Convoluciones usando multiprocesamiento

Pedro González Roa (A01651517)
Tecnológico de Monterrey, Campus Querétaro
A01651517@itesm.mx

27 de noviembre de 2015

I. Resumen

II. Introducción

En matemáticas una convolución es una operación entre dos funciones f(x) y g(x) que genera una tercera función que representa la magnitud en la que se superponen f(x) y una versión trasladada e invertida de g(x). Esta operación se utiliza mucho para el procesamiento de imágenes, ya que permite hacer transformaciones únicas con algoritmos de menor esfuerzo.

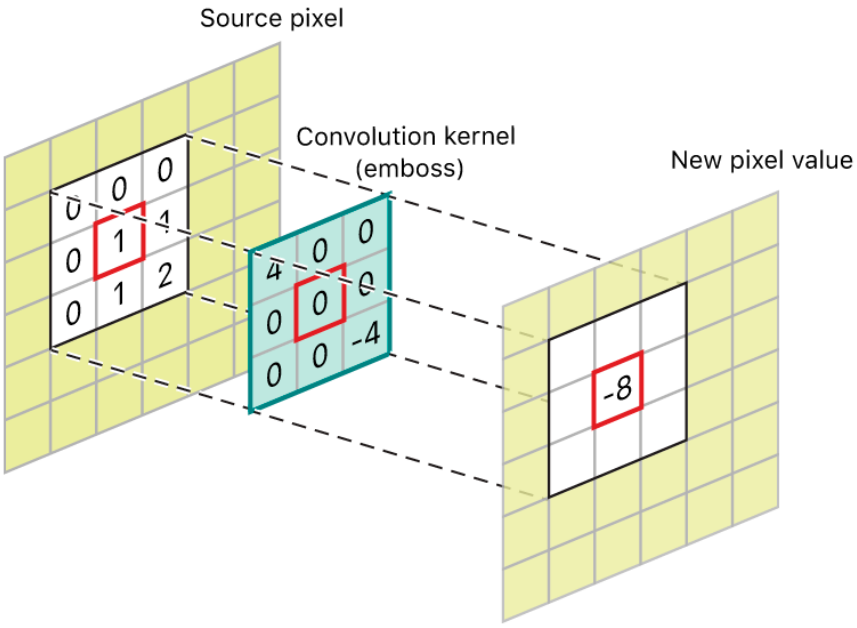
$$g(x,y) = \omega * f(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx,dy) f(x+dx, y+dy),$$

https://en.wikipedia.org/wiki/Kernel_(image_processing)

Muchas de estas transformaciones son utilizadas para realizar filtros o modificaciones en herramientas populares como Photoshop o Gimp, ayudando a modificar la nitidez, el delineado o el relieve de una imagen. Por otro lado, estas transformaciones también son utilizadas para extraer *'features'* o características específicas que son utilizadas para inteligencia artificial ya que ayudan a tener una mejor abstracción en el reconocimiento de imágenes.

https://en.wikipedia.org/wiki/Kernel_(image_processing)

El proceso para realizar una convolución de una imágen consiste en sobreponer un filtro, o *kernel,* sobre cada píxel de la imágen. Este filtro realizará una suma de la multiplicación de cada uno de sus elementos con los elementos de la imagen debajo. Al finalizar con el primer pixel, pasará el siguiente elemento de la imagen base para realizar la misma operación.



https://developer.apple.com/documentation/accelerate/blurring_an_image

Todo este proceso implica una gran cantidad de sumas y multiplicaciones que se incrementan en complejidad de manera proporcional con el aumento de detalle en una imagen. Gracias a los avances tecnológicos del último siglo, somos capaces de obtener imágenes con mucha mayor calidad y también somos capaces de almacenar un mayor número de estas. Por lo que el procesamiento con un sólo núcleo o hilo ya no tiene suficiente velocidad para mantener el paso con estos desarrollos, y es por eso que ahora utilizamos métodos de multiprocesamiento para la realización de esta tarea.

