PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA

FILTRO DE MEDIANA 2D DE ALTA VELOCIDAD EN PUNTO FLOTANTE ACELERADO CON GPUs

Tesis para optar al Título de **Ingeniero Electrónico**, que presenta el bachiller:

GABRIEL ALEJANDRO SALVADOR ROJAS

ASESOR: DR. ING.CESAR ALBERTO CARRANZA DE LA CRUZ

Lima, marzo de 2019

Resumen

El presente proyecto de tesis consiste en el diseño e implementación de un filtro de mediana 2D acelerado por GPU. El filtro es capaz de operar con valores en punto fijo y valores en punto flotante, además de hacer uso de un algoritmo que no cause divergencia permitiendo así que el desempeño de la ejecución del programa no disminuya.

El primer capítulo muestra los conceptos básicos del filtro de mediana 2D y de los GPUs. Se establece el contexto y la problemática que se busca resolver, se mencionan implementaciones previas como parte del estado del arte y se indican los objetivos a cumplir con la presente tesis.

En el segundo capítulo se establece la teoría de los métodos que se utilizaran para el desarrollo del filtro. Adicionalmente, se mencionan técnicas para aumentar el desempeño del programa de forma independiente al desarrollo del filtro.

En el tercer capítulo se establece el diseño del filtro a implementar, mencionando las consideraciones necesarias para la ejecución satisfactoria del programa y las técnicas utilizadas para aumentar el desempeño por medio del código.

En el cuarto capítulo se muestra los resultados tras la ejecución del filtro sobre un grupo de imágenes. Se comparan los resultados obtenidos por la presente implementación contra implementaciones previas.

Finalmente se muestra las conclusiones a las que se llegaron con la presente tesis, mencionando recomendaciones para añadir al trabajo realizado.

FACULTAD DE
CIENCIAS E INGENIERÍA

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

TEMA DE TESIS PARA OPTAR EL TÍTULO DE INGENIERO ELECTRÓNICO

Título	: Filtro de Mediana 2D de Alta Velocidad en Punto Flotante Acelerado con GPUs
Área	: Procesamiento Digital de Imágenes # 1416
Asesor	: Cesar Alberto Carranza de la Cruz
Alumno	: Gabriel Alejandro Salvador Rojas
Código	: 20121283
Fecha	: 13 de Noviembre del 2017

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
Facultad de Ciencias e Ingeniería
29 NOV 2017

Descripción y Objetivos

Un filtro de mediana en dos dimensiones (FM2D) es una técnica de procesamiento digital no lineal donde cada pixel de la imagen es reemplazado por la mediana aritmética de pixeles vecinos. Esta técnica es ampliamente utilizada para remover ruido impulsivo.

De forma natural, el FM2D es una operación que requiere cómputo intensivo, situación que ha sido favorecida con la aparición de unidades de procesamiento gráfico (GPUs). En los últimos 12 años se han realizado diversas nuevas implementaciones, así como mejoras en algoritmos ya existentes de filtros de mediana utilizando GPUs. Cada una de estas ha contribuido a resolver los problemas de desempeño ocasionados por las limitaciones tecnológicas del hardware, además de incidir en algoritmos que permita disminuir la complejidad computacional.

El presente trabajo de tesis muestra dos vertientes en el desarrollo de FM2D, los que operan sobre valores en punto fijo y los que operan sobre valores en punto flotante. Es importante resaltar que la gran mayoría de las mejoras se ha centrado en filtros de punto fijo, sin embargo, en diversas aplicaciones es necesario aplicar el FM2D sobre datos en punto flotante, donde aún no se ha explorado de manera amplia el diseño e implementación de filtros de mejor desempeño.

La presente tesis tiene por objetivo el diseño e implementación de un algoritmo paralelo de FM2D para el procesamiento de imágenes que contengan datos en punto fijo o flotante. Se empleará las capacidades de las arquitecturas de GPUs de NVIDIA y de la plataforma CUDA para maximizar el desempeño del algoritmo. Finalmente, se compararan los resultados del algoritmo desarrollado con implementaciones previas, cuantificando el desempeño utilizando la métrica de megapíxeles por segundo.

MÁXIMO 50 PÁGINAS

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE CIENCIAS E INGENIERÍA
MILLY CARRERA SORIA
Jefa de la Especialidad de Ingeniería Electrónica

FACULTAD DE
CIENCIAS E INGENIERÍA

PONTIFICIA **UNIVERSIDAD CATÓLICA** DEL PERÚ

TEMA DE TESIS PARA OPTAR EL TÍTULO DE INGENIERO ELECTRÓNICO

Título : Filtro de Mediana 2D de Alta Velocidad en Punto Flotante Acelerado con GPUs

Índice

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE CIENCIAS E INGENIERÍA

M. Sc. Ing. WILLY CARRERA SORIA
Coordinador de la Especialidad de Ingeniería Electrónica

Índice General

Índice de figuras

Índice de Tablas

CAPÍTULO 1

CONCEPTOS BÁSICOS Y MARCO PROBLEMÁTICO

1.1 Filtro de Mediana 2D

Un filtro de mediana es una técnica de procesamiento digital no lineal que realiza la operación de hallar la mediana aritmética en un arreglo de números. En el caso especial de la mediana 2D (figura 1.1), se realiza la operación sobre un arreglo bidimensional de *N x M* números, escogiendo un área de análisis de dimensión *L x L* alrededor de un único valor del arreglo, de modo que se realiza esta misma operación en todos los valores originales del arreglo original (*N x M veces)* y solo operando sobre los números originales del arreglo.

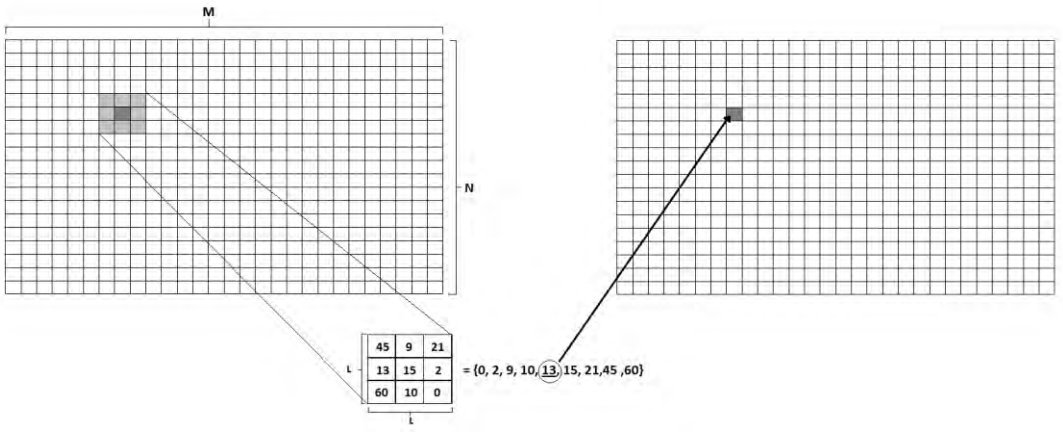


Figura 1.1 Aplicación del filtro de mediana 2D.

El filtro de mediana 2D es un filtro usado en el procesamiento de imágenes [1], generalmente utilizado para tratar de eliminar ruido sobre la imagen. En el ámbito

médico, el filtro de mediana 2D es utilizado para eliminar el ruido de tipo impulsivo (salt and pepper) [1].

El filtro de mediana 2D por su propia naturaleza es altamente paralelizable, lo cual ha ocasionado que se realizaran varias implementaciones del filtro usando FPGAs, GPUs, y CPUs al aprovechar las capacidades multinucleo de estos últimos. En el caso particular de los GPUs, los algoritmos tradicionales de los filtros de mediana 2D hacen un uso considerable de instrucciones condicionales las cuales ocasionan ramificaciones en el código. En consecuencia, se ocasiona divergencia de instrucciones en el GPU lo cual provoca que la ejecución del código se serialice, provocando una pérdida de desempeño en el GPU.

1.2 GPUs y CUDA

1.2.1 GPU

La GPU (Graphics Processor Unit) es un elemento de hardware que actúa como coprocesador dedicado especialmente al procesamiento de gráficos y operaciones de punto flotante, dejando que el CPU se encargue de otro tipo de procesos. A diferencia de un CPU, el GPU está conformado por una cantidad inmensamente superior de núcleos de procesamiento, lo cual permite que la paralelización sobre ciertos tipos de proceso se realice mucho más eficientemente utilizando un GPU que un CPU [2].

Actualmente una GPU se puede encontrar en una amplia gama de dispositivos, de los cuales las más populares son las tarjetas gráficas para ordenador. Debido al constante avance tecnológico las tarjetas gráficas son renovadas año tras año, lo que ocasiona que cada vez se convierta en un producto más asequible.

Debido a la especialización en operaciones en punto flotante, la fácil accesibilidad y la superioridad en cantidad de núcleos respecto a los CPUs convencionales, los GPUs (en su forma de tarjeta gráfica) se popularizan cada vez más como una herramienta indispensable en diferentes campos de la ciencia e ingeniería, permitiendo reducir considerablemente el tiempo que se demora en realizar una tarea computacional que requiera operaciones para las cuales el GPU pueda marcar una mejora desempeño respecto a otras alternativas.

1.2.2 CUDA

CUDA (lanzado su versión 1.0 en 2007) [3] significa *Compute Unified Device Architecture* (Arquitectura Unificada de Dispositivos de Cómputo). CUDA es una

arquitectura para el cómputo paralelo creado por NVIDIA, que aprovecha el aumento en potencia de cálculo de los GPUs fabricados por la misma para incrementar el desempeño de los programas [2].

CUDA proporciona extensiones de C y C++, las cuales le permiten implementar el paralelismo en el procesamiento de datos y tareas mediante lenguajes de programación de alto nivel [2]. Actualmente en los sistemas operativos modernos la GPU no actúa únicamente como procesador gráfico, sino como procesador paralelo de propósito general accesible para cualquier aplicación.

Todas las herramientas y ayudas que proporciona la plataforma CUDA vienen dada mediante la aplicación *CUDA Toolkit* [4],el cual es un completamente gratuito y distribuido por la propia NVIDIA. El requisito indispensable para hacer uso del *CUDA Toolkit* es el tener instalado una GPU GeForce, ION, Quadro o Tesla, pudiendo ejecutarse las aplicaciones hechas en CUDA en todas las versiones y modelos de estas GPUs sin tener que realizar cambios en el código del programa en la gran mayoría de los casos.

1.3 Contexto

Como se explicó en la sección 1.2.1, las GPUs han sido utilizadas por varios años para la aceleración del procesamiento y tareas computacionales. Uno de los campos beneficiados por el uso del GPU es la medicina, donde es común encontrar que la recolección de datos requiere una gran capacidad de cómputo. En particular, el procesamiento de imágenes de la retina para la detección de retinopatía diabética, las cuales son imágenes con datos en punto flotante, al utilizar descomposición AM-FM requiere gran cantidad de poder de procesamiento de datos [5].

El trabajo desarrollado en [5] muestra una reducción dramática en el tiempo de procesamiento durante la descomposición AM-FM al utilizar un GPU. Sin embargo, posterior al análisis AM-FM se requiere un filtrado de mediana que no se implementó con GPUs, pues por las limitaciones tecnológicas de la época no se pudo obtener un resultado satisfactorio que valla en conjunto con el aumento de desempeño que obtuvieron el resto de operaciones explicadas en el artículo. Gracias a la evolución de la arquitectura de los GPUs por parte de NVIDIA, el rendimiento de los GPUs se espera que sea superior. Esto en conjunto con un adecuado algoritmo paralelizable abre la puerta a una posible implementación rápida del filtro de mediana. Por lo que se busca que el filtro de mediana a implementar durante la presente tesis tenga la

capacidad de operar de forma rápida sobre imágenes las cuales tengan datos en punto flotante (tipo *float)*.

1.4 Estado del Arte

El filtro de mediana 2D existe desde hace muchos años, sin embargo no ha sido hasta la última década desde que se empezaron a diseñar filtros de mediana usando las facilidades que ofrecen los GPUs, más específicamente utilizando la plataforma CUDA para el desarrollo e implementación del algoritmo.

