
  dest->imageData[pos + 0] = (unsigned char) (((0.272f * r +
0.534f * g + 0.131f * b) > 255) ? 255 : (0.272f * r + 0.534f
* g + 0.131f * b));


}
```

```
void grayscale(IplImage* src, IplImage* dest) {
  int size = src->width * src->height;
  int row, col;
  for(int i = 0; i < size; i++) {
    row = i / src->width;
    col = i % src ->width;
    set_pixel(src, dest, row, col);
  }
}


int main(int argc, char* argv[]) {
  if(argc != 2) {
    printf("Usage: %s [dir_image_source]\n", argv[0]);
    return -1;
  }


  IplImage *src = cvLoadImage(argv[1], CV_LOAD_IMAGE_COLOR);
     IplImage *dest = cvCreateImage(cvSize(src->width, src-
>height), IPL_DEPTH_8U, 3);
     if (!src) {
          printf("Could not load image file: %s\n", argv[1]);
          return -1;
     }
  Timer t;
  double acum = 0;
  for(int i = 0; i < 10; i++) {
    t.start();
    grayscale(src, dest);
    acum += t.stop();
  }


  printf("avg time = %.5lf ms\n", (acum / 10));


  cvShowImage("Original", src);
  cvShowImage("Sepia", dest);
  cvWaitKey(0);
  cvDestroyWindow("Original");
  cvDestroyWindow("Sepia");


  return 0;
}
```

CUDA Paralelizado

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv/highgui.h>
#include "cuda_runtime.h"
#include "cppheader.h"


__global__ void grayscale(unsigned char *src, unsigned char
*dest, int width,
                          int height, int nChannels) {
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  int x = blockIdx.x * blockDim.x + threadIdx.x;


  if(y < height && x < width) {
    int pos = (y * width + x) * nChannels;


    float r = src[pos + 2];
    float g = src[pos + 1];
    float b = src[pos + 0];


    dest[pos + 2] = ((0.393f * r + 0.769f * g + 0.189f * b) >
255) ? 255 : (0.393f * r + 0.769f * g + 0.189f * b);
    dest[pos + 1] = ((0.349f * r + 0.686f * g + 0.168f * b) >
255) ? 255 : (0.349f * r + 0.686f * g + 0.168f * b);
    dest[pos + 0] = ((0.272f * r + 0.534f * g + 0.131f * b) >
255) ? 255 : (0.272f * r + 0.534f * g + 0.131f * b);
  }
}


int main(int argc, char* argv[]) {
  int i, size;
     double acum;
     unsigned char *dev_src, *dev_dest;
  Timer t;


     if (argc != 2) {
          printf("usage: %s [dir_image_source]\n", argv[0]);
          return -1;
     }


     IplImage *src = cvLoadImage(argv[1],
CV_LOAD_IMAGE_COLOR);
     IplImage *dest = cvCreateImage(cvSize(src->width, src-
>height), IPL_DEPTH_8U, 3);
```

```
     if (!src) {
          printf("Could not load image file: %s\n", argv[1]);
          return -1;
     }


     size = src->width * src->height * src->nChannels *
sizeof(uchar);
     cudaMalloc((void**) &dev_src, size);
     cudaMalloc((void**) &dev_dest, size);


     cudaMemcpy(dev_src, src->imageData, size,
cudaMemcpyHostToDevice);


  dim3 dimGrid(ceil((float)src->width / 16),
               ceil((float)src->height / 16));
  dim3 dimBlock(16, 16, 1);


     acum = 0;
     for (i = 0; i < 10; i++) {
          t.start();
          grayscale<<<dimGrid, dimBlock>>>(dev_src, dev_dest,
src->width, src->height, src->nChannels);
          acum += t.stop();
     }


     cudaMemcpy(dest->imageData, dev_dest, size,
cudaMemcpyDeviceToHost);


     cudaFree(dev_dest);
     cudaFree(dev_src);


     printf("avg time = %.5lf ms\n", (acum / 10));


     cvShowImage("Original", src);
     cvShowImage("Sepia", dest);
     cvWaitKey(0);
     cvDestroyWindow("Original");
     cvDestroyWindow("Sepia");


     return 0;
}
```