III. Desarrollo

Esta investigación tiene como propósito comparar la velocidad entre el procesamiento de un solo hilo y el procesamiento con varios hilos con diferentes lenguajes de programación y librerías especializadas. Los lenguajes de programación y librerías que utilizaremos en esta investigación son:

C++

El lenguaje de programación C++, es un lenguaje de propósito general creado por Bjarne Stroustrup como una extensión del lenguaje de programación C. También conocido como 'C con clases', este lenguaje se ha expandido mucho en los últimos años agregando nuevas funcionalidades. C++ es muy conocido por su velocidad y por ser un lenguaje de bajo nivel.

OpenMP

OpenMP, o cómo sus siglas significan Open Multi-processing, es una API (Interfaz de programación de aplicaciones) construida para los lenguajes de C, C++ y fortran. Consiste en directivas de compilador, librerías de rutinas y variables de ambiente para la utilización de los hilos del procesador.

TBB

oneAPI, o conocido previamente como TBB (Threading Building Blocks), es una librería del lenguaje C++ desarrollada por Intel para la utilización de programación en paralelo.

CUDA

CUDA es una plataforma de computación y un modelo de programación desarrollado por NVIDIA para computación y procesamiento en paralelo. A diferencia de los previamente mencionados, OpenMP y TBB, esta está enfocada el la unidad de procesamiento gráfico (GPU).

Java

Java es un lenguaje de programación de alto nivel de propósito general con el enfoque *write once, run anywhere* que consiste en que el código base pueda ser utilizado por cualquier máquina. Fue desarrollado por la compañía de Oracle y tuvo su primera aparición en Mayo de 1995.

El desarrollo de esta investigación consistió en comparar la velocidad de ejecución de cada una de las herramientas previamente mencionadas con cuatro imágenes diferentes:

1. La primera imágen es la foto de una flor que tiene dimensiones de 128x128 pixeles. Es decir un total de 49,152 valores, ya que cada pixel tiene los tres canales para generar un color (rgb).

2. La segunda imágen es una pintura famosa del artista Vincent van Gogh. Esta tiene dimensiones de 720x1280 píxeles, con un total de 2.764,800 valores.

3. Nuestra tercer imágen es un fondo de pantalla publicado en reddit, inspirado en la caricatura de Avatar: el último maestro aire. Esta tiene dimensiones de 1080x1920 píxeles, dando un total de 6,220,800 valores.

4. Finalmente, nuestra última imagen que utilizaremos será la de una pintura de un dinosaurio. Esta tiene dimensiones de 1440x3440 píxeles, dando un total de 14,860,800 valores.

Para comprobar el funcionamiento correcto de las imágenes, utilizaremos dos filtros de los cuales sabemos el resultado de la transformación. Ambos tienen el propósito de dar un efecto de imágen borrosa utilizando valores Gausianos. El primer filtro tiene dimensiones 3x3 y el último tiene dimensiones 5x5.

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

$$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

IV. Resultados

Primero nos enfocaremos en comparar los tiempos de ejecución de cada una de las herramientas. En la figura 1, podemos observar que incluso utilizando un optimizador de compilador para que se obtenga una mayor velocidad, el lenguaje de C fue casi nueve veces más lento en promedio que el lenguaje de programación de Java. Incluso este último supera en tiempo de ejecución a las otras librerías de multiprocesamiento con el CPU.

En la tabla de los resultados, el tiempo de ejecución de CUDA se mantuvo constante (0.0023 milisegundos). Esto es gracias a que tiene suficientes hilos para que cada pixel, incluso de la imágen más grande, sea atendido sin ningún tiempo de espera.

