UNIVERSIDAD
NACIONAL
DE COLOMBIA

Benchmark entre la implementación por software y la de shaders de máscara de convolución blur-effect

Computación visual

Juan Sebastian Herrera Maldonado

Profesor: Jean Pierre Charalambos Hernandez

Universidad Nacional de Colombia
Facultad de Ingeniería
Bogotá, Colombia
2018-II

Resumen—En el siguiente artıculo se presenta el resultado y la comparación de la ejecución del algoritmo blur-effect utilizando dos implementaciones diferentes, software (OpenMP,Posix,Cuda) vs shaders (OpenGL en processing). Se exponen los tiempos obtenidos al emplear el algoritmo con imágenes de diversas resoluciones y tamaños, realizando variaciones en el kernel de difuminación, con el objetivo de distinguir los diferentes comportamientos según la herramienta y para el caso de la implementación por software la cantidad de hilos accionados para el procedimiento.

I. INTRODUCCIÓN

El procesamiento de imágenes digitales es el conjunto de técnicas que se aplican a las imágenes digitales con el objetivo de mejorar la calidad o facilitar la búsqueda de información. A estas imágenes se les realiza un proceso de filtrado que es el conjunto de técnicas englobadas dentro del preprocesamiento de imágenes cuyo objetivo fundamental es obtener, a partir de una imagen origen, otra final cuyo resultado sea más adecuado para una aplicación específica mejorando ciertas características de la misma que posibilite efectuar operaciones del procesado sobre ella.

Los principales objetivos que se persiguen con la aplicación de filtros son:

- Suavizar la imagen: reducir la cantidad de variaciones de intensidad entre píxeles vecinos.
- Eliminar ruido: eliminar aquellos píxeles cuyo nivel de intensidad es muy diferente al de sus vecinos y cuyo origen puede estar tanto en el proceso de adquisición de la imagen como en el de transmisión.
- Realzar bordes: destacar los bordes que se localizan en una imagen.
- Detectar bordes: detectar los píxeles donde se produce un cambio brusco en la función intensidad.

Por tanto, se consideran los filtros como operaciones que se aplican a los píxeles de una imagen digital para optimizarla, enfatizar cierta información o conseguir un efecto especial en ella.

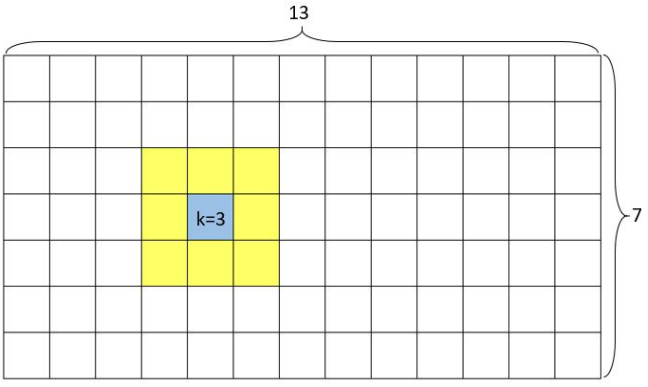
Para el desarrollo del presente artículo se tendrá en cuenta solamente él el filtrado lineal (filtros basados en núcleos o máscaras de convolución)en el dominio del espacio, en este tipo de filtrado Las operaciones de filtrado se llevan a cabo directamente sobre los píxeles de la imagen. En este proceso se relaciona, para todos y cada uno de los puntos de la imagen, un conjunto de píxeles próximos al píxel objetivo con la finalidad de obtener una información útil, dependiente del tipo de filtro aplicado, que permita actuar sobre el píxel concreto en que se está llevando a cabo el proceso de filtrado para, de este modo, obtener mejoras sobre la imagen y/o datos que podrían ser utilizados en futuras acciones o procesos de trabajo sobre ella.

En el caso de estudio presentado a través del artıculo, se realiza un proceso de alteración a distintas imágenes. El objetivo es aplicar varios niveles del efecto de difuminado o borroso, y comparar la diferencia utilizando distintas herramientas (Cuda, OpenMP y posix) de software VS el uso de shaders mediante processing y OpenGL. La intención de los shaders es definir mediante código un comportamiento personalizado del procesamiento gráfico para que sea ejecutado en el núcleo de las tarjetas gráficas. Mediante la programación de shaders, las aplicaciones visuales actuales pueden lograr en tiempo real efectos de gran realismo.

