

# Filtro de mediana implementado en C++ con manejo de hilos usando OpenMP, POSIX y CUDA

Universidad Nacional de Colombia

Juan Jesus Pulido Sanchez  
jjpulidos@unal.edu.co

Cristian Camilo Garcia Barrera  
ccgarciaib@unal.edu.co

Danier Elian Gonzalez Ordóñez  
dgonzalezo@unal.edu.co

## I. DISEÑO

El filtro de mediana es un tipo de filtro no lineal comúnmente usado para eliminar el ruido de una imagen, este filtro funciona moviéndose a través de una imagen píxel a píxel, reemplazando cada valor con el valor de la mediana de los píxeles vecinos.

El patrón de píxeles vecinos se denomina como "ventana", la cual se desliza píxel a píxel sobre toda la imagen. La mediana se calcula ordenando primero todos los valores de píxeles de la ventana en orden numérico y luego se reemplaza el píxel que se está considerando con el valor del píxel medio.

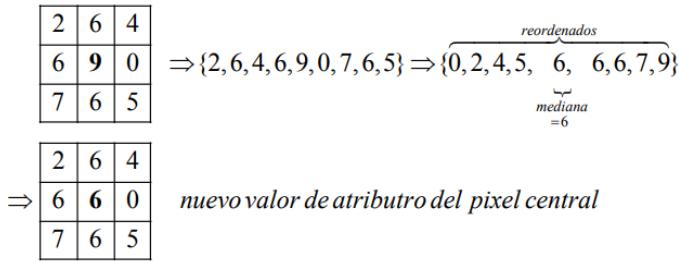


Figure 1. Ejemplo para un píxel y una ventana de 3x3.

Para el proceso de lectura y escritura de la imagen se hace uso de la biblioteca OpenCV, al momento de la lectura la función imread me permite cargar una imagen desde un archivo especificado que retorna una matriz y al momento de escritura me permite guardar una imagen en un archivo especificado a partir de una imagen en forma de matriz.

```
//Lectura de la imagen
Mat image=imread(filename, IMREAD_COLOR);
//Escritura de la imagen
imwrite(filename,image,compression_params)
```

Para la implementación secuencial se recorrió la matriz de la imagen por sus filas y columnas con un tamaño de ventana "ksize", reordenando la ventana y devolviendo su valor medio para cada iteración.

```
uchar medianFilterWindow(const cv::  
                           Mat &src, int i, int j){  
    vector<uchar> pixel(ksize * ksize);  
    for(int k = 0; k < ksize * ksize; ++k){  
        pixel[k]=src.at<uchar>(i+  
                               delta[k].first,  
                               j+ delta[k].second);  
    }  
    sort(pixel.begin(), pixel.end());  
    return pixel[(ksize * ksize) / 2];  
}  
  
Mat medianFilter(const cv::Mat &src){  
    dst = src.clone();  
    for(int i=1; i<src.rows-ksize; ++i){  
        for(int j=1; j< src.cols-ksize; ++j){  
            dst.at<uchar>(i, j) =  
                medianFilterWindow(src, i, j);  
        }  
    }  
    return dst;  
}
```

Para el proceso de paralelización se usa block-wise, de tal manera que se separa el problema entre el número de hilos. En este caso se separaron la cantidad de filas de la imagen entre el número de hilos, de forma que cada hilo calcula todos los píxeles de un conjunto consecutivo de filas.

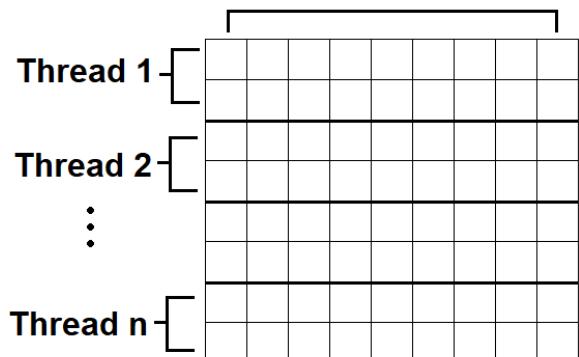


Figure 2. Ejemplo de funcionamiento de hilos.