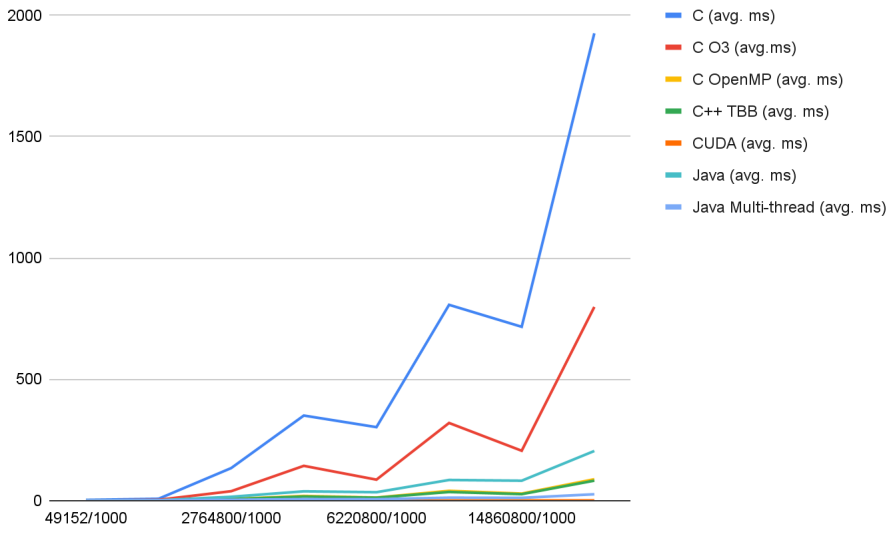
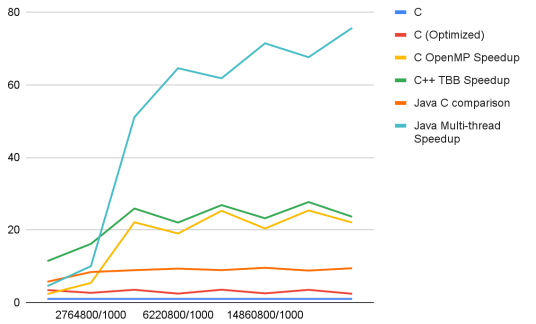


Figura 1: Tiempo de procesamiento



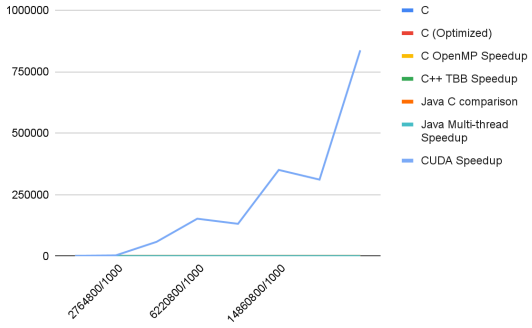


Figura 2 y Figura 3: Speed up

V. Conclusiones

Al contrario de lo que muchos programadores consideramos, java tuvo una mayor velocidad que el lenguaje de C++. Esto demuestra que las optimizaciones que realiza java al compilar, pueden ser incluso más rápidas que las de un lenguaje de bajo nivel. Aunque java fue superior a sus compañeros de multiprocesamiento en el procesador, queda muy lejos de CUDA, quien claramente se ve beneficiado por su arquitectura que fue diseñada para estos tipos de procesamientos.

VII. Referencias

CUDA Zone. (n.d.). NVIDIA Developer. Retrieved November 25, 2021, from https://developer.nvidia.com/cuda-zone

Kennedy, P. (n.d.). *OpenMP*. Wikipedia. Retrieved November 25, 2021, from https://en.wikipedia.org/wiki/OpenMP

Kernel (image processing). (n.d.). - Wikipedia. Retrieved November 25, 2021, from https://en.wikipedia.org/wiki/Kernel_(image_processing)

Kroll, R. (n.d.). *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. Towards Data Science. Retrieved November 25, 2021, from https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Peters, A. (n.d.). *Convolution*. Wikipedia. Retrieved November 25, 2021, from https://en.wikipedia.org/wiki/Convolution

Powell, V. (n.d.). *Image Kernels explained visually*. Setosa.IO. Retrieved November 25, 2021, from https://setosa.io/ev/image-kernels/

Stroustrup, B., Bloch, J., & De Simone, S. (n.d.). *C++*. Wikipedia. Retrieved November 25, 2021, from https://en.wikipedia.org/wiki/C%2B%2B

Threading Building Blocks. (n.d.). Wikipedia. Retrieved November 25, 2021, from https://en.wikipedia.org/wiki/Threading_Building_Blocks