II. ESTRUCTURA DEL ALGORITMO

Aplicar una máscara de convolución consiste en utilizar una matriz, comúnmente de 3x3 creada o determinada por matemáticos o personas dedicadas al filtrado de imágenes, para aplicarla en los pixeles de una imagen. El proceso consiste en tomar un pixel para utilizar sus vecinos y de esta manera colocar la matriz encima y provocar distintos filtros o cambios en la imagen. Para este benchmark se utilizó el blur-effect que utiliza un kernel de convolución que tiene un efecto de promediación. Así que el resultado será un ligero desenfoque (proporcional al kernel).

Dada una imagen de **n × m** píxeles, para cada pıxel se hallará el promedio entre sus vecinos cercanos definido por un kernel impar k, dada la condición $3 \leq k \leq 15$.



En la figura 3 se encuentra una imagen de tamaño 13 × 7, donde el cuadro de color azul hace referencia al pıxel sobre el cual se encuentra posicionado, y los cuadros amarillos el área (o kernel) de 3. En este caso se hallará el promedio de todas las celdas de color para cada valor RGB y se le asigna el resultado al pıxel posición (cuadro azul). Este procedimiento se realiza para todos y cada uno de los p´ıxeles de la imagen. La operación realizada permite realizar el efecto anteriormente mencionado como efecto borroso o difuminado.

Implementación por software

Para este caso el algoritmo descrito fue codificado en el lenguaje C, con la ayuda de la librería <png.h> mediante el uso de memoria dinámica y apuntadores.
El procedimiento llevado a cabo es la extracción de información de los pıxeles de la imagen. Cada uno de los pıxeles es representado en un arreglo de tamaño **n × m**, en donde se almacenan los valores determinados de RGB en forma de arreglo de tamaño 3. La estructura puede verse representada con la siguiente figura.

PÍXEL	1	2	3	...	$n \times m$
R	[0, 255]	[0, 255]	[0, 255]	...	[0, 255]
G	[0, 255]	[0, 255]	[0, 255]	...	[0, 255]
B	[0, 255]	[0, 255]	[0, 255]	...	[0, 255]

Se realizó la paralelización del algoritmo mediante el uso de 3 herramientas distintas. La primera de ellas es mediante la librerıa pthread.h de POSIX que permite la programación de hilos. Para esta herramienta fue necesario tener en cuenta la implementación de un balanceador de carga que distribuye de la mejor forma posible los procesos. La segunda herramienta usada fue la librerıa OpenMP, la cual permite la inclusión del paradigma de paralelismo de una manera muy breve y sencilla, ya que provee distintos métodos y funciones que faculta al programador a realizar la implementación con la inclusión de lıneas básicas, que definen los par ametros y el tipo de paralelismo a realizar, entre ellos la ejecución del proceso y el método de balanceo de carga. La tercer técnica fue mediante el uso de una GPU (o tarjeta gráfica) con la arquitectura CUDA (Compute Unified Device Architecture) de la compania NVIDIA. Aquı fue necesario del uso de conceptos básicos, tales como reserva de memoria en la GPU (cudaMalloc), paso de información entre memorias (cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost), y liberación de memoria (cudaFree).
Para el balanceo de cargas y distribución de datos entre hilos se implementó blockwise. Este método pretende de manera equitativa dividir discretamente la carga y asignar dicha cantidad a los hilos correspondientes para que operen bajo cierto conjunto el conjunto de datos. Como se comentó anteriormente, dado que cada pıxel se representa en un arreglo la división de este ser realiza teniendo en cuenta el coeficiente de carga, dado por:

$$c = \frac{Filas \times Columnas}{Hilos}$$

Implementación por shaders

Los shaders son pequeños programas que se basan en la GPU. Estos programas se ejecutan para cada sección específica de la tubería de gráficos. En un sentido básico, los shaders no son más que programas que transforman entradas en salidas. Los Shaders también son programas muy aislados, ya que no se les permite comunicarse entre sí; La única comunicación que tienen es a través de sus entradas y salidas.
Este procesamiento está predeterminado en las tarjetas de vídeo actuales y consta de tres etapas:

- Una primera etapa en la que se procesan los vértices de los modelos (transformaciones e iluminación).
- Una segunda etapa en la que se analiza cada primitiva para detectar qué porción de la misma queda dentro del área visual o viewport y se procesa cada triángulo para obtener los píxeles que lo forman (rasterization).
- Por último, una etapa en la que se analiza cada píxel y se determina su color y si dicho píxel debe ser visible o no dependiendo de su profundidad.