C++ Secuencial

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv/highgui.h>
#include "cppheader.h"


class SepiaSeq {
private:
  IplImage *src;
  IplImage *dest;


  void set_pixel(int ren, int col) {
    int pos;
    int step = src->widthStep / sizeof(uchar);
    pos = (ren * step) + (col * src->nChannels);
    unsigned char r = (float)src->imageData[pos + 2];
    unsigned char g = (float)src->imageData[pos + 1];
    unsigned char b = (float)src->imageData[pos + 0];
    dest->imageData[pos + 2] = (unsigned char) (((0.393f * r
+ 0.769f * g + 0.189f * b) > 255) ? 255 : (0.393f * r +
0.769f * g + 0.189f * b));
    dest->imageData[pos + 1] = (unsigned char) (((0.349f * r
+ 0.686f * g + 0.168f * b) > 255) ? 255 : (0.349f * r +
0.686f * g + 0.168f * b));
    dest->imageData[pos + 0] = (unsigned char) (((0.272f * r
+ 0.534f * g + 0.131f * b) > 255) ? 255 : (0.272f * r +
0.534f * g + 0.131f * b));


  }


public:
  SepiaSeq(IplImage* source, IplImage* destination)
:src(source), dest(destination) {}


  void grayscale(IplImage* src, IplImage* dest) {
    int size = src->width * src->height;
    int row, col;
    for(int i = 0; i < size; i++) {
      row = i / src->width;
      col = i % src ->width;
      set_pixel(row, col);
    }
  }
```

```
};



int main(int argc, char* argv[]) {
  if(argc != 2) {
    printf("Usage: %s [dir_image_source]\n", argv[0]);
    return -1;
  }


  IplImage *src = cvLoadImage(argv[1], CV_LOAD_IMAGE_COLOR);
     IplImage *dest = cvCreateImage(cvSize(src->width, src-
>height), IPL_DEPTH_8U, 3);
     if (!src) {
          printf("Could not load image file: %s\n", argv[1]);
          return -1;
     }
  Timer t;
  SepiaSeq sq(src, dest);
  double acum = 0;
  for(int i = 0; i < 10; i++) {
    t.start();
    sq.grayscale(src, dest);
    acum += t.stop();
  }


  printf("avg time = %.5lf ms\n", (acum / 10));


  cvShowImage("Original", src);
  cvShowImage("Sepia", dest);
  cvWaitKey(0);
  cvDestroyWindow("Original");
  cvDestroyWindow("Sepia");


  return 0;
}
```

C++ TBB

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv/highgui.h>
```

```
#include <tbb/blocked_range.h>
#include <tbb/task_scheduler_init.h>
#include <tbb/parallel_for.h>

#include "cppheader.h"

using namespace tbb;

const int GRAIN = 10000;

class SepiaPar {
private:
  IplImage *src;
  IplImage *dest;

  void set_pixel(int ren, int col) const{
    int pos;
    int step = src->widthStep / sizeof(uchar);
    pos = (ren * step) + (col * src->nChannels);
    unsigned char r = (float)src->imageData[pos + 2];
    unsigned char g = (float)src->imageData[pos + 1];
    unsigned char b = (float)src->imageData[pos + 0];
    dest->imageData[pos + 2] = (unsigned char) (((0.393f * r
+ 0.769f * g + 0.189f * b) > 255) ? 255 : (0.393f * r +
0.769f * g + 0.189f * b));
    dest->imageData[pos + 1] = (unsigned char) (((0.349f * r
+ 0.686f * g + 0.168f * b) > 255) ? 255 : (0.349f * r +
0.686f * g + 0.168f * b));
    dest->imageData[pos + 0] = (unsigned char) (((0.272f * r
+ 0.534f * g + 0.131f * b) > 255) ? 255 : (0.272f * r +
0.534f * g + 0.131f * b));

  }

public:
  SepiaPar(IplImage* source, IplImage* destination)
:src(source), dest(destination) {}

  void operator() (const blocked_range<int> &r) const {
    int row, col;
    for(int i = r.begin(); i != r.end(); i++) {
      row = i / src->width;
      col = i % src ->width;
```

```
      set_pixel(row, col);
    }
  }

};



int main(int argc, char* argv[]) {
  if(argc != 2) {
    printf("Usage: %s [dir_image_source]\n", argv[0]);
    return -1;
  }


  IplImage *src = cvLoadImage(argv[1], CV_LOAD_IMAGE_COLOR);
     IplImage *dest = cvCreateImage(cvSize(src->width, src-
>height), IPL_DEPTH_8U, 3);
     if (!src) {
          printf("Could not load image file: %s\n", argv[1]);
          return -1;
     }
  Timer t;
  double acum = 0;
  int size = src->width * src->height;
  for(int i = 0; i < 10; i++) {
    t.start();
    parallel_for( blocked_range<int>(0, size, GRAIN),
SepiaPar(src, dest) );
    acum += t.stop();
  }


  printf("avg time = %.5lf ms\n", (acum / 10));


  cvShowImage("Original", src);
  cvShowImage("Sepia", dest);
  cvWaitKey(0);
  cvDestroyWindow("Original");
  cvDestroyWindow("Sepia");


  return 0;
}
```