VIII. Apéndices

1. Código Fuente C++

```
/*-----------------------------------------------------------
-----
* Programación avanzada: Proyecto final
* Fecha: 25-Nov-2021
* Autor: A01651517 Pedro González
*-----------------------------------------------------------
--*/
#ifndef UTILS_H
#define UTILS_H

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <sys/types.h>

#define ITERATIONS 100

struct timeval startTime, stopTime;
int started = 0;

//
============================================================
====
// Records the initial execution time.
// @author: Manchas2k4
//
============================================================
====
```

```
void start_timer() {
     started = 1;
     gettimeofday(&startTime, NULL);
}

//
================================================================
====
// Calculates the number of microseconds that have elapsed
since
// the initial time.
//
// @returns the time passed
// @author: Manchas2k4
//
================================================================
====
double stop_timer() {
     long seconds, useconds;
     double duration = -1;

     if (started) {
          gettimeofday(&stopTime, NULL);
          seconds  = stopTime.tv_sec  - startTime.tv_sec;
          useconds = stopTime.tv_usec - startTime.tv_usec;
          duration  =  (seconds  *  1000.0)  +  (useconds  /
1000.0);
          started = 0;
     }
     return duration;
}

//
================================================================
====
// Class that contains all the relevant data and the method
to extract
// said data for the implementations of Convolution.
//
================================================================
====
```

```
class ConvContext {
    private:
        cv::Mat src, dst;
        int kRows, kCols;
        float *kernel;

        /* Scan kernel using scanf */
        float * scanKernel() {
            float *kernel;

            // Make space in memory
                                            kernel    =    (float
*)malloc(sizeof(float)*kRows*kCols);

            // Scan kernel
            for (int i = 0; i < kRows*kCols; i++)
                fscanf(stdin, "%f", &kernel[i]);

            return kernel;
        }

    public:
        ConvContext(std::string imagePath, bool color) {
            // Read image
            src = (color)?
                cv::imread(imagePath, cv::IMREAD_COLOR):
                cv::imread(imagePath, cv::IMREAD_GRAYSCALE);

            // Scan kernel
            fscanf(stdin, "%i %i", &kRows, &kCols);
            if (kCols <= 0 || kRows <= 0)
                    throw std::runtime_error("Invalid kernel
dimensions!");
            kernel = scanKernel();
        }

        ~ConvContext() { free(kernel); }

        void setDestination(uchar *dstRaw) {
            dst = cv::Mat(
                src.rows,
```

```
                src.cols,
                src.type(),
                dstRaw,
                cv::Mat::AUTO_STEP
            );
        }

          /* Function to display the original image and the
destination image */
        void display() {
            cv::namedWindow("Original", cv::WINDOW_AUTOSIZE);
            cv::imshow("Original", src);

                                    cv::namedWindow("Convolved",
cv::WINDOW_AUTOSIZE);
            cv::imshow("Convolved", dst);

            cv::waitKey(0);
        }

        void printSize(FILE *file) {
            fprintf(file, "Image size: %ix%ix%i = %i\n",
                          src.rows, src.cols, src.channels(),
getSize());
        }

        // Getters
        const cv::Mat *getSrc() { return &src; }
        const cv::Mat *getDst() { return &dst; }
        uchar *getData() const { return src.data; }
        float *getKernel() const { return kernel; }
        int getKRows() { return kRows; }
        int getKCols() { return kCols; }
        int getKSize() { return kRows*kCols; }
        int getRows() { return src.rows; }
        int getCols() { return src.cols; }
        int getChannels() { return src.channels(); }
        int imgSize() { return src.cols*src.rows; }
                                int   getSize()   {   return
src.rows*src.cols*src.channels(); }
};
```

```
#endif
```

```
/*-------------------------------------------------------------
-----
* Programación avanzada: Proyecto final
* Fecha: 25-Nov-2021
* Autor: A01651517 Pedro González
*-------------------------------------------------------------
--*/
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/highgui/highgui_c.h>
#include <opencv2/imgcodecs/imgcodecs.hpp>
#include <opencv2/imgproc/imgproc.hpp>

#include "utils.h"

uchar * convolution(ConvContext *context) {
    uchar *dst, byte, *dta = context->getData();
    float *k = context->getKernel();
    int spos, pos, channels = context->getChannels();

    // Make space for destination
               dst    =    (uchar    *)malloc(sizeof(uchar)    *
context->getSize());

    for (int i = 0; i < context->getSize(); i++) {
        byte = 0;
        spos = i - (int)(context->getKSize()/2)*channels;
        for (int f = 0; f < context->getKSize(); f++) {
            int pos = spos+f*channels;
             if (pos > 0 && pos < context->getSize()) byte +=
dta[pos]*k[f];
        }
        dst[i] = byte;
    }

    return dst;
}

int main(int argc, char *argv[]) {
```

```
    ConvContext *context;
    double ms;
    uchar *dst;
    bool grayscale = true;

    if (argc != 2 && argc != 3) {
        fprintf(stderr, "usage: %s <image file>\n", argv[0]);
        return -1;
    }

    if (argc == 3) grayscale = false;

    // Get context
    context = new ConvContext(argv[1], grayscale);
    context->printSize(stdout);

    // Run algorithm n times
    for (int i = 0; i < ITERATIONS; i++) {
        start_timer();

        // Only save on last iteration
        if (i == ITERATIONS-1) dst = convolution(context);
        else convolution(context);

        ms += stop_timer()/ITERATIONS;
    }
    fprintf(stdout, "Calculation time: %.5f ms\n", ms);

    context->setDestination(dst);
    context->display();

    delete context;
    free(dst);
    return 0;
}
```