El código que se utiliza para la implementación de cada una de estas etapas se conoce, respectivamente, como **vertex shader** en el caso del procesamiento de vértices, **geometry shader** en lo relativo a incluir nuevos vértices y triángulos como parte de la primitiva que está siendo procesada, y **pixel/fragment shader** para definir el procesamiento final del píxel a visualizar. Es importante señalar que una vez que se especifica un vertex shader o un pixel shader para definir el procesamiento de vértices o píxeles, los procedimientos predeterminados del hardware gráfico para cada una de estas etapas son reemplazados por el shader. Para el actual experimento nos centraremos en la tercera etapa **pixel/fragment shader.**

Para este caso el algoritmo descrito fue codificado en el lenguaje processing junto el lenguaje de sombreado de OpenGL. Los sombreadores están escritos en el lenguaje C-like GLSL. GLSL está diseñado para su uso con gráficos y contiene características útiles específicamente dirigidas a la manipulación de vectores y matrices. Los sombreadores siempre comienzan con una declaración de versión, seguida de una lista de variables de entrada y salida, uniformes y su principal función (main). El punto de entrada de cada sombreador está en su principal función donde procesamos las variables de entrada y generamos los resultados en sus variables de salida.

El objetivo es tomar el promedio de los texels (la cantidad de texeles va a depender del kernel) que forman una cuadrícula de #Kernel × #Kernel alrededor de un texel dado y el resultado será generar un efecto borroso. Estas nueve muestras de textura serían luego ponderadas y sumadas por el shader de píxeles. Recuerde que para este artículo el kernel será un valor impar k, dada la condición $3 \leq k \leq 15$.

IV. EXPERIMENTOS Y RESULTADOS

Luego de realizado todo el procedimiento de planeación y codificación del programa, se llevó a un ambiente de pruebas, en el cual se tomaron 3 resoluciones base para cada uno de los casos: **720p**, **1080p**, **4K**. Se tomaron cada una de estas resoluciones y se ejecutaron los distintos programas y luego se registraron los tiempos empleados en cada uno de los casos.
En el caso de la implementación por software se tomó cada una de esas resoluciones y se ejecutaron con distinto número de hilos (entre 1 y 16 en el caso de paralelización mediante CPU y el uso de pthreads y OpenMP, y entre 32 y 2496 hilos para el caso de paralelización mediante GPU) Con el objetivo de encontrar el resultado más óptimo tratando de utilizar la gpu que es la gran ventaja de la implementación por shaders.
Posteriormente se realizan gráficas para cada resolución por cada herramienta utilizada.
Para el caso de la implementación en software se decidió graficar el tiempo que se demora en ejecutar las transformaciones, contrapuesta por el número de hilos usado, esto para evidenciar el tiempo mas optimo en esta implementación.
La toma de muestras se realizó en una máquina linux con 8 GB de memoria RAM corriendo sobre un procesador Intel Core i5-7200U a una frecuencia de 2.5 GHz para sus 2 núcleos fısicos más 2 núcleos virtuales . Del mismo modo para el caso de uso de GPU se usó el mismo equipo con una tarjeta gráfica tipo NVIDIA GeForce 940MX.

Ejemplos aplicación algoritmo:

- Imagen original 720 píxeles

- Resultado imagen 720 píxeles con kernel 15

- Imagen original 1080 píxeles

- Resultado imagen 1080 píxeles con kernel 15

- Imagen original 4K píxeles

- Resultado imagen 4K píxeles con kernel 15

Como podemos observar el efecto blur es más notorio cuando la resolución (cantidad de pixeles) es menor.