En estos últimos 12 años se han realizado diferentes implementaciones nuevas y mejoras sobre algoritmos ya existentes de filtros de mediana, cada una de ellas tratando de resolver los problemas de rendimiento ocasionada por las limitaciones tecnológicas del hardware existente en su respectivo año de implementación o tratando de mejorar el desempeño por medio de un algoritmo que le permita disminuir el tiempo de procesamiento de datos.

1.4.1 Branchless Vectorized Median Filter

Filtro de mediana vectorizado sin ramificaciones o BVM (Branchless Vectorized Median Filter), el cual propone un algoritmo de ordenamiento tal que evita las ramificaciones en el código [6], la cual es de las principales causantes de perdida de desempeño en el GPU [7], sin embargo al aumentar el tamaño del área de análisis a valores mayores de 3 x 3, el desempeño se reduce a valores 4 veces inferiores, siendo una opción no muy atractiva para áreas de análisis mayores.

1.4.2 Constant Time Median Filter

El filtro de mediana de tiempo constante o CTMF plantea el uso de un algoritmo tal que la cantidad de operaciones para realizar el ordenamiento de los números a analizar sea constante [1], que en consecuencia ocasiona que tenga la complejidad computacional más baja. Sin embargo en la implementación real del filtro, su desempeño constante es superado ampliamente en los tamaños de áreas de análisis más bajos.

1.4.3 Parallel Ccdf-based Median Filter

Filtro de mediana paralelo basado en CCDF o PCMF es un filtro de mediana que usa el ordenamiento de números tipo CCDF (Complementary Cumulative Distribution Function) [8][9], el cual permite en el mejor de los casos tener una complejidad

computacional constante, pero teniendo velocidades 7 veces mayores que el CTMF [8]. No obstante, este algoritmo de ordenamiento está limitado a ordenar valores dentro de un intervalo discreto y finito.

1.4.4 Parallel Register-only Median Filter

Filtro de mediana paralelo de solo registro o PRMF es un filtro de mediana cuyo principal método para incrementar el desempeño y reducir los tiempos es trabajar sobre la transferencias de datos del CPU al GPU y viceversa al “ocultar” las latencias provocadas por las instrucciones de acceso a memoria [10]. Sin embargo el trabajo realizado en el artículo se enfoca sobre imágenes con datos de 8 bits y 16 bits, además de realizar sus pruebas sobre imágenes cuyo número de pixeles verticales y horizontales sea múltiplo de 512 [10].

Analizando los filtros de mediana durante la última década se llegó a la conclusión de que existen dos vertientes en lo que se refiere al tipo de algoritmo de ordenamiento de números, las cuales son las que ordenan valores de tipo enteros y las que ordenan valores de punto flotante (también pueden ordenar valores de tipo entero) por lo cual la presente tesis se enfocara en un filtro de mediana que trabaje con valores de punto flotante ya que con ese tipo de valores trabajan las imágenes de retinopatía diabética.

1.5 Objetivos

En la presente tesis se busca el diseño e implementación de un algoritmo de filtro de mediana 2D que sirva para el procesamiento de imágenes que contengan datos en punto flotante utilizando las capacidades de las GPU de NVIDIA y la plataforma CUDA para maximizar el desempeño de nuestro algoritmo.

Objetivos específicos:

- Crear un filtro de mediana 2D que funcione para datos en punto flotante y en punto fijo.

- Establecer una comparación del algoritmo propuesto contra algoritmos previos, al compararlos usando el criterio de *Megapíxeles por segundo vs Área de análisis* y cuantificar los resultados.

CAPÍTULO 2

TEORÍA DE MÉTODOS UTILIZADOS

Para el diseño e implementación de un filtro de mediana 2D una de las partes más críticas que determinarán su desempeño en velocidad es el algoritmo que se usa para hallar la mediana en el área de análisis [11], el cual es en la mayoría de casos un algoritmo de ordenamiento numérico. En el presente capítulo se explicará la teoría básica sobre el tipo de técnica que se va a usar (redes de ordenamiento y métodos que las contengan), así como el funcionamiento del algoritmo especifico escogido para la implementación del presente filtro, justificando su elección. Adicionalmente, se tratarán los conceptos de programación en paralelo, específicamente para el caso de CUDA y los GPUs de NVIDIA, y se describirán métodos adicionales para mejorar el desempeño en velocidad de ejecución del programa.

2.1 Redes de Ordenamiento

2.1.1 Concepto Básico

Las redes de ordenamiento (Sorting Networks en inglés) o redes de comparación (Comparator Networks en inglés), son la agrupación de comparadores de forma concurrente tal que cada comparador tenga dos datos de entrada. Cada operación de comparación puede ser realizada de forma paralela si las entradas de estos no dependen de salidas de comparadores anteriores. La diferencia principal entre las redes de ordenamiento y otras agrupaciones de comparadores es que una red de ordenamiento funciona para cualquier combinación aleatoria de valores de entrada y que el número de pasos para ordenar los elementos de entrada siempre es el mismo, siempre y cuando se utilice el mismo algoritmo y el número de elementos de entrada no cambie [12]. En resumen, para ordenar N elementos de entrada siempre se necesitará M números de comparaciones para asegurar de manera inequívoca que el conjunto de elementos de entrada está ordenado.

Las redes de ordenamiento generalmente son representadas como conexiones entre cables o flechas entre líneas, indicando para cada distinto algoritmo en particular cuales posiciones de números hay que comparar para poder ordenar el conjunto tal como se muestra en la figura 2.1.

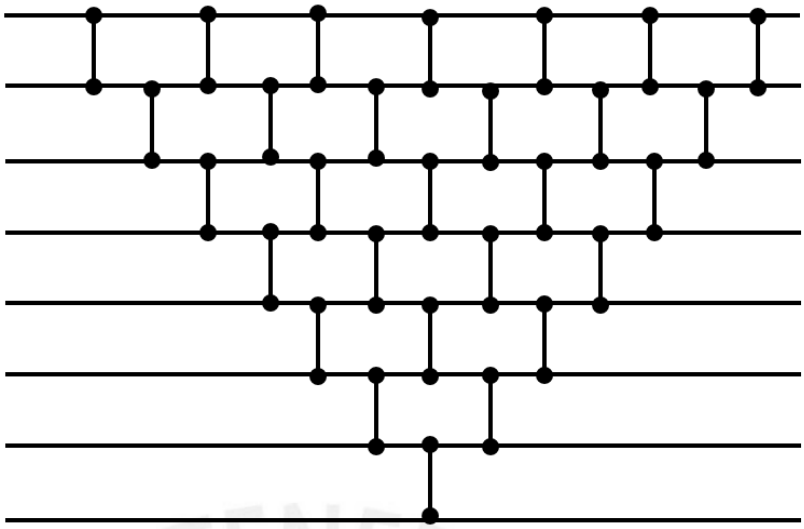


Figura 2.1 Representación gráfica de red de ordenamiento.

2.1.2 El Elemento de Comparación

El comparador (figura 2.2, 2.3 y 2.4) es el elemento básico que conforma toda red de ordenamiento. Cada comparación ordena los dos datos de entrada (a, b) al realizar las operaciones *máximo* y *mínimo*, obteniendo a la salida del comparador Max(a, b) y Min(a, b) ordenadas de forma ascende nte o descendente según se desee [13].

Figura 2.2 Bloque comparador.

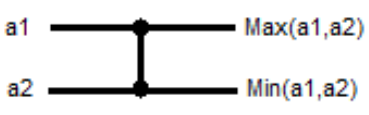


Figura 2.3 Comparador con cables

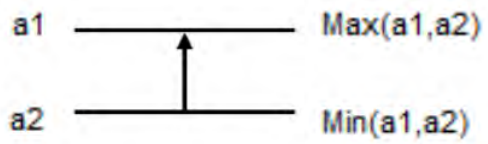


Figura 2.4 Comparador usando flechas

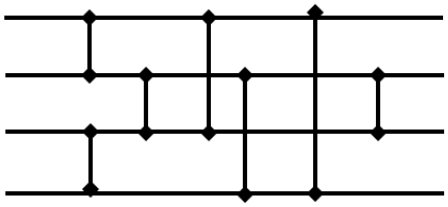


Figura 2.5 Red de ordenamiento con cables.

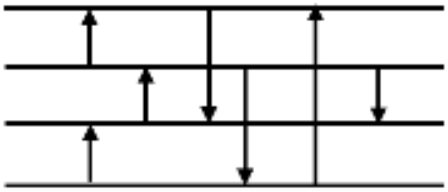


Figura 2.6 Red de ordenamiento con flechas.

En el caso de la representación con cables todas las comparaciones tienen el mismo sentido.

La independencia de operación entre cada comparador y la simplicidad de las operaciones que lo conforman han hecho que las redes de ordenamiento (figura 2.5 y 2.6) sean opciones muy a tener en cuenta en cualquier tarea computacional que requiera del ordenamiento de datos para realizar un proceso mayor como los explicados en [14].

2.2 Selección del Algoritmo

Existen varios algoritmos de ordenamiento de datos. Sin embargo, desde hace años se conocen unos cuantos algoritmos que demostraron tener un desempeño superior al resto de redes de ordenamiento. Estas redes son el ordenamiento bitonico (figura 2.7) y el ordenamiento de unió n par-impar de Batcher (Batcher´s Odd-even Merging Sort, figura 2.8) [13].

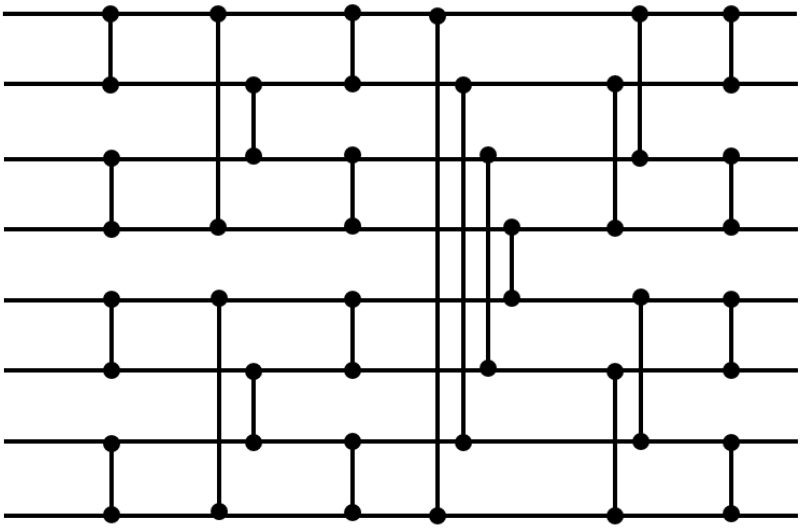


Figura 2.7 Ordenamiento bitonico para 8 entradas.

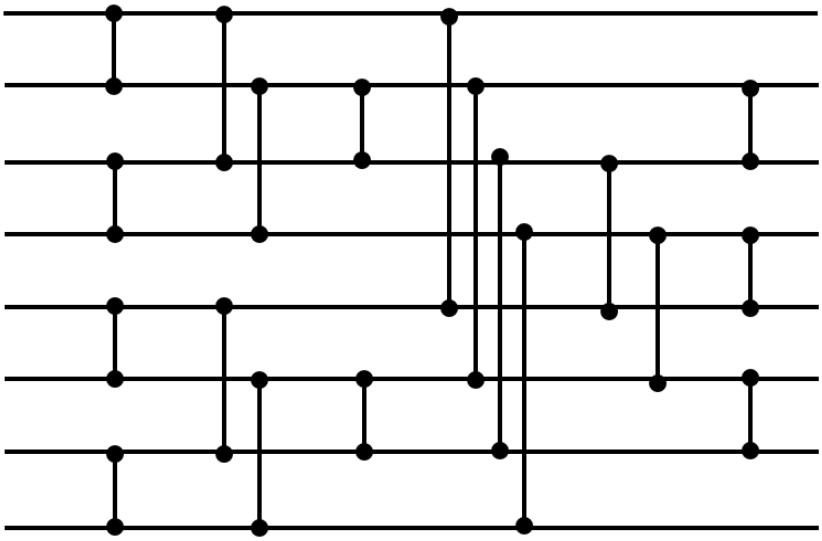


Figura 2.8 Ordenamiento de unión par-impar de Batcher para 8 entradas.