Para la paraleización del código se realiza por medio de hilos POSIX con la librería pthread.h implementada en c++. Primero se inicializan arreglos para guardar los parámetros de ID's de los hilos:

```
//Thread variables
int threadId[total_threads];
pthread_t thread[total_threads];
```

Posteriormente se desarrolla un ciclo que creara todos los hilos. En este caso la función que se ejecuta se llama medianFilter.

```
for(int i = 0; i < total_threads; i++) {
    threadId[i] = i;
    pthread_create(&thread[i], NULL,
                  medianFilter, &threadId[i]);
}
```

Con los hilos creados se utiliza la función pthread\_join() de tal manera que se espere hasta que todos los hilos terminen su ejecución.

```
//Join Threads
for(int i = 0; i < total_threads; i++) {
    pthread_join(thread[i], NULL);
}
```

Para la paraleización del código por medio de OpenMP se realizo por medio de la librería omp.h implementada en c++. Haciendo uso del pragma for.

```
//parallel processing
#pragma omp parallel for
for(int i=0; i<total_threads; i++) {
    medianFilter(i);
}
```

Para la paraleización del código usando CUDA se realizo haciendo uso de kerneles que operan sobre la memoria global, usando bloques bidimensionales de hilos.

```
__global__ void median_filter_thread(
    const uchar *inputImageKernel,
    uchar *outputImagekernel,
    const int imageWidth,
    const int imageHeight) {

    int WINDOW_SIZE = 3;
    int row = blockIdx.y *
              blockDim.y + threadIdx.y;
    int col = blockIdx.x *
              blockDim.x + threadIdx.x;
    unsigned char filterVector[9] =
        {0, 0, 0, 0, 0, 0, 0, 0, 0};

    if ((row == 0) || (col == 0)
        || (row == imageHeight - 1) ||
        (col == imageWidth - 1)) {
```

```
        outputImagekernel[row * imageWidth + col] = 0;
    } else {
        for (int x=0;x<WINDOW_SIZE;x++) {
            for (int y=0;y<WINDOW_SIZE;y++) {
                filterVector[x*WINDOW_SIZE+y] =
                    inputImageKernel[(row+x-1) * imageWidth+(col+y-1)];
            }
        }
        for (int i=0;i<9;i++) {
            for (int j=i+1;j<9;j++) {
                if (filterVector[i]>
                    filterVector[j]) {
                    // Swap the variables.
                    char tmp = filterVector[i];
                    filterVector[i] =
                        filterVector[j];
                    filterVector[j]=tmp;
                }
            }
        }
        outputImagekernel[row * imageWidth+col]=filterVector[4];
    }
}
```

## II. PRUEBAS

En los experimentos se variaron el numero de hilos (2, 4, 8, 14) y la calidad de las imágenes (720p, 1080p y 4K), con un valor de "ksize" estático en 5, sacando como resultado el promedio de 10 ejecuciones del programa con cada variación. La maquina usada para las pruebas cuenta con un procesador Intel i7-8750H con 12 hilos y 6 núcleos, con una capacidad de 16 GB de RAM.

Secuencial	
Calidad	Rt
720p	11,91679
1080p	27,67189
4k	109,9624

Figure 3. Resultado de tiempo de respuesta para la implementación secuencial

### A. Pruebas con hilos POSIX

Imagen 720p	
#hilos	Rt
1	11,91679
2	6,12430
4	3,07214
8	2,07238
16	1,67992

Figure 4. Resultado de tiempo de respuesta para la implementación paralela con imagen 720p (POSIX)

Imagen 1080p	
#hilos	Rt
1	27,67189
2	14,09668
4	7,05849
8	4,75588
16	3,79395

Figure 5. Resultado de tiempo de respuesta para la implementación paralela con imagen 1080p (POSIX)

Imagen 4k	
#hilos	Rt
1	109,96240
2	56,82300
4	28,08668
8	18,82331
16	14,81538

Figure 6. Resultado de tiempo de respuesta para la implementación paralela con imagen 4k (POSIX)

Imagen 1080p	
#hilos	SpeedUp
1	1,00000
2	1,96301
4	3,92037
8	5,81846
16	7,29368

Figure 8. Resultado de SpeedUp con imagen 1080p (POSIX)

Imagen 4k	
#hilos	SpeedUp
1	1,00000
2	1,93517
4	3,91511
8	5,84182
16	7,42218

Figure 9. Resultado de SpeedUp con imagen 4k (POSIX)

### B. Pruebas con OpenMP

Imagen 720p	
#hilos	Rt
1	11,91307
2	6,14676
4	3,09225
8	2,10741
16	1,96774

Figure 10. Resultado de tiempo de respuesta para la implementación paralela con imagen 720p (OMP)

Imagen 720p	
#hilos	SpeedUp
1	1,00000
2	1,94582
4	3,87899
8	5,75029
16	7,09368

Figure 7. Resultado de SpeedUp con imagen 720p (POSIX)

Imagen 1080p	
#hilos	Rt
1	27,73678
2	14,18401
4	7,08900
8	4,81688
16	4,48887

Figure 11. Resultado de tiempo de respuesta para la implementación paralela con imagen 1080p (OMP)

Imagen 4k	
#hilos	SpeedUp
1	1,00000
2	1,95217
4	3,90009
8	5,78216
16	6,15029

Figure 15. Resultado de SpeedUp con imagen 4k (OMP)