1. Generalidades

El trabajo de investigación deberá ser elaborado de manera individual. Cada estudiante seleccionará un problema de programación que pueda ser paralelizado usando al menor cuatro de las cinco herramientas que fueron cubiertas en clase:

- *Threads* en Java.
- Fork/Join *framework* en Java.
- OpenMP en C.
- Intel Threading Building Blocks en C++.
- CUDA.

El problema seleccionado también debe ser resuelto de manera secuencial en cada uno de los lenguajes utilizados.

Se deben comparar los tiempos de ejecución de los programas escritos en los diferentes lenguajes y herramientas, tanto en sus versiones secuenciales como paralelas. Así mismo, se deben calcular los diferentes *speedups* obtenidos y se deben realizar una o varias gráficas comparativas que presenten la información de manera clara.

Los resultados de la investigación deberán reportarse como un artículo de divulgación escrita en idioma español, utilizando el formato de este documento. El artículo deberá tener una extensión de por lo menos cinco páginas (sin incluir los apéndices). En particular, el trabajo debe contar con las siguientes secciones:

- **Datos:** Título y autor de la investigación.
- **Resumen:** Contenido abreviado y preciso del artículo.
- **Introducción:** Descripción del problema que se va a resolver.
- **Desarrollo:** Explicación de la manera en que se resolvió el problema usando cada una de las herramientas.
- **Conclusiones:** Enunciar las conclusiones personales sobre los resultados obtenidos.
- **Agradecimientos:** Esta sección es opcional.
- **Referencias:** Citar al menos 5 fuentes bibliográficas usando el estilo APA.
- **Apéndices:** Incluir en esta sección todos los códigos fuentes que fueron elaborados para la investigación.

El público al que está dirigido el artículo son los mismos compañeros de esta clase, así que se debe tener esto en mente al momento de realizar la redacción.

Se evaluará la calidad técnica del artículo, así como su redacción, ortografía y claridad y orden de la presentación de las ideas.

La fecha de entrega de este trabajo es el día del examen final.

2. Consejos

2.1 Código fuente

Para colocar listados de código fuente se deberá usar tipo de letra `Courier` de 12 pts. Por ejemplo:

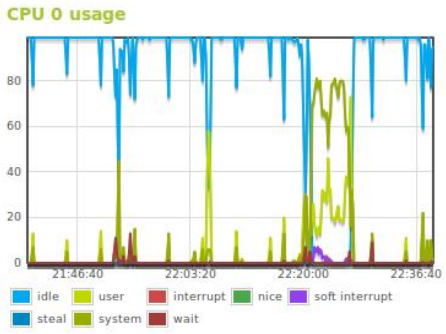
```
include <stdio.h>


int main() {
     printf("Hola Mundo!\n");
     return 0;
}
```

Los elementos de un programa que aparecen en un párrafo de texto normal (como el nombre de la función `main`) deben ir con el mismo tipo de letra.

2.2 Imágenes.

Se sugiere una caja de texto para insertar una imagen (idealmente de 300 pdi) porque, es un documento de Word, este método es de alguna manera más estable que insertar la imagen directamente.



Para que el margen no será visible, selecciona Formato > Línea > Sin Línea.

3. Referencias

[1] GIYBF. Google is your best Friend!, http://www.giybf.com/ Accedido el 9 de abril del 2013.