Resultados graficados:

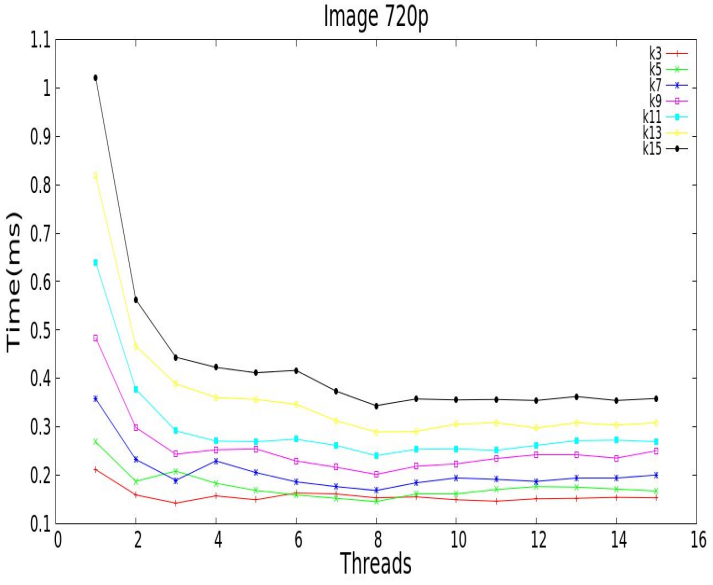
Las gráficas obtenidas de la ejecución del algoritmo serán agrupadas según la implementación (por software y por shaders) y en el caso de la implementación por software también se agruparan las distintas herramientas.

IV-A Software

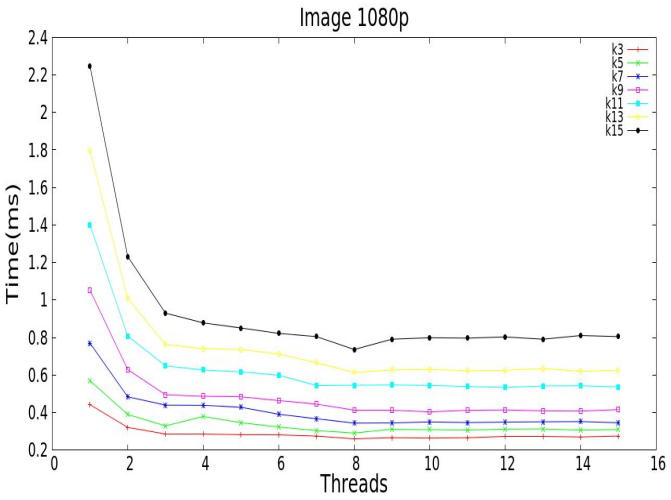
Cada línea representa el valor del kernel asignado.

➔ POSIX

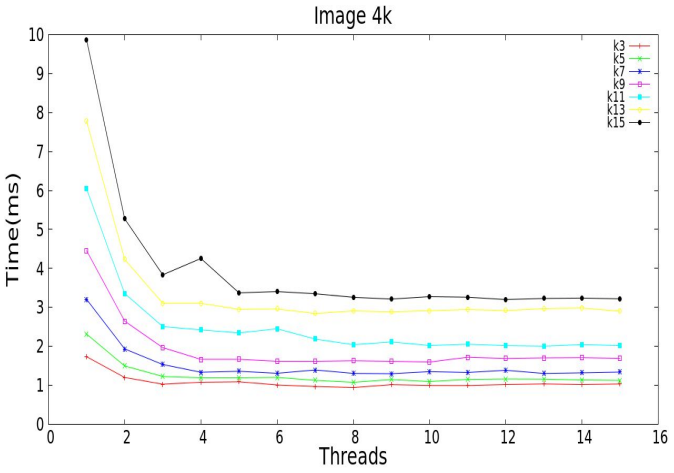
❖ **Gráfica 1 de tiempo (segundos) vs hilos. Imagen 720 píxeles**