Sin embargo estas redes fueron originalmente diseñadas para ordenar una cantidad de elementos tal que sean una potencia de dos [14]. Aunque actualmente existen varios métodos para aplicar los ordenamientos antes mencionados para una cantidad cualquiera N de números, siendo N entero mayor a 0, aun algunos de estos métodos requieren un ordenamiento previo específico en los valores de entrada o la omisión de un número determinado de comparadores según sea el caso, aun siendo la profundidad de la red el de una red de ordenamiento para la siguiente cantidad mayor de elementos que sea potencia de dos [14].

Por los inconvenientes mencionados en el párrafo anterior, se optó por escoger algoritmos los cuales tengan como objetivo principal la obtención de la mediana de los elementos de entrada. Los algoritmos utilizados para realizar el ordenamiento en la implementación del filtro son el median_odd_N y la selección olvidadiza, ya que entre sus características se encuentran el estar diseñado con el objetivo principal de hallar la mediana dado un área de análisis dado, además de que ambos se pueden adaptar para que utilicen redes de ordenamiento de menor tamaño como parte del algoritmo principal, y por consiguiente se conserven las propiedades de estabilidad de las redes de ordenamiento.

2.2.1 Algoritmo median_odd_N

Este algoritmo consiste en obtener la mediana aritmética de un área de análisis de *N x N* elementos de entrada, siendo N un número impar (figura 2.9). El primer paso consiste en ordenar las columnas del área de análisis de forma creciente de abajo hacia arriba. A continuación, se realiza el ordenamiento de las filas de forma ascendente en sentido de derecha a izquierda. Finalmente se ordenan las diagonales con pendiente k de forma consecutiva, tal que k empieza en 1 y termina en *(N-1)/2,* de modo que el valor que quedara en el centro del área de análisis será el valor de la mediana aritmética[15].

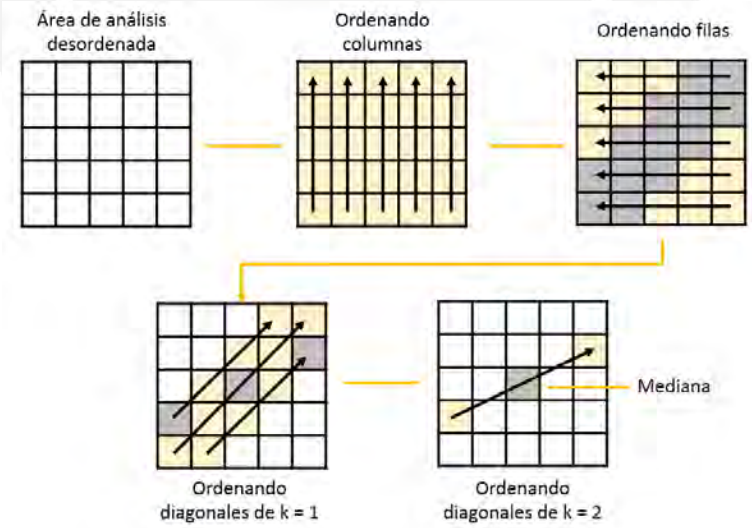


Figura 2.9 median_odd_N para N = 5.

Una de las características del median_odd_N es que para valores de N pequeños no es necesario ejecutar el ordenamiento completo para hallar la mediana, siendo

únicamente necesario encontrar valores extremos (máximos o mínimos) o valores próximos a estos durante el ordenamiento de las filas y diagonales (figura 2.10). Al ejecutar una cantidad menor de operaciones de comparación se tiene como consecuencia una disminución en el tiempo de ejecución[15].

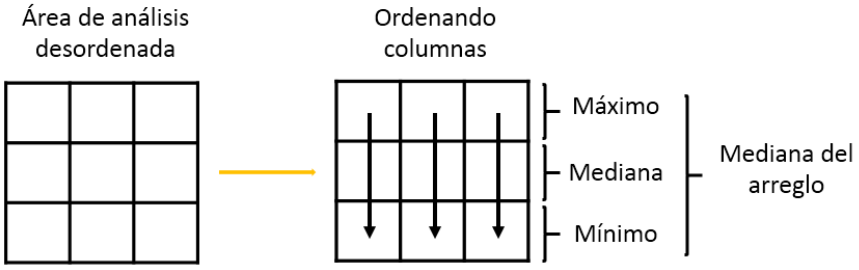


Figura 2.10 median_odd_N para N = 3.

Aunque es posible extender este algoritmo para cualquier valor de N, ya sea par o impar, en [15] se indica que para valores altos de N la cantidad de operaciones de comparación de esta red de ordenamiento terminara siendo mayor que la de otras redes de ordenamiento ya conocidas, por lo cual este algoritmo solo será eficiente con N siendo un valor pequeño. Además, no se especifica el tipo de algoritmo se usa para el ordenamiento de las filas, columnas y diagonales, por lo cual el ordenamiento usado para la ejecución del median_odd_N también influirá en la cantidad de operaciones de totales de comparación del algoritmo final.

2.2.2 Método de Selección Olvidadiza

El método de selección olvidadiza (figura 2.11) es una técnica enfocada en hallar la mediana de un conjunto de números. Este método consiste en escoger los primeros *N* elementos de una cantidad *C* impar de números desordenados, de modo que *N* sea igual a *(C + 3) / 2*. A continuación, se halla el máximo y mínimo de los *N* elementos seleccionados para retirarlos. Prosiguiendo, se agrega a los números restantes de los del paso anterior otro elemento del conjunto *C* que no se encontrara entre los *N* números escogidos inicialmente. Estos últimos dos pasos se repiten hasta que solo quede un numero sin retirar, el cual será la mediana aritmética del conjunto *C* de números iniciales [10].

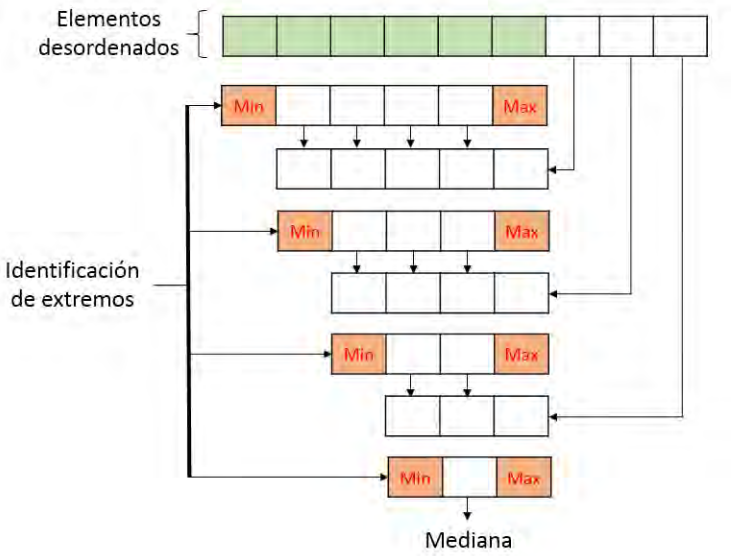


Figura 2.11 Selección olvidadiza para 9 elementos.

Al igual que el algoritmo presentado en la Sección 2.2.1, este método está enfocado específicamente para hallar la mediana. Se tiene que mencionar que esta técnica no especifica el método para hallar el máximo y el mínimo cada vez que se identifican extremos, por lo cual el desempeño recaerá principalmente en el algoritmo utilizado para este propósito. Para la implementación de la presente tesis se usará una secuencia incompleta de una red de ordenamiento para hallar los extremos, ya de esa manera se conservará la predictibilidad al trabajar con este método.

Se escogió este método ya que a diferencia de las redes de ordenamiento convencionales se obtiene una mayor flexibilidad para trabajar con los elementos a ordenar. Además, conforme avanza el algoritmo cada vez es necesario trabajar con una cantidad menor de números.

2.3 Programación en Paralelo en CUDA

CUDA C le permite al programador crear sus propias funciones llamadas kernels. Los kernels son funciones que se ejecutarán M veces en paralelo en una cantidad M de hilos de CUDA diferentes, siendo M un numero elegido por el programador, pero a la vez delimitado por la capacidad del GPU que se use.

En CUDA los hilos están agrupados en bloques, teniendo tanto los bloques como los hilos su propio identificador (figura 2.12). El identificador le permite al programador llamar a la cantidad de hilos que necesite para realizar su programa (limitado por la capacidad del GPU). En los GPUs actuales se tienen hasta 1024 hilos por bloque. Usando el identificador de bloques se puede llamar a una cantidad deseada de bloques. Por lo tanto la cantidad de hilos máximos a usar es igual a la cantidad máxima de hilos por bloques multiplicada por la cantidad máxima de bloques del GPU (millones o billones de hilos dependiendo del dispositivo) [16].

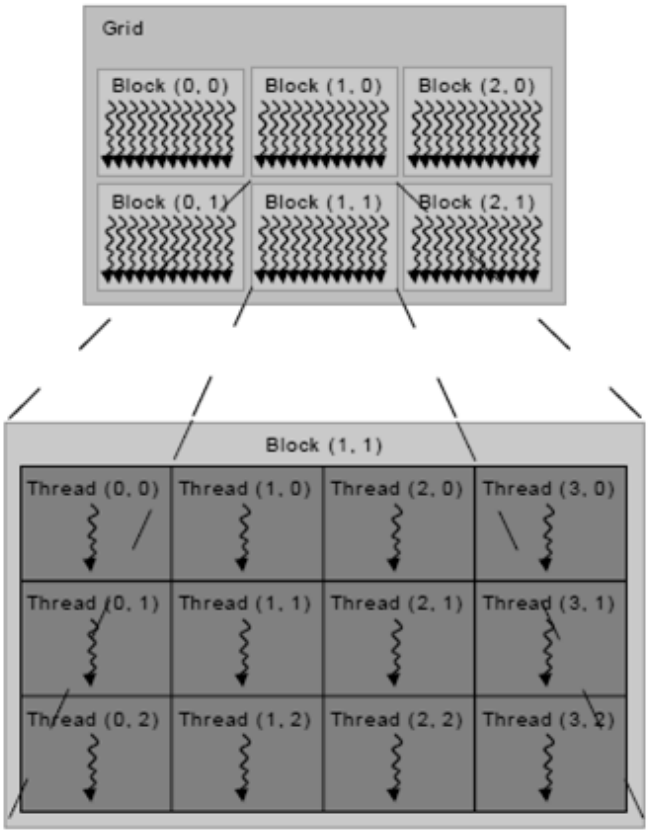


Figura 2.12 Representación gráfica de los hilos y bloques [16].

Los hilos al interior de cada bloque pueden cooperar compartiendo información ya que tienen acceso a una memoria compartida. Estos hilos por defecto sincronizan su ejecución para coordinar su acceso a memoria, pero en caso de posibilidad de desincronismo el operario puede llamar en el *kernel* a la función específica (_syncthreads( )) para recuperar el sincronismo [16].

La jerarquía de acceso a memoria compartida en los GPU no es global, distribuyéndose como se explica en la Figura 2.13.

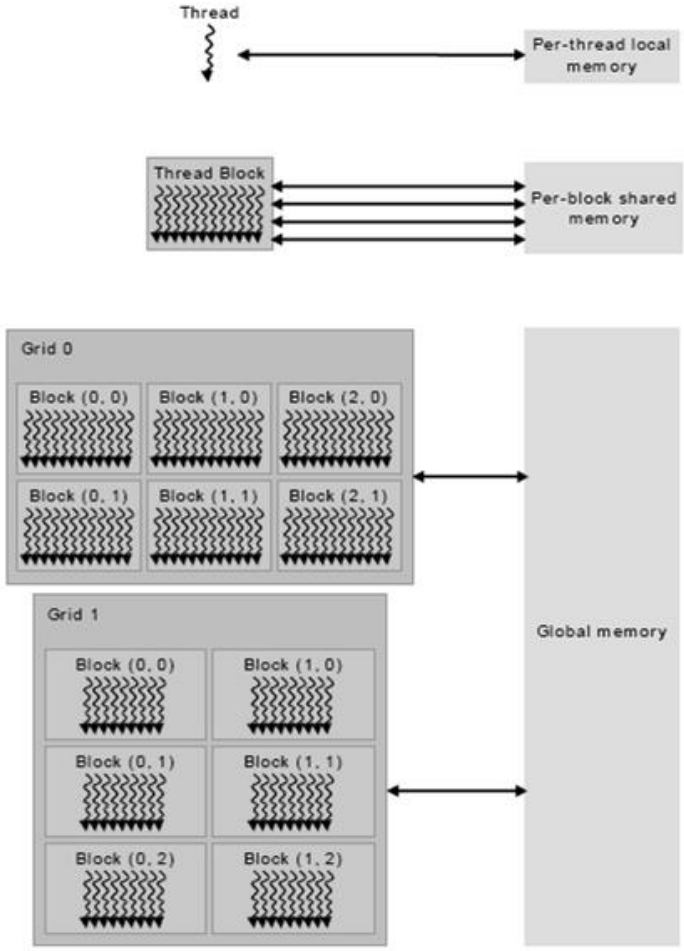


Figura 2.13 Diagrama de jerarquía de memoria en el GPU [16].