2. Código Fuente OpenMP

```
/*-------------------------------------------------------------
-----
* Programación avanzada: Proyecto final
* Fecha: 25-Nov-2021
* Autor: A01651517 Pedro González
*-------------------------------------------------------------
--*/
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/highgui/highgui_c.h>
#include <opencv2/imgcodecs/imgcodecs.hpp>
#include <opencv2/imgproc/imgproc.hpp>

#include "utils.h"

uchar * convolution(ConvContext *context) {
    uchar *dst, byte, *dta = context->getData();
    float *k = context->getKernel();
    int spos, pos, channels = context->getChannels();

    // Make space for destination
              dst    =    (uchar    *)malloc(sizeof(uchar)    *
context->getSize());

    #pragma omp parallel for
    for (int i = 0; i < context->getSize(); i++) {
        byte = 0;
        spos = i - (int)(context->getKSize()/2)*channels;
        for (int f = 0; f < context->getKSize(); f++) {
            int pos = spos+f*channels;
             if (pos > 0 && pos < context->getSize()) byte +=
dta[pos]*k[f];
        }
        dst[i] = byte;
    }

    return dst;
```

```
}

int main(int argc, char *argv[]) {
    ConvContext *context;
    double ms;
    uchar *dst;
    bool grayscale = true;

    if (argc != 2 && argc != 3) {
        fprintf(stderr, "usage: %s <image file>\n", argv[0]);
        return -1;
    }

    if (argc == 3) grayscale = false;

    // Get context
    context = new ConvContext(argv[1], grayscale);
    context->printSize(stdout);

    // Run algorithm n times
    for (int i = 0; i < ITERATIONS; i++) {
        start_timer();

        // Only save on last iteration
        if (i == ITERATIONS-1) dst = convolution(context);
        else convolution(context);

        ms += stop_timer()/ITERATIONS;
    }
    fprintf(stdout, "Calculation time: %.5f ms\n", ms);

    context->setDestination(dst);
//    context->display();

    delete context;
    free(dst);
    return 0;
}
```