❖ **Gráfica 2 de tiempo (segundos) vs hilos. Imagen 1080 píxeles**

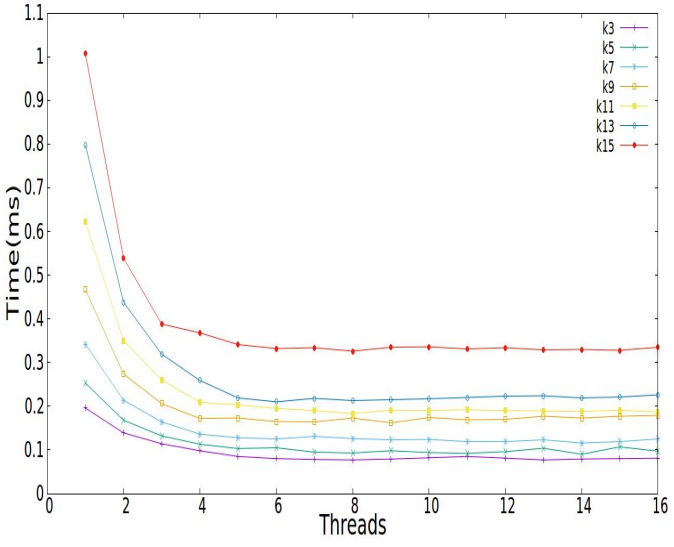


❖ **Gráfica 3 de tiempo (segundos) vs hilos. Imagen 4K píxeles**

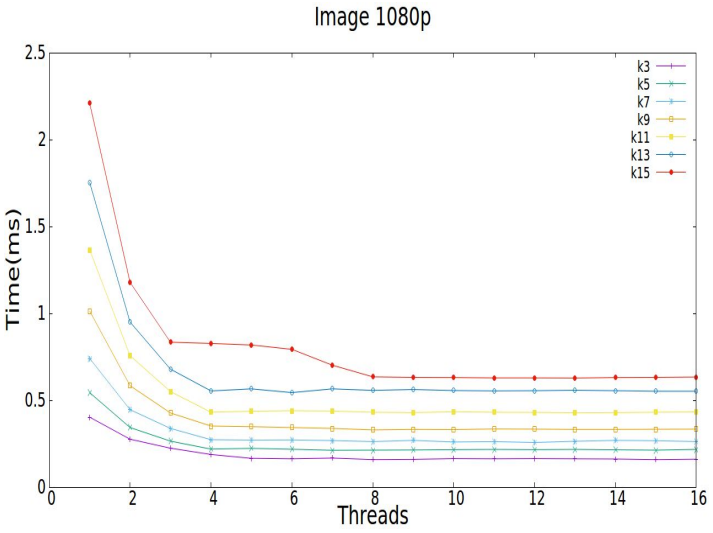


➔ OpenMP

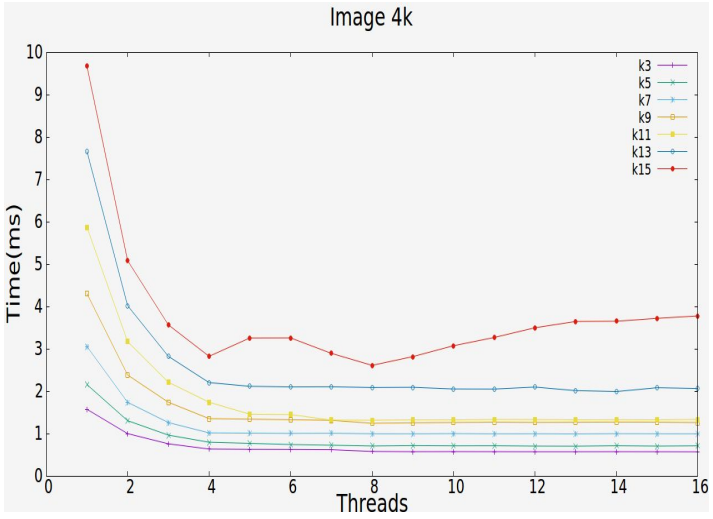
❖ **Gráfica 4 de tiempo (segundos) vs hilos. Imagen 720 píxeles**