El modelo de jerarquía puede ser visto con más detalles en [16].

El modelo de programación de CUDA asume que los hilos son ejecutados por otro dispositivo físico, por lo tanto, mientras que los kernels son operados por el GPU, el resto del programa es procesado por el CPU. El GPU ejecuta a los kernels en su propia memoria, la cual se espera que sea más rápida que la memoria global [16].

2.4 Técnicas para aumentar el desempeño

- **Mejorar los accesos a la memoria:**
  Se adecuará el número de accesos a los diferentes niveles de memoria, como los necesarios para el alineamiento de los datos (coalescencia de memoria), con el objetivo de disminuir el tiempo que toman los accesos a memoria en el programa.

- **Realizar un correcto uso de los núcleos del GPU (carga de trabajo):**
  Se establecerá un adecuado balance entre el número de hilos por bloque y los recursos de cada bloque de núcleos (streaming multiprocessors) según el GPU específico con el que se implementará el programa.

- **Desenrollado de Bucles ("Loop unrolling"):**
  Se intentará disminuir la velocidad de ejecución del programa eliminando o reduciendo las instrucciones que controlan los bucles (operaciones iterativas), acosta de aumentar la cantidad de código escrito para realizar las operaciones dentro del bucle. Ejemplo (tabla 1):

Bucle normal	Bucle desenrollado

for (i=0;i<5;i++)
arreglo[i] = a[i]+b[i];
	suma[1] = a[1]+b[1];
	suma[2] = a[2]+b[2];
	suma[3] = a[3]+b[3];
	suma[4] = a[4]+b[4];
	suma[5] = a[5]+b[5];

Tabla 1

- **Utilizar instrucciones diseñadas de forma específicas para operar en el kernel**:
  Instrucciones condicionales que no generen divergencia.

CAPÍTULO 3

DISEÑO, IMPLEMENTACION Y CONSIDERACIONES PREVIAS

3.1 Consideraciones Previas

3.1.1 Bordes de la Imagen

Al realizar la operación de filtrado de mediana 2D en una imagen se tiene que considerar la existencia de fronteras en la imagen, ya que en dicha zona el área de análisis estaría considerando valores inexistentes o externos a la imagen para realizar la operación de mediana (figura 3.1).

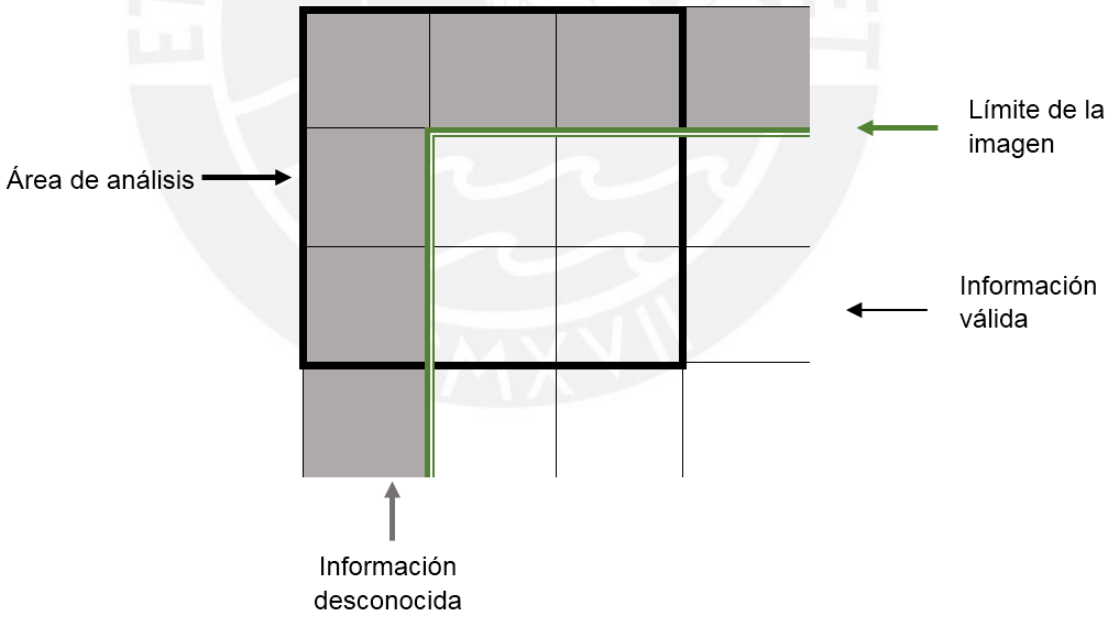


Figura 3.1 Datos en las fronteras de la Imagen.

Un método usado para solventar este problema es el llenado de la información fuera de los límites con valores previamente establecidos antes de someter a la imagen al filtro de mediana. Los valores utilizados para rellenar los bordes pueden

tener relación con la información de la imagen, como los usados en [17], o un valor constante para todos los bordes al igual que en [18], la elección del tipo de información afuera de los bordes dependerá de la imagen a analizar.

La profundidad de los bordes a tomar en cuenta depende del tamaño del área de análisis utilizado en la operación a realizar. El valor de la profundidad se puede calcular realizando la operación *k = (L - 1) / 2,* donde k es el tamaño de la profundidad de los bordes y L la longitud de lado de área de análisis (figura 3.2).

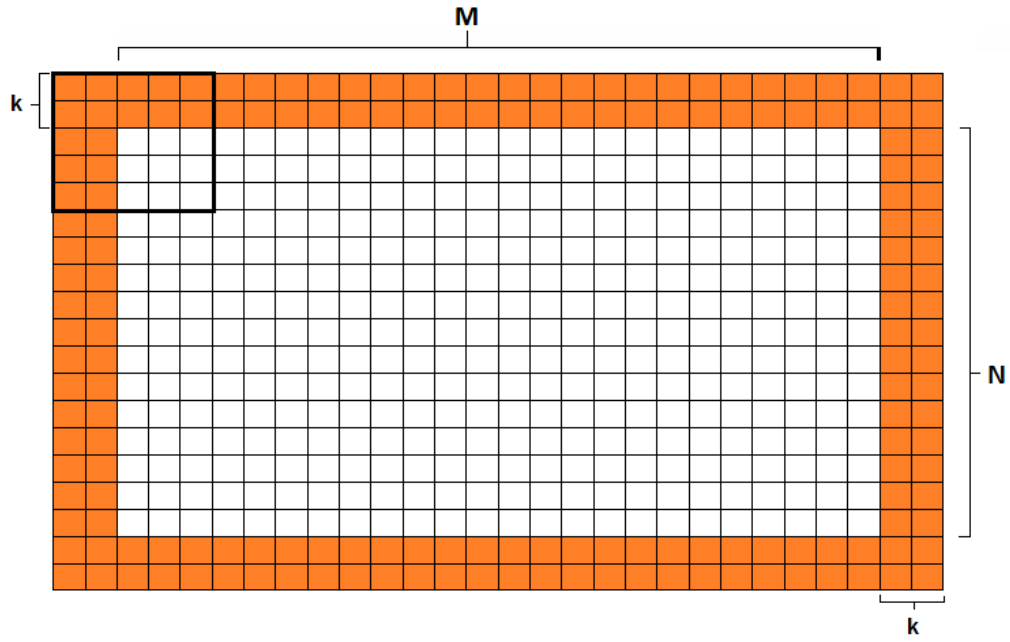


Figura 3.2 Rellenado de datos para un L = 5.

3.1.2 Mapeado de Pixeles

Para obtener los valores necesarios para realizar el proceso de filtro de mediana 2D sobre una imagen es necesario tener acceso al valor y la ubicación de cada uno de sus pixeles. Para ello se mapea las posiciones de los pixeles para poder moverse libremente sobre la imagen. Algunas implementaciones tienen como valor de referencia para conocer la ubicación en la imagen y obtener los valores del área de análisis el pixel central del área de análisis, como en [17] o en la Figura 1.1. Para la implementación propuesta en esta tesis se usará como referencia el pixel de la esquina superior derecha del área de análisis para ubicarse sobre la imagen y obtener los valores de los pixeles a analizar. El proceso para obtener la mediana y ubicarla en la posición adecuada se realizará tal como se muestra en la Figura 3.3.

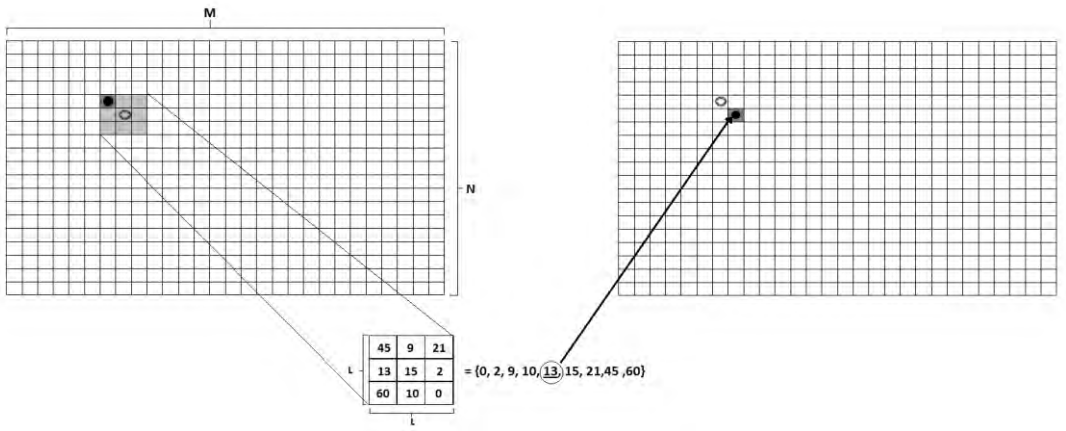


Figura 3.3 Filtro de mediana 2D con pixel superior derecho como pivote.

3.2 Diseño

3.2.1 Algoritmos para el Cálculo de la Mediana

En la sección 2.2 se propuso el uso del algoritmo median_odd_N y la selección olvidadiza para hallar la mediana en la implementación del filtro. Al tener en cuenta las limitaciones que poseen y la dependencia sobre un algoritmo de ordenamiento adicional no especificado, se decidió usar el median_odd_N para áreas de análisis pequeñas y el método de selección olvidadiza para áreas de análisis más grandes.

Este conjunto de algoritmos que utilizan redes de ordenamiento en diferentes medidas se realizara en tamaños de área de análisis de *3 x 3* con el median_odd_N, y de *5 x 5* o superiores con la selección olvidadiza.

3.2.2 Algoritmos Paralelos para el Filtrado de la Imagen

Para el diseño de algoritmos paralelos existe la falta de una metodología bien definida [19]. Sin embargo, existen varias técnicas para paralelizar un programa que cubren un amplio espectro de problemas, tales como canalización (pipelining), salto de puntero (pointer jumping), divide y conquista, árboles balanceados, particionamiento, etc [19].

Para la implementación del filtro de mediana 2D en la presente tesis se usará la técnica de particionamiento.

La estrategia de particionamiento consiste en separar el problema principal, el cual en este caso será la ejecución del filtro de mediana sobre la imagen, en subproblemas más pequeños del mismo tamaño e independientes, con el objetivo de resolverlas de forma paralela [19]. Las operaciones previas de preparación de información y operaciones de transferencia de datos se realizarán de manera secuencial.