### C. Pruebas con CUDA

Imagen 4k	
#hilos	Rt
1	109,83700
2	56,26410
4	28,16270
8	18,99583
16	17,85882

Figure 12. Resultado de tiempo de respuesta para la implementación paralela con imagen 4k (OMP)

Imagen 720p	
#Bloques	Rt
1	0,01551
2	0,00474
5	0,00127
10	0,00084
20	0,00073
30	0,00072

Figure 16. Resultado de tiempo de respuesta para la implementación paralela con imagen 720p (CUDA)

Imagen 720p	
#hilos	SpeedUp
1	1,00000
2	1,93811
4	3,85256
8	5,65294
16	6,05418

Figure 13. Resultado de SpeedUp con imagen 720p (OMP)

Imagen 1080p	
#Bloques	Rt
1	0,03411
2	0,01066
5	0,00254
10	0,00168
20	0,00128
30	0,00129

Figure 17. Resultado de tiempo de respuesta para la implementación paralela con imagen 1080p (CUDA)

Imagen 1080p	
#hilos	SpeedUp
1	1,00000
2	1,95550
4	3,91265
8	5,75825
16	6,17901

Figure 14. Resultado de SpeedUp con imagen 1080p(OMP)

Imagen 4k	
#Bloques	Rt
1	0,13099
2	0,04036
5	0,00947
10	0,00583
20	0,00437
30	0,00436

Figure 18. Resultado de tiempo de respuesta para la implementación paralela con imagen 4k (CUDA)

Imagen 4k	
#Bloques	SpeedUp
1	838,54384
2	2.721,15096
5	11.596,70042
10	18.855,71005
20	25.158,52779
30	25.218,62556

Figure 19. Resultado de tiempo de respuesta para la implementación paralela con imagen 4k (CUDA)

Imagen 720p	
#Bloques	SpeedUp
1	768,25357
2	2.510,70198
5	9.416,08447
10	14.219,98194
20	16.329,80110
30	16.639,89047

Figure 20. Resultado de SpeedUp con imagen 720p (CUDA)

Imagen 1080p	
#Bloques	SpeedUp
1	813,06627
2	2.602,16660
5	10.935,99571
10	16.485,77951
20	21.606,24551
30	21.498,82998

Figure 21. Resultado de SpeedUp con imagen 1080p (CUDA)

Imagen 4k	
#Bloques	SpeedUp
1	838,54384
2	2.721,15096
5	11.596,70042
10	18.855,71005
20	25.158,52779
30	25.218,62556

Figure 22. Resultado de SpeedUp con imagen 4k (CUDA)

### III. RESULTADOS

#### A. Resultados del SpeedUp para hilos POSIX



Figure 23. Comparación de tiempos de respuesta (POSIX)

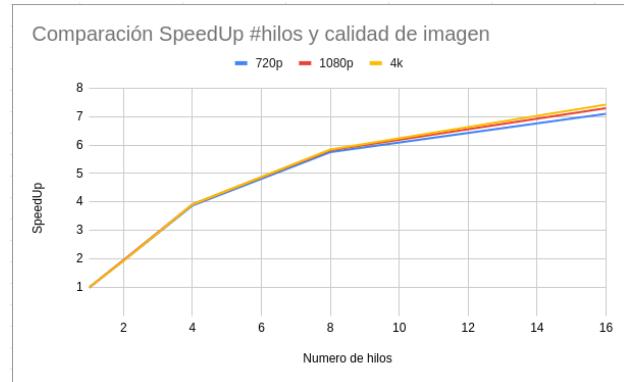


Figure 24. Comparación de SpeedUp (POSIX)

#### B. Resultados del SpeedUp para OpenMP



Figure 25. Comparación de tiempos de respuesta (OMP)

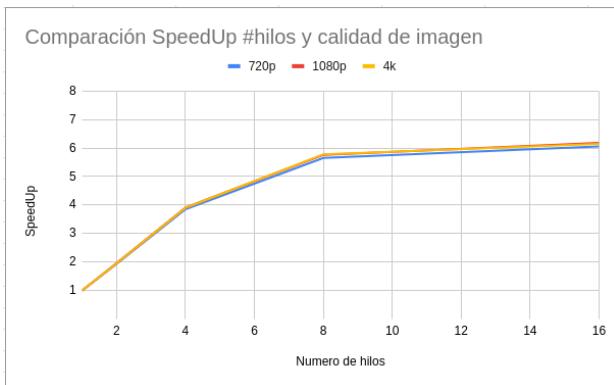


Figure 26. Comparación de SpeedUp (OMP)