❖ **Gráfica 5 de tiempo (segundos) vs hilos. Imagen 1080 píxeles**

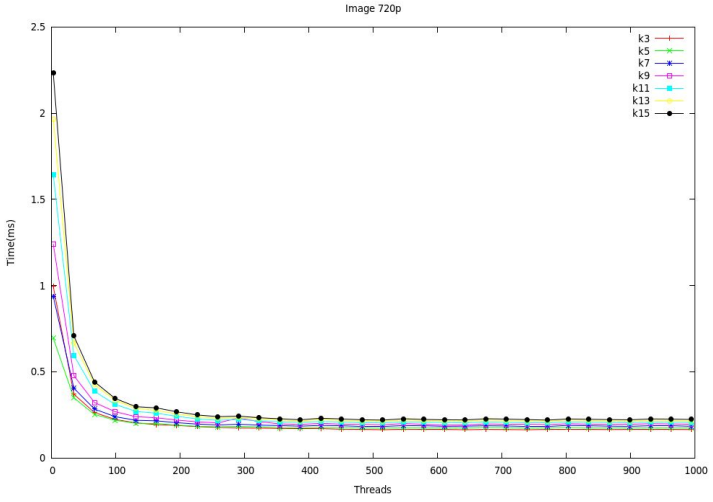


❖ **Gráfica 6 de tiempo (segundos) vs hilos. Imagen 4K píxeles**

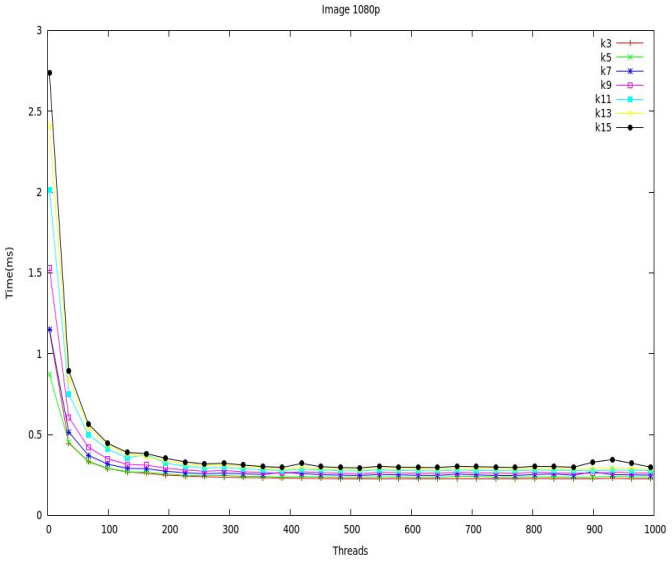


➔ CUDA

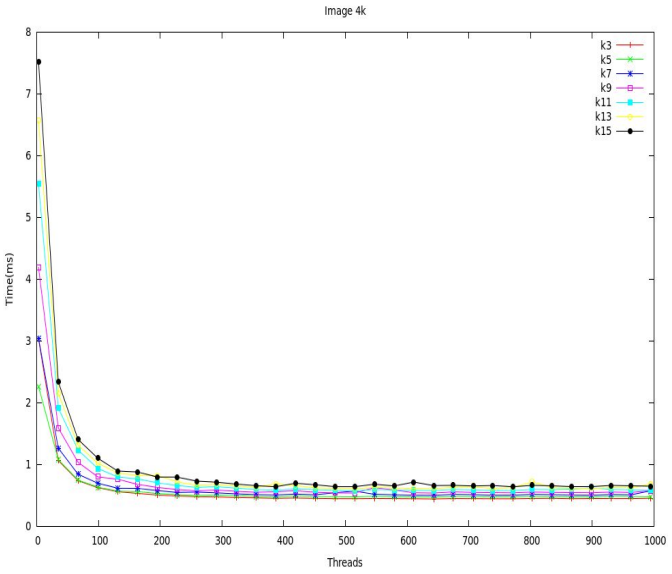
❖ **Gráfica 7 de tiempo (segundos) vs hilos. Imagen 720 píxeles**