3. Código Fuente TBB

```
/*-------------------------------------------------------------
-----
* Programación avanzada: Proyecto final
* Fecha: 25-Nov-2021
* Autor: A01651517 Pedro González
*-------------------------------------------------------------
--*/
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/highgui/highgui_c.h>
#include <opencv2/imgcodecs/imgcodecs.hpp>
#include <opencv2/imgproc/imgproc.hpp>

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

#include "utils.h"

using namespace tbb;

class ConvolutionTBB {
private:
    ConvContext *context;
    uchar *dst;

public:
    ConvolutionTBB(ConvContext *context_) : context(context_)
{
                  dst   =   (uchar   *)malloc(sizeof(uchar)   *
context->getSize());
    }

    void operator() (const blocked_range<int> &r) const {
        uchar byte;
        int spos, pos;

        for (int i = r.begin(); i != r.end(); i++) {
```

```
            byte = 0;
                                                spos    =    i    -
(int)(context->getKSize()/2)*context->getChannels();
            for (int f = 0; f < context->getKSize(); f++) {
                int pos = spos+f*context->getChannels();
                if (pos > 0 && pos < context->getSize())
                                                          byte   +=
context->getData()[pos]*context->getKernel()[f];
            }
            dst[i] = byte;
        }
    }

    uchar * getRes() { return dst; }
};

int main(int argc, char *argv[]) {
    ConvContext *context;
    double ms;
    bool grayscale = true;

       if  (argc  ==  3  &&  strcmp(argv[2],  "--grey")  ==  0)
grayscale = true;
    else if (argc != 2) {
        fprintf(stderr, "usage: %s source_file\n", argv[0]);
        return -1;
    }

    // Get context
    context = new ConvContext(argv[1], grayscale);
    context->printSize(stdout);

    // Run algorithm n times
    for (int i = 0; i < ITERATIONS; i++) {
        start_timer();

          ConvolutionTBB obj(context);
          parallel_for(blocked_range<int>(0,
context->getSize()),  obj);

        // Only save on last iteration
```

```
                                if    (i    ==    ITERATIONS-1)
context->setDestination(obj.getRes());

        ms += stop_timer()/ITERATIONS;
        fprintf(stdout, "%.5f ms\n", ms);
    }
    fprintf(stdout, "Calculation time: %.5f ms\n", ms);

//    context->display();

    delete context;
    return 0;
}
```

4. Código Fuente CUDA

```
/*-------------------------------------------------------------
-----
* Programación avanzada: Proyecto final
* Fecha: 25-Nov-2021
* Autor: A01651517 Pedro González
*-------------------------------------------------------------
--*/
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/core/core.hpp>
#include <opencv2/core/cuda.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/highgui/highgui_c.h>
#include <opencv2/imgcodecs/imgcodecs.hpp>
#include <opencv2/imgproc/imgproc.hpp>

#include "utils.h"

#define TPB 256

__global__
void convolution(const uchar *src, const int channels, const
float *kernel,
const int kSize, uchar *dst, const int size) {
    int tid = threadIdx.x + (blockIdx.x * blockDim.x);
    if (tid < size) {
        int spos = tid - (int)(kSize/2)*channels;
        uchar byte = 0;
        for (int i = 0; i < kSize; i++)
            byte += src[spos+i*channels]*kernel[i];
        dst[tid] = byte;
    }
}

int main(int argc, char *argv[]) {
    ConvContext *context;
    double ms;
    bool grayscale = false;
    uchar *gpu_src, *gpu_dst, *dstRaw;
```

```
    float *gpu_kernel;

       if (argc == 3 && strcmp(argv[2], "--gray") == 0)
grayscale = true;
    else if (argc != 2) {
        fprintf(stderr, "usage: %s source_file\n", argv[0]);
        return -1;
    }

    // Get context
    context = new ConvContext(argv[1], grayscale);
    context->printSize(stdout);

    // Upload image and kernel to gpu
         fprintf(stdout, "Uploading image and kernel to
GPU...\n");
                          cudaMalloc((void**)       &gpu_src,
sizeof(uchar)*context->getSize());
                  cudaMemcpy(gpu_src,     context->getData(),
sizeof(uchar)*context->getSize(),
        cudaMemcpyHostToDevice);
                        cudaMalloc((void**)      &gpu_kernel,
sizeof(float)*context->getKSize());
               cudaMemcpy(gpu_kernel,    context->getKernel(),
sizeof(float)*context->getKSize(),
        cudaMemcpyHostToDevice);

    // Make space for result
            dstRaw    =    (uchar    *)malloc(sizeof(uchar)    *
context->getSize());
          cudaMalloc((void**)   &gpu_dst,   sizeof(uchar)   *
context->getSize());

    // Run algorithm n times
    for (int i = 0; i < ITERATIONS; i++) {
        start_timer();

        convolution<<<context->getSize()/TPB, TPB>>>(gpu_src,
                        context->getChannels(),   gpu_kernel,
context->getKSize(), gpu_dst,
        context->getSize());
```

```
        ms += stop_timer()/ITERATIONS;
    }
    fprintf(stdout, "Calculation time: %.5f ms\n", ms);

    // Copy processed image to CPU
                            cudaMemcpy(dstRaw,       gpu_dst,
sizeof(uchar)*context->getSize(),
        cudaMemcpyDeviceToHost);
    context->setDestination(dstRaw);
//    context->display();

                    cudaFree(gpu_src);     cudaFree(gpu_dst);
cudaFree(gpu_kernel);
    free(dstRaw);
    return 0;
}
```