Se plantean dos formas de particionamiento:

a) Por pixel: Dividir la totalidad de la imagen en áreas de análisis traslapados e independientes, donde cada área da como resultado la mediana del pixel (ver figura 3.4). Se decidió esta forma de particionamiento ya que para un filtro de mediana las áreas de análisis tendrán igual tamaño, lo que permite que al usar las redes de ordenamiento las operaciones necesarias para hallar la mediana se realizaran siempre en el mismo número de pasos, lo que evita divergencia en la ejecución del programa.

Figura 3.4 Diagrama de ejecución del ordenamiento de un área de análisis de *3x3.*

En la Figura 3.5 se muestra un ejemplo de la forma de particionamiento.

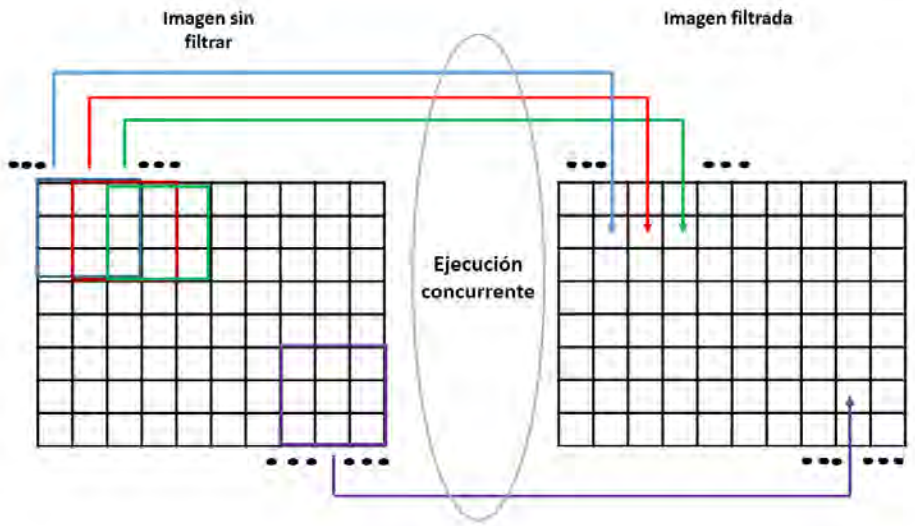


Figura 3.5 Particionamiento en filtro de mediana 2D.

b) Por grupo de 4 pixeles: Para procesar un área de análisis en el filtro de mediana 2D es necesario tomar los pixeles adyacentes al pixel central el cual se desea remplazar, tal como se explicó en el capítulo 1. Al procesar pixeles contiguos, tanto vertical como horizontalmente, en la ejecución del filtro se puede apreciar que las áreas de análisis comparten una cantidad de pixeles comunes. La implementación del filtro PRMF (sección 1.4.4) plantea la ejecución de dos vecindades de pixeles por cada hilo con el objetivo de reducir la latencia en las operaciones de acceso a la memoria [10]. El algoritmo PRMF aprovecha el área conjunta de pixeles que comparten ambas áreas de análisis para realizar una única operación para el área común, la cual puede ser aprovechada por ambas vecindades de pixeles, reduciendo el número total de operaciones a realizar para procesar 2 pixeles [10].

   Para la implementación del presente filtro, se propone el procesamiento de 4 pixeles por hilo (figura 3.6), expandiendo el trabajo realizado en [10] al aprovechar la gran cantidad de pixeles del área común que hace posible el uso del método de selección olvidadiza.

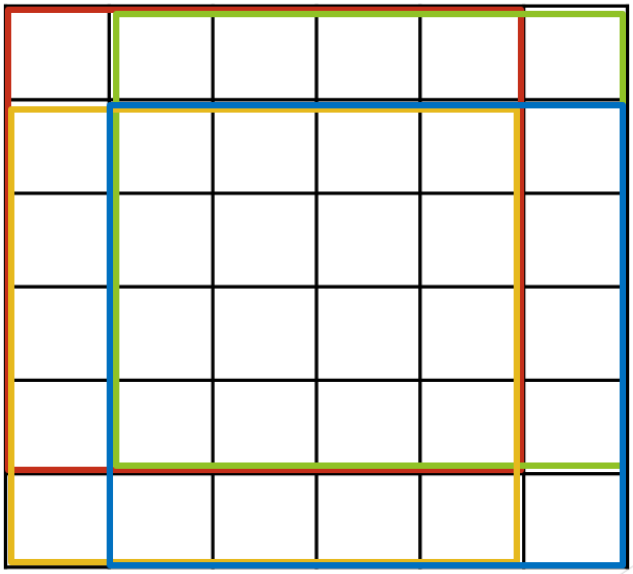


Figura 3.6 Cuatro ventanas de *5x5* en un arreglo de *6x6*.

El procesamiento de 4 pixeles por hilo se ejecutará para ventanas de *5 x 5* o mayores, ya que para un área de análisis de *3 x 3* la cantidad de pixeles comunes que comparten 4 ventanas del mismo tamaño no es lo suficientemente grande para ejecutar la técnica de selección olvidadiza (sección 2.2.2).

En la figura 3.7 se muestra el diagrama de ejecución al procesar 4 pixeles por hilo.

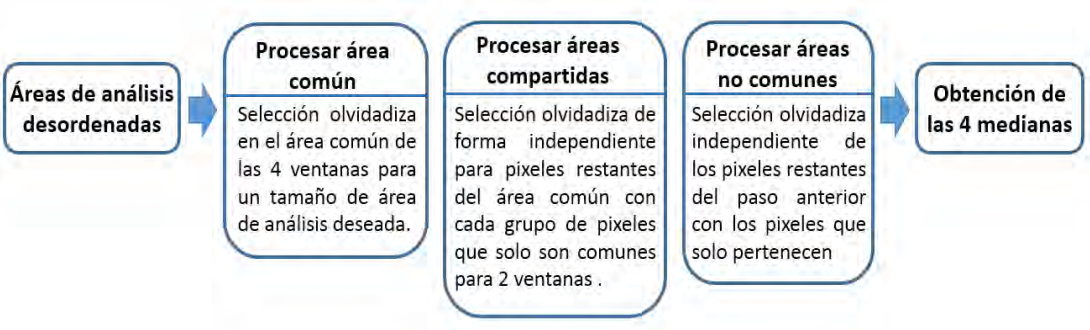


Figura 3.7 Diagrama de ejecución para procesar 4 pixeles.

3.2.3 Modelo del programa

El programa a implementar hará uso tanto de recursos del GPU como del CPU (ver figura 3.8). Como se mencionó en la sección 3.2.2 se busca que solo la operación que tenga que ser paralelizada sea únicamente la que se refiere a la ejecución del filtro de mediana 2D en sí, dejando que el CPU se haga cargo reste de operaciones (transferencia de datos, lectura y escritura de la imagen, llenado de bordes, etc). Las operaciones realizadas en el GPU serán las partes aritméticas para el ordenamiento del filtro de mediana y el llenado del área de análisis mediante el mapeo de pixeles de la imagen original.

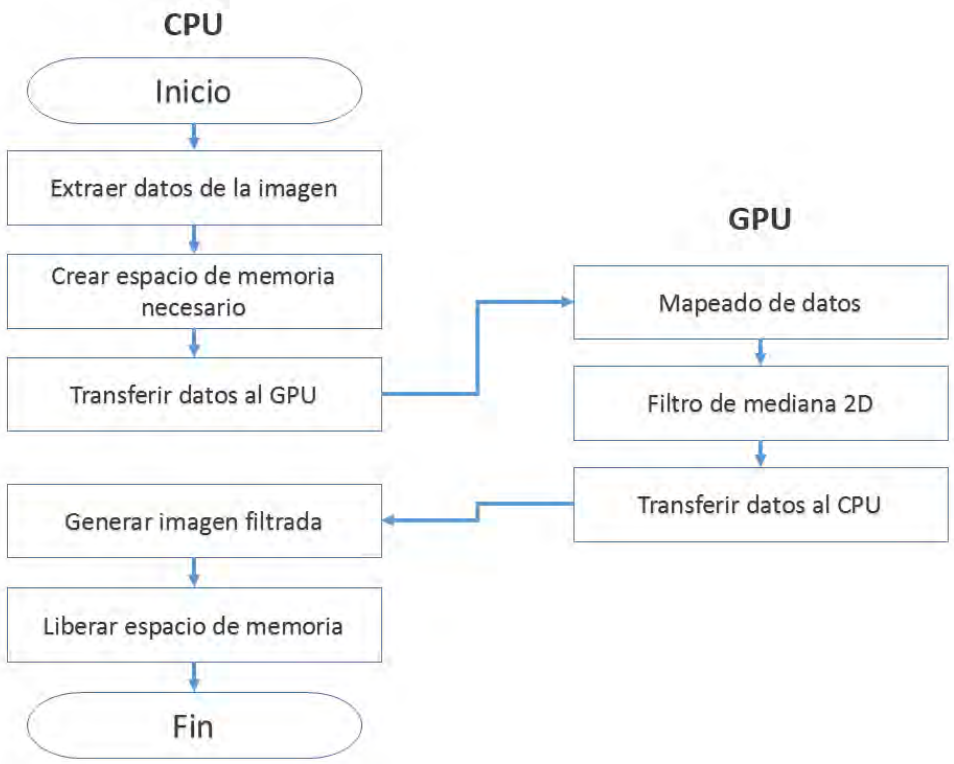


Figura 3.8 Procesos del CPU y del GPU.

3.3 Técnicas para Mejorar el Desempeño en Código

Adicionalmente al modelado del diseño del algoritmo, es necesario el uso de técnicas que permitan el aumento del desempeño en el GPU, ya que este dispositivo procesa los datos y ejecuta las instrucciones de forma diferente a como lo hace el

CPU. Las técnicas, propuestas en la sección 2.4, incrementaran el rendimiento en tiempo de ejecución del programa de forma adicional al propio incremento ocasionado por el uso de una mayor cantidad de núcleos por parte del GPU.

3.3.1 Accesos a Memoria utilizando máximo tamaño de Cache L1

Tal como se pudo apreciar en la Figura 2.12 de la sección 2.3, los hilos del GPU primero buscan la información en la memoria local (cache) y en la memoria compartida, en caso de no encontrar los datos en dichas memorias se busca en la memoria global del GPU. Se es recomendado por [16] el uso de las memorias más altas en la jerarquía para realizar las operaciones de carga y almacenamiento de datos, debido a que su ancho de banda en lo que a velocidad de transferencia de memoria se refiere es mayor al de la memoria global.

En la presente tesis se buscara maximizar el tamaño de la memoria cache L1, por lo cual se hará uso de la función “cudaFuncSetCacheConfig(*nombre_kernel*, cudaFuncCachePreferL1)”. Esta función indica al GPU que el espacio de memoria que es compartido entre la cache L1 y la memoria compartida sea totalmente asignado hacia la memoria cache L1 para de esta manera poder reducir la cantidad de veces que el GPU pide datos a la memoria global debido a la reducida cantidad de tamaño de la memoria cache L1.

3.3.2 Coalescencia de Memoria

El acceso a los diferentes tipos de memoria del GPU se realiza mediante transacciones de una cantidad determinada de bytes (leen una cantidad fija de bytes en cada transacción) [16]. Debido a esta característica del GPU, se desea que los elementos estén alineados para minimizar el número de transacciones para la transferencia de la información [16]. Mientras más datos no estén alineados al tamaño del bloque transferido (figura 3.9), aumenta la probabilidad de requerir un mayor número de transacciones de memoria, lo cual afectara de manera negativa el rendimiento del programa al reducir su tiempo de ejecución.