❖ **Gráfica 8 de tiempo (segundos) vs hilos. Imagen 1080 píxeles**

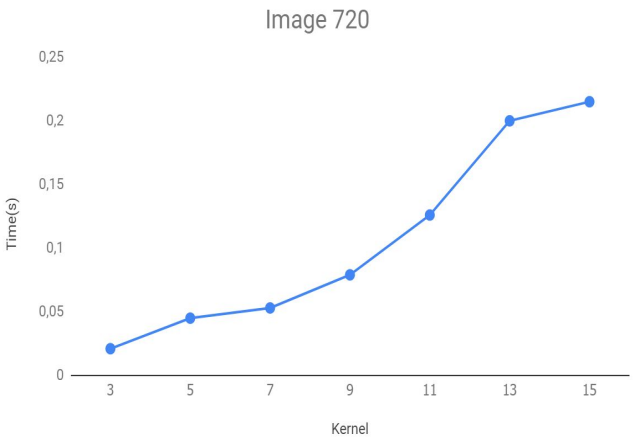


❖ **Gráfica 9 de tiempo (segundos) vs hilos. Imagen 4K píxeles**

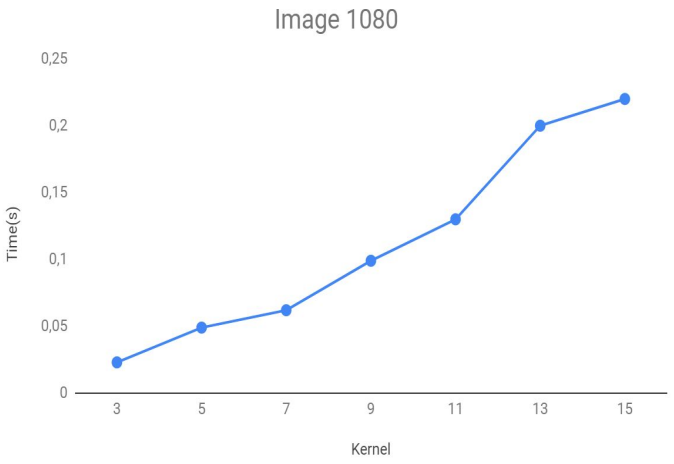


IV-B SHADERS

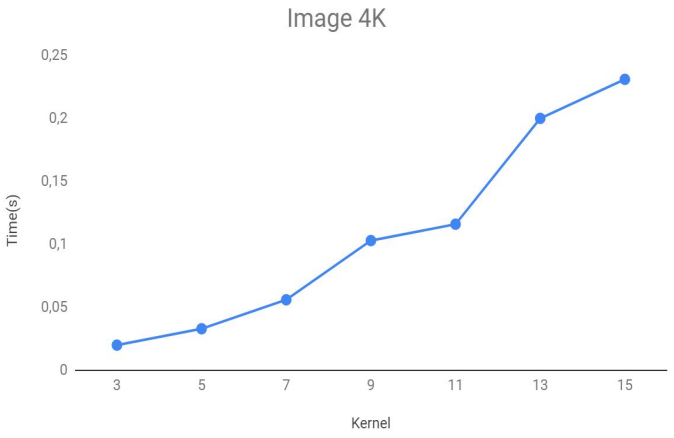
❖ **Gráfica 10 de tiempo (segundos) vs kernel. Imagen 720 píxeles**



❖ **Gráfica 11 de tiempo (segundos) vs kernel. Imagen 1080 píxeles**



❖ **Gráfica 12 de tiempo (segundos) vs kernel. Imagen 4K píxeles**

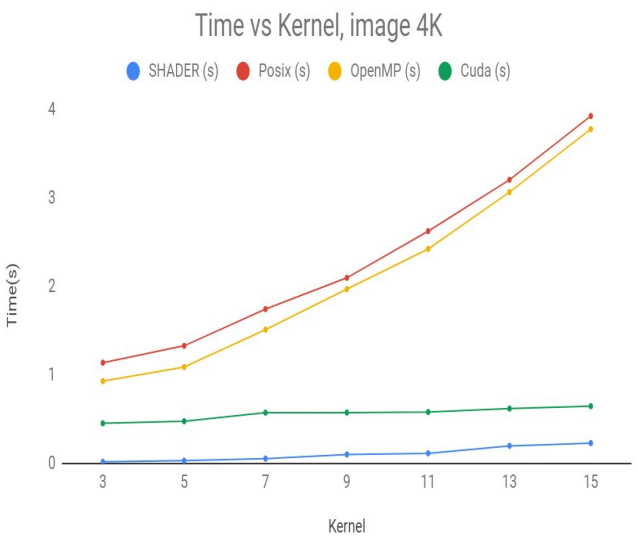


IV- C SOFTWARE VS SHADERS

Para esta gráfica se tomaron los mejores tiempos de cada herramienta de la implementación por software usando la imagen en 4K y se compararon con los tiempos de la implementación por shaders usando la imagen en 4K.

		15 hilos	15 hilos	995 hilos
Kernel	SHADER (s)	Posix (s)	OpenMP (s)	Cuda (s)
3	0,02	1,139	0,931	0,455
5	0,033	1,330	1,090	0,478
7	0,056	1,744	1,512	0,575
9	0,103	2,097	1,970	0,576
11	0,116	2,624	2,423	0,582
13	0,2	3,204	3,064	0,621
15	0,231	3,924	3,776	0,648

❖ **Gráfica 13 comparativo entre implementaciones, software (Posix,openMP,CUDA) vs shaders, de tiempo (segundos) vs kernel. Imagen 4k píxeles**



V. ANÁLISIS DE DATOS

Según las gráficas de los resultados anteriores se puede evidenciar que en la implementación por software el uso de una mayor cantidad de hilos para paralelizar la convolución disminuye el tiempo de ejecución del mismo, sin embargo se debe tener en cuenta cual es numero optimo de hilos con el cual se debe correr un programa para tener el mejor resultado y evitar el malgasto de recursos.