5. Código Fuente Java

```
/*-------------------------------------------------------------
-----
* Programación avanzada: Proyecto final
* Fecha: 25-Nov-2021
* Autor: A01651517 Pedro González
*-------------------------------------------------------------
--*/
import java.awt.image.BufferedImage;
import java.io.File;
import javax.imageio.ImageIO;
import java.io.IOException;
import java.util.Scanner;

public class Convolution {
    public static final int ITERATIONS = 100;
     private int src[], dst[], rows, cols;
    private float kernel[];
    private int kRows, kCols;

     public Convolution(int src[], int dst[], int rows, int
cols,
    float kernel[], int kRows, int kCols) {
          this.src = src; this.dst = dst;
          this.cols = cols; this.rows = rows;

         this.kernel = kernel; this.kRows = kRows; this.kCols
= kCols;
     }

     private void convolution() {
        int size = rows*cols, ksize = kRows*kCols;
        int spos, pos, pixel, dpixel;
        float r, g, b;

        for (int i = 0; i < size; i++) {
            r = 0; g = 0; b = 0;
            spos = i - (int)Math.floor(ksize/2);
            for (int k = 0; k < ksize; k++) {
                pos = spos + k;
```

```
                if (pos > 0 && pos < size) {
                    pixel = src[spos+k];
                     r += (float)((pixel & 0x00ff0000) >> 16)
* kernel[k];
                    g += (float)((pixel & 0x0000ff00) >> 8) *
kernel[k];
                    b += (float)((pixel & 0x000000ff) >> 0) *
kernel[k];
                }
            }

            dpixel = (0xff000000)
                    | (((int)r) << 16)
                    | (((int)g) << 8)
                    | (((int)b) << 0);
            dst[i] = dpixel;
        }
     }

     public static void main(String args[]) throws Exception
{
          long startTime, stopTime;
          double ms;

          if (args.length != 1) {
               System.out.println("usage:    java    Example10
image_file");
               System.exit(-1);
          }

        // Read image
              final  BufferedImage  source  =  ImageIO.read(new
File(args[0]));

          int rows = source.getHeight();
          int cols = source.getWidth();
          int  src[]  =  source.getRGB(0,  0,  cols,  rows,  null,
0, cols);
          int dst[] = new int[src.length];

        // Scan kernel
```

```
        Scanner scanner = new Scanner(System.in);
        int kRows = scanner.nextInt();
        int kCols = scanner.nextInt();
        float kernel[] = new float[kRows*kCols];

          for (int i = 0; i < kRows*kCols; i++) kernel[i] =
scanner.nextFloat();

          ms = 0;
          Convolution conv = new Convolution(src, dst, rows,
cols, kernel, kRows,
            kCols);
          for (int i = 0; i < ITERATIONS; i++) {
               startTime = System.currentTimeMillis();

               conv.convolution();

               stopTime = System.currentTimeMillis();
               ms += (stopTime - startTime);
          }
          System.out.printf("avg     time     =     %.4f\n",
ms/ITERATIONS);
          final    BufferedImage    destination    =    new
BufferedImage(cols, rows,
            BufferedImage.TYPE_INT_ARGB);
          destination.setRGB(0, 0, cols, rows, dst, 0, cols);

          try {
               ImageIO.write(destination,     "png",     new
File("conv.png"));
               System.out.println("Image     was     written
succesfully.");
          } catch (IOException ioe) {
               System.out.println("Exception  occured  :"  +
ioe.getMessage());
          }
     }
}
```