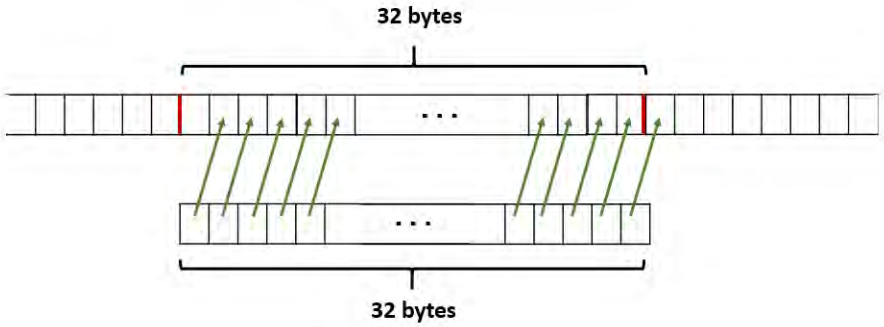


Figura 3.9 Desalineamiento para una transacción de 32 bytes.

Para mantener un rendimiento deseable en el ritmo en que la información es transferida en el GPU se recomienda que la primera dirección de memoria sea múltiplo del tamaño de bloque que se transfiere [16].

En la implementación de esta tesis, el tamaño de la imagen a procesar no es necesariamente múltiplo del tamaño del bloque a transferir, más aun, al extender la imagen (por el rellanado) es muy probable que el tamaño final no sea múltiplo del tamaño de bloque a transferir. En la figura 3.10 se muestra el código que extiende la imagen a un tamaño múltiplo del bloque a transferir, asegurando así que todas las transacciones están alineadas.

```
#define ver H //# de pixeles verticales en la imagen
#define hor W //# de pixeles horizontales en la imagen
#define LSize x// Size of memory line
#define k l //Define tamaño del lado del área de análisis

#define horp ((hor+k-1+LSize-1)/LSize)*LSize
#define verp ((ver+k-1+LSize-1)/LSize)*LSize
```

Figura 3.10 Código para extender el tamaño del rellenado para que sea múltiplo de un número.

A partir de las operaciones numéricas realizadas, *horp* y *verp* serán los utilizados para definir el tamaño de los bordes de la imagen a filtrar (ver figura 3.11), los cuales serán múltiplo de *LSize* (tamaño de la transacción de memoria designada por el dispositivo).

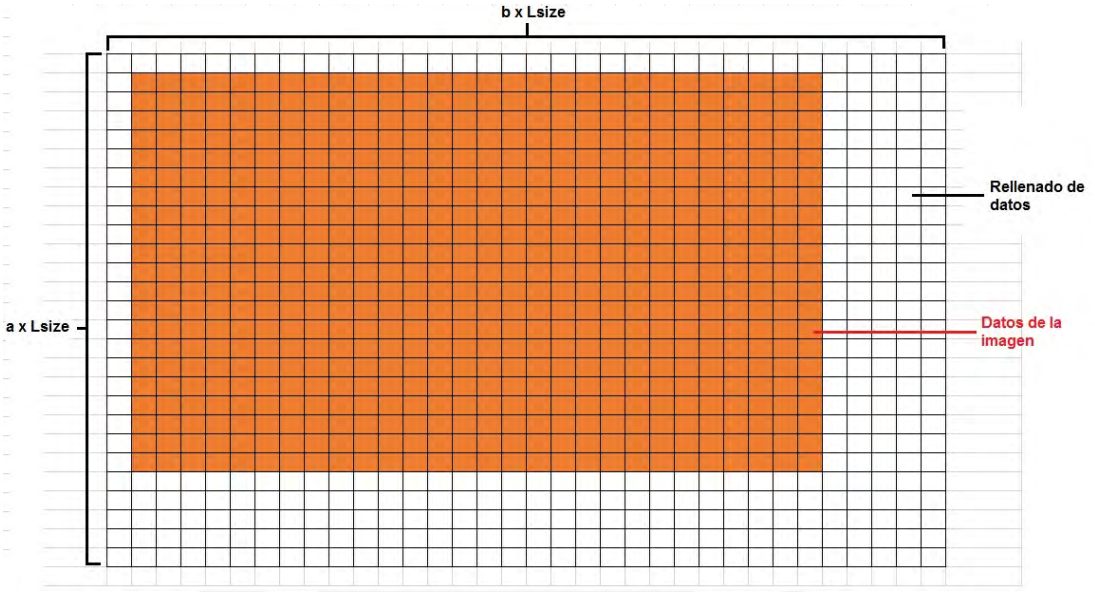


Figura 3.11 Nuevo tamaño de bordes para asegurar el alineamiento de datos, a y b son constantes.

3.3.3 Maximizar Ocupación de las Tramas de Multiprocesador

Al ejecutar un programa en el GPU se buscará que el dispositivo este ocupado al completo procesando las instrucciones. Para lograr este fin será necesario proponer en el programa un tamaño de hilos por bloques para la ejecución del kernel. Se utilizara el CUDA Occupancy Calculator (anexo 1) para encontrar el número adecuado de hilos por bloque a utilizar en nuestro programa [16].

3.3.4 Desenrollado de Bucles

Para un adecuado uso del GPU en el programa del filtro de mediana 2D se busca que el tiempo que este ocupado el dispositivo en la ejecución del algoritmo lo ocupe realizando operaciones aritméticas, ya que estas son las que procesaran la imagen. Las instrucciones de control de flujo se convierten en operaciones no deseadas, ya que consumen tiempo que se podría utilizar en instrucciones aritméticas.

Como se indicó en la sección 2.4, se pueden reducir el número de instrucciones de control de flujo a costa de aumentar la cantidad de código escrito. Otra forma en la que se ejecuta el desenrollado de bucles es mediante el compilador de CUDA de Nvidia. El compilador ejecuta las operaciones iterativas escritas en C mediante un desenrollado de bucle en lenguaje ensamblador de forma automática [16]. Sin

embargo, es necesario que la función iterativa sea predecible para que se ejecute el desenrollado de bucle satisfactoriamente (figura 3.12).

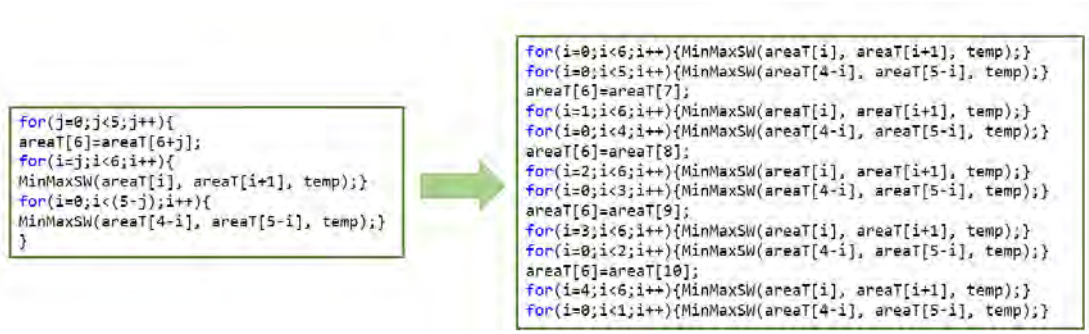


Figura 3.12 Desenrollado de bucle para una instrucción *For* de tamaño variable.

CAPITULO 4

RESULTADOS

4.1 Datos de Prueba

4.1.1 Descripción de la Implementación

En la presente implementación del filtro de mediana 2D se realizará el algoritmo para procesar áreas de análisis de *3x3*, *5x5* y *7x7*. La ejecución del filtro de mediana *3x3* se realizará con el algoritmo median_odd_N, mientras que en las implementaciones de *5x5* y *7x7* se utilizara el método de selección olvidadiza procesando 4 ventanas por hilo.

El programa a ejecutar realizara el filtrado de mediana sobre una imagen de un solo canal. El programa a su vez también registrara el tiempo que demora la ejecución del filtro y su respectiva conversión a megapíxeles por segundo como parámetro de velocidad.

Con el objetivo de obtener el valor de velocidad del filtro con mayor precisión y exactitud se busca eliminar factores que contribuyan a obtener resultados erróneos en la medición del tiempo, como por ejemplo el tiempo de sobrecarga. El tiempo de sobrecarga se eliminará al “precalentar” (activarlos previamente) los núcleos del GPU para que el tiempo extra que consume en la primera ejecución del programa en un periodo de tiempo prolongado de inactividad no afecte los resultados.

4.1.2 Equipo de Prueba

Las pruebas se realizaron con un procesador Intel Core i5-4690 de 3.50 GHz, 16 GB de memoria RAM y el GPU GeForce GTX 1060 de 3 GB de memoria interna a 4004 MHz con un ancho de bus de 192 bits, 1152 núcleos CUDA con una velocidad de reloj de 1.86 GHz.

Se implementó sobre el sistema operativo Windows 10 Pro de 64 bits. Los drivers utilizados para la GPU estuvieron en su versión 417.22, con CUDA en su versión 9.1.

4.1.3 Referencias de Comparación

Se analizaron los resultados para diferentes tamaños de áreas de análisis, teniendo como medida de desempeño el número de pixeles procesados por segundo. Se compararon los desempeños de la implementación con GPU (usando CUDA). Como referencia de comparación se utilizó el algoritmo de filtro de mediana 2D de Matlab (en su versión con GPU), el algoritmo PCMF y PRMF, teniendo en cuenta los mejores resultados obtenidos de cada uno y corrigiendo ambos valores por un factor el cual busca equiparar a los GPUs usados en dichas ejecuciones con el GPU propio de esta implementación.

4.1.4 Imágenes de Prueba

Las imágenes utilizadas para la prueba del algoritmo están en formato .TIF. Con el objetivo de demostrar la flexibilidad del filtro, y sus propiedades de estabilidad frente a tamaños de imágenes variables las pruebas se realizarán para 3 archivos diferentes, cada una a diferentes niveles de ruido sal y pimienta.

- Imagen 1 (figura 4.1):
  Nombre: Lena
  Tamaño: 512 x 512

Figura 4.1 Imagen Lena.

- Imagen 2 (figura 4.2):
  Nombre: auto
  Tamaño: 1920 x 1080

Figura 4.2 Imagen auto.

- Imagen 3 (figura 4.3):
  Nombre: bmw
  Tamaño: 2560 x 2560

Figura 4.3 Imagen bmw.

Todas las imágenes serán probadas a niveles de ruido sal y pimienta del 20% y 50%.

4.2 Resultados Computacionales

4.2.1 Resultados del Filtro

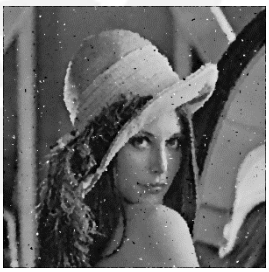
En las figuras 4.4, 4.5 y 4.6 se muestran el resultado de las imágenes filtradas dado un cierto nivel de ruido impulsivo para un tamaño diferente de área de análisis en cada imagen.

- Lena:

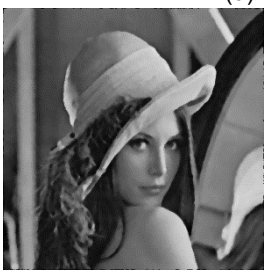
(a) Ruido impulsivo 20%.

(b) Ruido impulsivo 50%.

(c) Filtro de mediana 3x3.



(d) Filtro de mediana 5x5.

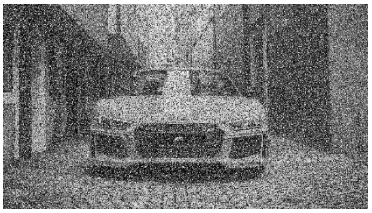


(e) Filtro de mediana 7x7.

Figura 4.4 Aplicación del filtro de mediana 3x3, 5x5 y 7x7 sobre la imagen Lena a un nivel de ruido impulsivo de 20% y 50%.

- auto:

(a) Ruido impulsivo 20%



(b) Ruido impulsivo 50%

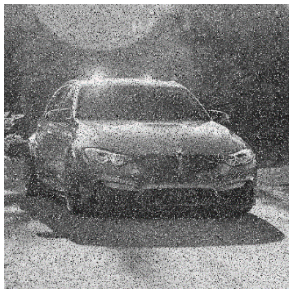
(c) Filtro de mediana 3x3.

(d) Filtro de mediana 5x5.

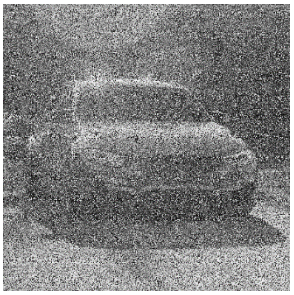
(e) Filtro de mediana 7x7.