Una vez analizadas todas las gráficas se puede decir que el tiempo de ejecución del algoritmo va a ser proporcional al número del kernel (3,5,7,9,11,13,15) y a la calidad de la imagen (720 px,1080 px,4K px). Es por esta razón que en el numeral **V- C** se decidió realizar la comparación entre las implementaciones utilizando el kernel y la imagen que mayor tiempo de ejecución tenían (kernel = 15 y imagen 4K píxeles). Además en cuanto a las herramientas utilizadas en la implementación por software se tomaron los tiempos con el mayor número de hilos utilizados con cada herramienta,15 hilos para POSIX, 15 hilos para OpenMP y 995 hilos para CUDA.

Siendo así y analizando la gráfica 13 se puede ver claramente como la implementación por shaders mejora notablemente los tiempos de ejecución comparado con las herramientas (implementación por software) de OpenMP y Posix. En cuanto a la comparación entre CUDA y Shaders, sigue siendo mejores los tiempos de la implementación en shaders aunque la diferencia sea menor.

La implementación por software utilizando CUDA y la implementación por shaders tienen en común que ambas buscan optimizar la ejecución del algoritmo blur-effect por medio del uso de la GPU (unidad de

procesamiento gráfico). Es por esta razón que estas dos herramientas tienen los mejores tiempos de ejecución.
Sin embargo la complejidad de la programación del algoritmo en CUDA es más alta en comparación a la programación de la implementación por shaders utilizando processing y OpenGL. Esto ocurre porque CUDA está diseñado para la implementar cualquier algoritmo en paralelo (programación paralela) utilizando la GPU, luego es tarea del programador realizar el balanceo de carga con respecto a los hilos con que cuente la GPU, mientras que OpenGL se especializa en gráficos y no es necesario realizar el balanceo de carga por que OpenGL es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D, esto hace a la implementación por shaders más adecuada cuando se habla de tratamiento de gráficos.

VI. CONCLUSIÓN

El desarrollo del hardware gráfico ha ido haciendo cada vez más programables las tarjetas gráficas, de modo que los programadores puedan utilizar mejor sus funcionalidades. Con la introducción de los lenguajes de shaders se han podido implementar numerosos algoritmos (iluminación, técnicas de mapeo de texturas, efectos atmosféricos, entre otros) para que sean procesados al nivel de la tarjeta gráfica.
Los procesadores gráficos (GPU) son el corazón de las tarjetas gráficas modernas, los métodos de programación para tarjetas gráficas están basados en shaders, por ende los programas de GPUs deben ser altamente paralelos intentando aprovechar al máximo la capacidad de la GPU.
Los shaders al estar especializados en el tratamiento de todo lo que viene de la mano con un objeto virtual en un mundo bi o tridimensional utilizan de la mejor manera los recursos de la GPU , esto permite potenciar el procesamiento de imágenes al realizar tareas de manera más sencilla que utilizando una implementación por software .
Hablando desde el punto de vista del programador , los shaders permiten el procesamiento de gráficos de manera sencilla y amigable.Esto permite la realización de procesamientos de imágenes mucho más complejos.

REFERENCIAS

- https://www.fing.edu.uy/inco/cursos/cga/Clases/2018/EfectosDeEspacioDeImagenGabrielMadruga.pdf
- Shader# Programación de shaders en .NET DotNetMania64p.26-32 || Ludwig Leonard Miguel Katrib
- http://programandonet.com/questions/8649/webgl-glsl-como-funciona-un-shadertoy
- https://learnopengl.com/Getting-started/Shaders
- https://www.fayerwayer.com/2006/10/el-futuro-de-las-tarjetas-graficas-shaders-unificados//
- https://es.wikipedia.org/wiki/Paralelizacion´ automatica
- https://es.wikipedia.org/wiki/Algoritmo paralelo
- http://visualcomputing.github.io/Shaders/