6. Código Fuente Java con hilos de multiprocesamiento

```
/*-------------------------------------------------------------
-----
* Programación avanzada: Proyecto final
* Fecha: 25-Nov-2021
* Autor: A01651517 Pedro González
*-------------------------------------------------------------
--*/
import java.awt.image.BufferedImage;
import java.io.File;
import javax.imageio.ImageIO;
import java.io.IOException;
import java.util.Scanner;

public class ThreadsConvolution extends Thread {
    public static final int ITERATIONS = 100;
              public   static   final   int   MAXTHREADS   =
Runtime.getRuntime().availableProcessors();
     private int src[], dst[], rows, cols, start, end;
    private float kernel[];
    private int kRows, kCols;

     public  ThreadsConvolution(int  src[],  int  dst[],  int
rows, int cols,
    float kernel[], int kRows, int kCols, int start, int end)
{
          this.src = src; this.dst = dst;
          this.cols = cols; this.rows = rows;

         this.kernel = kernel; this.kRows = kRows; this.kCols
= kCols;

        this.start = start; this.end = end;
     }

     private void convolution() {
        int size = rows*cols, ksize = kRows*kCols;
        int spos, pos, pixel, dpixel;
        float r, g, b;
```

```
        for (int i = start; i < end; i++) {
            r = 0; g = 0; b = 0;
            spos = i - (int)Math.floor(ksize/2);
            for (int k = 0; k < ksize; k++) {
                pos = spos + k;
                if (pos > 0 && pos < size) {
                    pixel = src[pos];
                     r += (float)((pixel & 0x00ff0000) >> 16)
* kernel[k];
                    g += (float)((pixel & 0x0000ff00) >> 8) *
kernel[k];
                    b += (float)((pixel & 0x000000ff) >> 0) *
kernel[k];
                }
            }

            dpixel = (0xff000000)
                    | (((int)r) << 16)
                    | (((int)g) << 8)
                    | (((int)b) << 0);
            dst[i] = dpixel;
        }
     }

    public void run() { convolution(); }

     public static void main(String args[]) throws Exception
{
          long startTime, stopTime;
          double ms;

          if (args.length != 1) {
               System.out.println("usage:    java    Example10
image_file");
               System.exit(-1);
          }

        // Read image
              final BufferedImage source = ImageIO.read(new
File(args[0]));
```

```
          int rows = source.getHeight();
          int cols = source.getWidth();
          int src[] = source.getRGB(0, 0, cols, rows, null,
0, cols);
          int dst[] = new int[src.length];

        // Scan kernel
        Scanner scanner = new Scanner(System.in);
        int kRows = scanner.nextInt();
        int kCols = scanner.nextInt();
        float kernel[] = new float[kRows*kCols];

          for (int i = 0; i < kRows*kCols; i++) kernel[i] =
scanner.nextFloat();

          ThreadsConvolution      threads[]      =      new
ThreadsConvolution[MAXTHREADS];
        int block = rows*cols/threads.length;

          ms = 0;
          for (int i = 0; i < ITERATIONS; i++) {
            startTime = System.currentTimeMillis();
            for (int t = 0; t < threads.length; t++)
                if (t != threads.length - 1)
                     threads[t] = new ThreadsConvolution(src,
dst, rows, cols,
                               kernel, kRows, kCols, (t*block),
((t+1)*block));
                else
                     threads[t] = new ThreadsConvolution(src,
dst, rows, cols,
                               kernel, kRows, kCols, (t*block),
(cols*rows));
                  for (int t = 0; t < threads.length; t++)
threads[t].start();
            for (int t = 0; t < threads.length; t++) {
                try {
                         threads[t].join();
                    } catch (InterruptedException e) {
                         e.printStackTrace();
```

```
                    }
                }
                stopTime = System.currentTimeMillis();
                ms += (stopTime - startTime);
            }
            System.out.printf("avg     time     =     %.4f\n",
ms/ITERATIONS);
            final    BufferedImage    destination    =    new
BufferedImage(cols, rows,
                BufferedImage.TYPE_INT_ARGB);
            destination.setRGB(0, 0, cols, rows, dst, 0, cols);

            try {
                ImageIO.write(destination,     "png",     new
File("conv.png"));
                System.out.println("Image     was     written
succesfully.");
            } catch (IOException ioe) {
                System.out.println("Exception  occured  :"  +
ioe.getMessage());
            }
        }
}
```