### C. Resultados del SpeedUp para bloques CUDA vs Hilos OpenMP

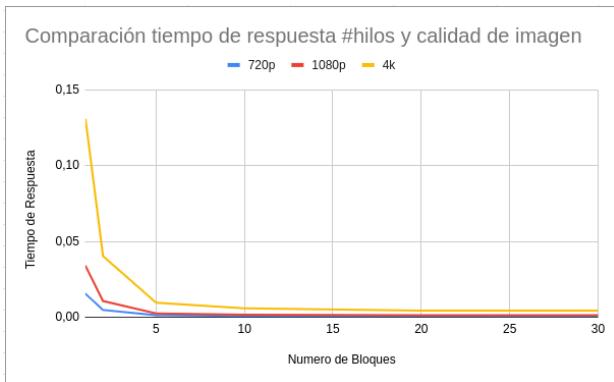


Figure 27. Comparación de tiempos de respuesta (Cuda)

A continuación se hace una comparación de imágenes originales que tienen aplicado un filtro "sal y pimienta" y las imágenes resultado después de aplicar el filtro de mediana para reducir este ruido.

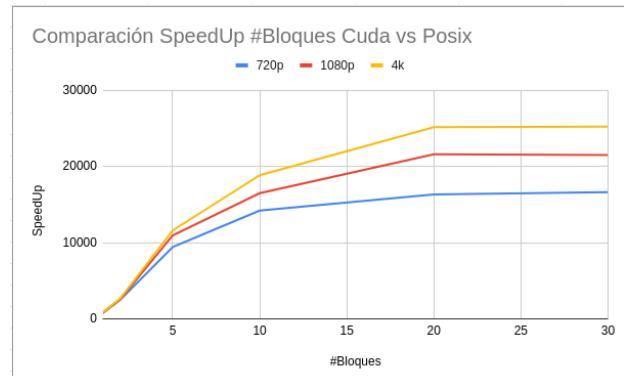


Figure 28. Comparación de SpeedUp de bloques Cuda vs hilos OpenMP



Figure 29. Imagen con ruido 720p



Figure 30. Imagen después de aplicar filtro de mediana 720p

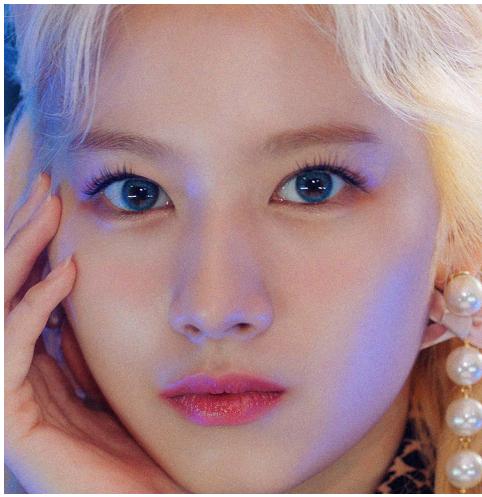


Figure 31. Imagen con ruido 4k



Figure 32. Imagen después de aplicar filtro de mediana 4k

#### IV. CONCLUSIONES

- La convolución como función matemática aplicada en imágenes es fácilmente paralelizable usando hilos POSIX usando C++, por lo tanto en caso de tener un proceso en el que se aplique dicho filtro a una cantidad de imágenes considerable, entonces podemos ahorrar costos computacionales al usar la implementación paralela del filtro de mediana.
- La mejora en velocidad gracias al paralelismo incrementa proporcionalmente al número de hilos después de 4 hilos.
- Dependiendo del tamaño de la imagen el aumento de velocidad de procesamiento cambia, como se evidencia que para imágenes muy grandes de resolución 4k el speed up tiende a ser menor que en imágenes 720p y 1080p después de 4 hilos.
- La paralelización permite acercar considerablemente los tiempos de respuesta para diferentes imágenes. Más investigación ha de ser realizada para llegar a una con-

clusión sólida, pero como hipótesis preliminar podría ser una mejora mayor en la localidad de la memoria procesada para imágenes más grandes.

- Los resultados entre ambos experimentos son muy similares, aunque claramente la diferencia se centra en el tiempo de desarrollo el cual es mucho menor usando OpenMP.
- Los resultados aunque con CUDA no son comparables debido a la forma de realización de la lógica bidimensional en el código, se puede notar que los tiempos de demora por imagen son muy bajos a causa del uso de multiples cores de la GPU.

#### V. REFERENCIAS

- Biblioteca OpenCV para la lectura y escritura de imágenes. Disponible en <https://opencv.org/>
- Marco teórico del filtro de mediana: [http://fourier.eng.hmc.edu/e161/lectures/smooth\\_sharpen/node2.html](http://fourier.eng.hmc.edu/e161/lectures/smooth_sharpen/node2.html)
- Repositorio del código del experimento <https://github.com/jjpulidos/Parallel-Computing-2021-1>