Figura 4.5 Aplicación del filtro de mediana 3x3, 5x5 y 7x7 sobre la imagen auto a un nivel de ruido impulsivo de 20% y 50%.

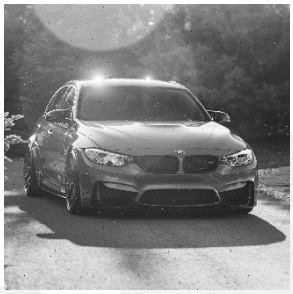
- bmw:



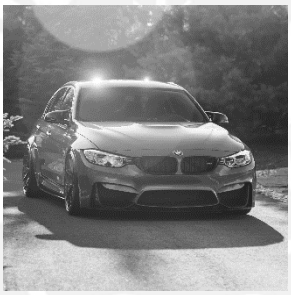
(a) Ruido impulsivo 20%.

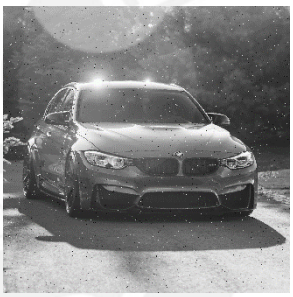


(b) Ruido impulsivo 50%.

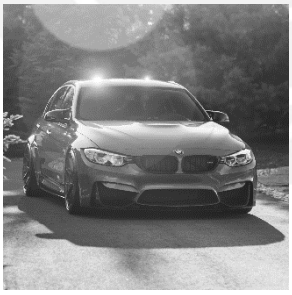


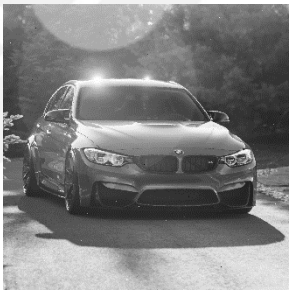
(c) Filtro de mediana 3x3





(d) Filtro de mediana 5x5.





(d) Filtro de mediana 7x7.

Figura 4.6 Aplicación del filtro de mediana 3x3, 5x5 y 7x7 sobre la imagen bmw a un nivel de ruido impulsivo de 20% y 50%.

En la tabla 2 se muestra los resultados de la ejecución del algoritmo del filtro en tiempo de ejecución y velocidad de megapíxeles por segundo.

Área de análisis	Lena (512x512)		auto (1920x1080)		bmw (2560x2560)	
	Tiempo (ms)	Velocidad (Mpix/s)	Tiempo (ms)	Velocidad (Mpix/s)	Tiempo (ms)	Velocidad (Mpix/s)
3x3	0.017	15374.64	0.1339	15487.96	0.4179	15679.74
5x5	0.0393	6671	0.2939	7054.18	0.9158	7156.08
7x7	0.1028	2550.44	0.8031	2581.88	2.4978	2623.75

Tabla 2

4.3 Análisis de Resultados

Para realizar la comparación de los resultados obtenidos con las referencias especificadas en la sección 4.1.3 se tomará el promedio de las velocidades obtenidas de nuestro filtro en cada tamaño de área de análisis. Adicionalmente se multiplicará por un factor determinado por el cociente de la frecuencia de reloj multiplicado por la cantidad de núcleos utilizados por el GPU de esta implementación (1152 núcleos a 1.86 GHz) entre la multiplicación de la cantidad de núcleos y la frecuencia de los GPUs de los filtros de referencia. En el caso del filtro PCMF se obtuvo el mejor resultado con una GPU en una GTX 480 [9] (480 núcleos a 1401 MHz). El filtro PRMF obtuvo mejor resultado con una GPU GTX 750[20] la cual tiene 512 núcleos a 1.08 GHz. Los factores para multiplicar los resultados de los filtros PCMF y PRMF serán 3.19 y 3.86 respectivamente.

La figura 4.7 muestra la comparación de nuestra implementación, la cual lleva de nombre 4PTMF (4 Pixeles per Thread Median Filter o Filtro de Mediana de 4 pixeles por hilo), contra las implementaciones PCMF, PRMF y el filtro de mediana de Matlab con GPU.

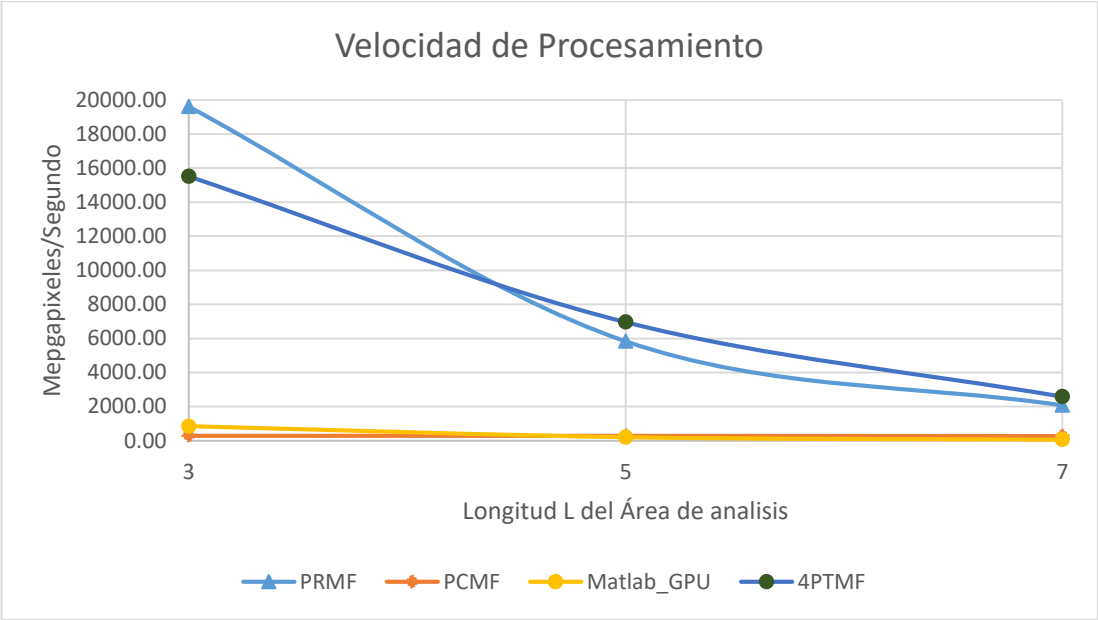


Figura 4.7 Velocidad de procesamiento del 4PTMF contra los filtros PRMF, PCMF y Matlab con GPU.

De la figura 4.7 se puede observar que la velocidad alcanzada por el presente filtro supera en velocidad a las implementaciones previas en las áreas de análisis de 5x5 y 7x7 (por 19% y 24% respectivamente). En el área de análisis de 3x3, al no ser posible el poder procesar 4 pixeles por hilo, la implementación PRMF sigue siendo con la que se alcanza mejores resultados.

Conclusiones

- Se creó un algoritmo de filtro de mediana 2D que funciona para procesar datos en punto flotante y datos en punto fijo.
- El 4PTMF mantiene una velocidad en Mpix/seg casi estable para cualquier tamaño de imagen dado un área de análisis fija, sin importar el nivel de ruido que esta contenga o la forma original en que estén ordenados los datos de entrada de la ventana de análisis.
- Para la implementación de filtros de mediana es más efectivo el uso de algoritmos que tengan como principal objetivo el hallar la mediana en vez de algoritmos de ordenamiento.
- El uso exclusivo de redes de ordenamiento completo o de forma parcial como sub partes de algoritmos que tengan un propósito diferente ocasiona que estos últimos ganen la propiedad de predictibilidad de las redes de ordenamiento, lo cual garantiza que no habrá divergencia en la ejecución paralela.
- La implementación del 4PTMF supero en velocidad a los filtros PCMF, PRMF y al filtro de Matlab usando GPU, teniendo mayor ventaja porcentual conforme el área de análisis crece.

Recomendaciones

- Se recomienda probar la presente implementación del filtro 4PTMF en hardware con mejores prestaciones (mayor cantidad de núcleos o mayor frecuencia) y GPUs con las más recientes arquitecturas para conocer el si el rendimiento mejora con el cambio de hardware (se recomienda realizar una implementación con GPUs que tengan tensor cores).
- Se recomienda optimizar la forma de almacenar los valores de la ventana de análisis a procesar mientras se ejecuta el filtro de mediana para que no exceda el número máximo de registros por hilo o bloque, y en consecuencia disminuya el rendimiento de la ejecución del programa.
- Se recomienda optimizar la búsqueda de valores extremos para la ejecución de la selección olvidadiza. Estos algoritmos para encontrar extremos tendrían que estar basados posiblemente en redes de ordenamiento o partes de ellas para que el filtro conserve su propiedad de no divergencia en la ejecución.

.

BIBLIOGRAFÍA

[1] S. Perreault and P. Hebert, “Median Filtering in Constant Time,” *IEEE Trans. Image Process.*, vol. 16, no. 9, pp. 2389–2394, 2007.

[2] NVIDIA Corporation, “Procesamiento Paralelo CUDA.” [Online]. Available: http://www.nvidia.es/object/cuda-parallel-computing-es.html. [Accessed: 30-Apr-2017].

[3] NVIDIA Corporation, “GPU frente a CPU ¿Qué es el GPU Computing?” [Online]. Available: http://www.nvidia.es/object/gpu-computing-es.html. [Accessed: 30-Apr-2017].

[4] NVIDIA Corporation, “CUDA Toolkit.” [Online]. Available: https://developer.nvidia.com/cuda-toolkit. [Accessed: 30-Apr-2017].

[5] C. Carranza, V. Murray, M. Pattichis, and E. S. Barriga, “Multiscale AM-FM decompositions with GPU acceleration for diabetic retinopathy screening,” *Proc. IEEE Southwest Symp. Image Anal. Interpret.*, pp. 121–124, 2012.

[6] M. Kachelrieß, “Branchless vectorized median filtering,” *IEEE Nucl. Sci. Symp. Conf. Rec.*, no. Dll, pp. 4099–4105, 2009.

[7] W. Chen, M. Beister, Y. Kyriakou, and M. Kachelrieß, “High Performance Median Filtering using Commodity Graphics Hardware,” pp. 4142–4147, 2009.

[8] P. A. R. Ricardo M. Sánchez, “BIDIMENSIONAL MEDIAN FILTER FOR PARALLEL COMPUTING ARCHITECTURES,” *ICASSP 2012*, pp. 1549–1552, 2012.

[9] R. M. Sánchez and P. A. Rodríguez, “Highly parallelable bidimensional median filter for modern parallel programming models,” *J. Signal Process. Syst.*, vol. 71, no. 3, pp. 221–235, 2013.

[10] G. Perrot, D. Stephane, and R. Couturier, “Fine-tuned high-speed implementation of a GPU-based median fi lter.,” 2014.

[11] E. Herruzo, G. Ruíz, and J. I. Benavides, “A new parallelsorting algorithm based on Odd-Even mergesort,” in *EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing*, 2007, pp. 2–6.

[12] H. Peters, O. Schulz-hildebrandt, N. Luttenberger, and A. B. Sort, “A novel sorting algorithm for many-core architectures based on adaptive bitonic sort,” *IEEE 26th Int. Parallel Distrib. Process. Symp.*, vol. 1, 2012.

[13] B. Mozaffari, “Optimization of Odd-Even Transposition Network,” in *2nd International Conference on Education Technology and Computer (ICETC)*, 2010, pp. 364–366.

[14] Goodyear Aerospace, “Sorting networks and their applications,” in *AFIPS ’68*, 1968, pp. 307–314.

[15] P. Kolte, R. Smith, and W. Su, “A Fast Median Filter using AltiVec,” in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1999.

[16] NVIDIA Corporation, *CUDA C Programing Guide*. 2017.

[17] R. Sánchez, “DISEÑO E IMPLEMETACIÓN DEL FILTRO MEDIANO DE DOS DIMENSIONES PARA ARQUITECTURAS SIMD,” Pontificia Universidad Católica del Perú, 2011.

[18] MathWorks, “2-D median filtering - MATLAB medfilt2.” .

[19] J. JáJá, *Introduction to Parallel Algorithms*. Addison Wesley Publishing Company Inc., 1992.

[20] O. Green, “Efficient scalable median filtering using histogram-based operations,” *IEEE Trans. Image Process.*, vol. 27, no. 5, pp. 2217–2228, 2018.

Anexos

Anexo 1: CUDA Occupancy Calculator

Anexo 2: 4PTMF.cu

Anexo 1: CUDA Occupancy Calculator

CUDA Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	6.2	(Help)
1.b) Select Shared Memory Size Config (bytes)	65536	
1.c) Select Global Load Caching Mode	L1+L2 (ca)	

2.) Enter your resource usage:		
Threads Per Block	128	(Help)
Registers Per Thread	32	
Shared Memory Per Block (bytes)	4096	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	2048	(Help)
Active Warps per Multiprocessor	64	
Active Thread Blocks per Multiprocessor	16	
Occupancy of each Multiprocessor	50%	

Physical Limits for GPU Compute Capability:	6.2
Threads per Warp	32
Max Warps per Multiprocessor	128
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	4096
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	49152
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	4

Allocated Resources		Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps	(Threads Per Block / Threads Per Warp)	4	128	32
Registers	(Warp limit per SM due to per-warp reg count)	4	64	16
Shared Memory (Bytes)		4096	49152	16

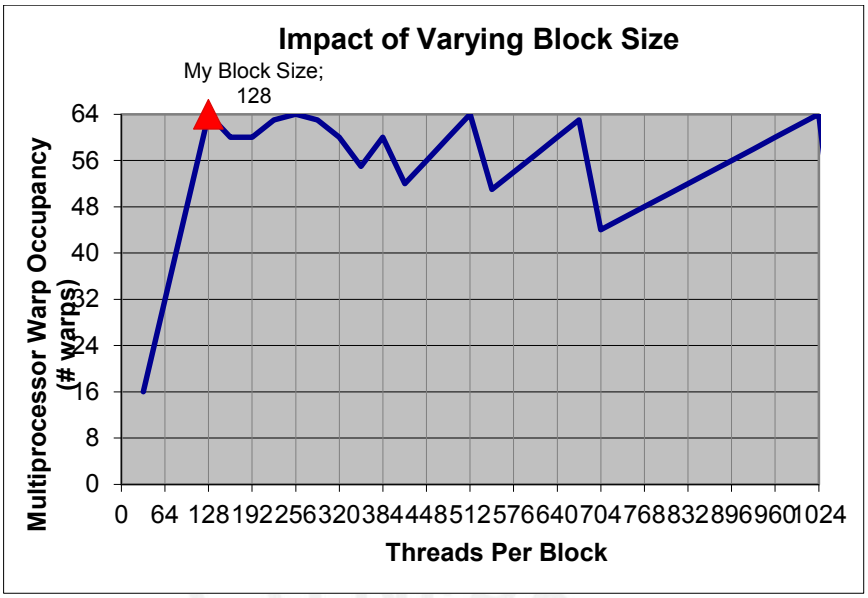
Note: SM is an abbreviation for (Streaming) Multiprocessor

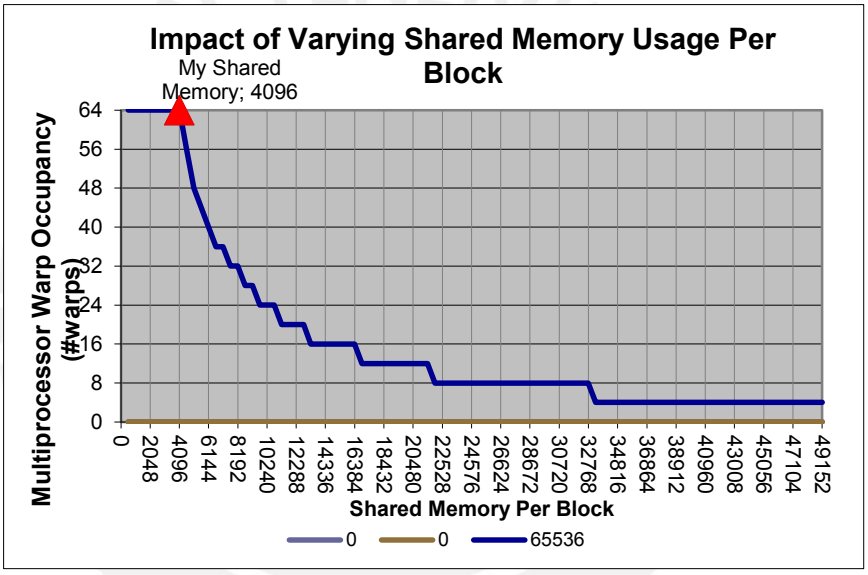
Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	32		
Limited by Registers per Multiprocessor	**16**	**4**	**64**
Limited by Shared Memory per Multiprocessor	**16**	**4**	**64**

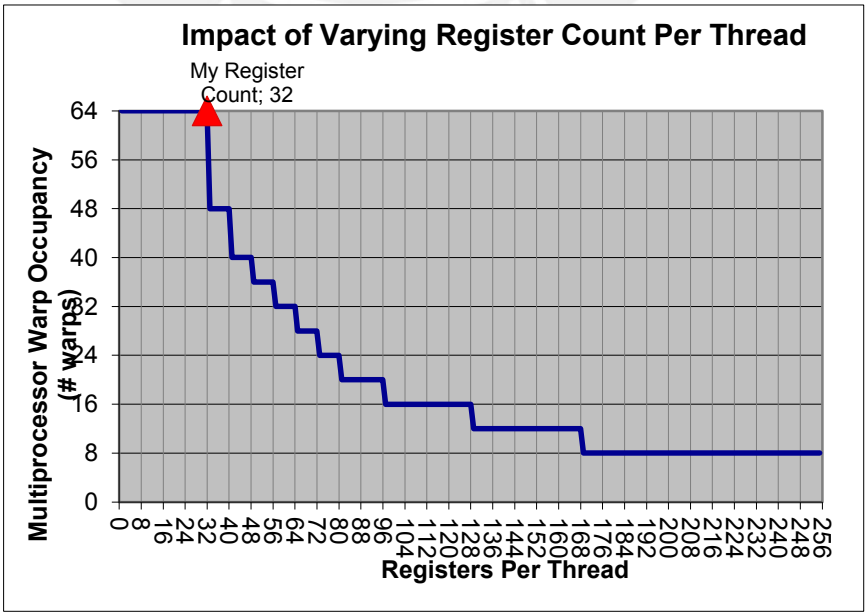
Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 128
Occupancy = 64 / 128 = 50%

CUDA Occupancy Calculator	
Version:	7.5
Copyright and License	


Impact of Varying Block Size
My Block Size; 128
Multiprocessor Warp Occupancy (# warps)
0
8
16
24
32
40
48
56
64
0 64 128 192 256 320 384 448 512 576 640 704 768 832 896 960 1024
Threads Per Block


Impact of Varying Shared Memory Usage Per Block
My Shared Memory; 4096
Multiprocessor Warp Occupancy (#warps)
0
8
16
24
32
40
48
56
64
0 2048 4096 6144 8192 10240 12288 14336 16384 18432 20480 22528 24576 26624 28672 30720 32768 34816 36864 38912 40960 43008 45056 47104 49152
Shared Memory Per Block
0
0
65536


Impact of Varying Register Count Per Thread
My Register Count; 32
Multiprocessor Warp Occupancy (# warps)
0
8
16
24
32
40
48
56
64
0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 128 136 144 152 160 168 176 184 192 200 208 216 224 232 240 248 256
Registers Per Thread

Anexo 2: 4PTMF.cu

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <math.h>

#define k 7 //Define las dimensiones del arreglo
#define ver 3840    //# de pixeles verticales en la imagen
#define hor 3840    // # de pixeles horizontales en la imagen
#define MC 1 //Memory coalescence padding: 0=off 1=on
#define LSize 128// Size of memory line
#define pixelsPerThread 4 //Number of pixels that are procesisng for each
thread, change if is 3x3 to 1, if it is 5x5 or higher put 4
#define pixelsPerThreadMode 4 //Number of pixels that are procesisng for each
thread

// For CUDA
#define threadsPerBlock 128

//Windows sizes
#if pixelsPerThreadMode == 0
#define winSize  k*k;      //number of pixels to take 4 windows
#define minAreaN  k*k;     //minimun quantity of tunbers to proces the windows
#endif
#if pixelsPerThreadMode == 1
#define winSize  (k+1)*(k+1);     //number of pixels to take 4 windows
#endif

//Memory coalescence padding
#if MC == 0
#define horp hor+k-1 // Computed as ((w+k-1+31)/32)*32
#define verp ver+k-1 // Computed as ((h+k-1+31)/32)*32
#endif
#if MC == 1
#define horp ((hor+k-1+LSize-1)/LSize)*LSize // Computed as ((w+k-1+31)/32)*32
#define verp ((ver+k-1+LSize-1)/LSize)*LSize // Computed as ((h+k-1+31)/32)*32
#endif

//Swap Max Min
#define MinMaxSW( x, y, t ) ( t=fmin(x,y),y=fmax(x,y),x=t );

// CUDA kernel L3
__global__ void SorNet_GPU_L3(float *filmedCuda, float *ImBaseCuda)
{
	float areaA[k*k], areaT[k*k];
	unsigned int a, b, x, y, i, p, pixelNumber;

	pixelNumber = (blockIdx.x*blockDim.x) + threadIdx.x; // Pixel position
in the 1D array
	// Remmaping to a (x,y) format. Not optimal
	x = pixelNumber / hor; // Integer division
	y = pixelNumber % hor; // Remainder

	//Load analysis area (window)
	for (a = 0; a<k; a++)
	{
		for (b = 0; b<k; b++)
```

```
		{
			areaA[a*k + b] = ImBaseCuda[(x + a)*(horp)+y + b];
		}
	}

	//kolte basic

	///sort colum
	for (p = 0; p<(k*k - 1); p = p + 3)
	{
		areaT[p] = fmin(areaA[p], areaA[p + 1]);
		areaT[p + 1] = fmax(areaA[p], areaA[p + 1]);
		areaA[p + 1] = fmin(areaT[p + 1], areaA[p + 2]);
		areaT[p + 2] = fmax(areaT[p + 1], areaA[p + 2]);
		areaT[p + 1] = fmax(areaT[p], areaA[p + 1]);
		areaT[p] = fmin(areaT[p], areaA[p + 1]);
	}

	////sort fil
	for (p = 0; p<k; p++)
	{
		areaA[p] = fmin(areaT[p], areaT[p + 3]);
		areaA[p + 3] = fmax(areaT[p], areaT[p + 3]);
		areaT[p + 3] = fmin(areaA[p + 3], areaT[p + 6]);
		areaA[p + 6] = fmax(areaA[p + 3], areaT[p + 6]);
		areaA[p + 3] = fmax(areaA[p], areaT[p + 3]);
		areaA[p] = fmin(areaA[p], areaT[p + 3]);
	}

	//Diagonal
	areaT[k - 1] = fmin(areaA[k - 1], areaA[k + 1]);
	areaT[k + 1] = fmax(areaA[k - 1], areaA[k + 1]);
	areaA[k + 1] = fmin(areaT[k + 1], areaA[(k - 1)*k]);
	areaT[(k - 1)*k] = fmax(areaT[k + 1], areaA[(k - 1)*k]);
	areaT[k + 1] = fmax(areaT[k - 1], areaA[k + 1]);
	areaT[k - 1] = fmin(areaT[k - 1], areaA[k + 1]);

	//Colocar valor de Mediana
	filmedCuda[x*hor + y] = areaT[4];
}

// CUDA kernel L5
__global__ void SorNet_GPU_L5(float *filmedCuda, float *ImBaseCuda)
{
	float areaA[36], areaT[16], areaD[16], areaTD[16], areaDD[16];
	float temp, temp2;
	unsigned int a, b, x, y, c, f, i, pixelNumber;
	unsigned int minAreaN = 14;

	pixelNumber = (blockIdx.x*blockDim.x) + threadIdx.x; // Pixel position
in the 1D array
	// Remmaping to a (x,y) format. Not optimal
	x = 2 * (pixelNumber / (hor / 2)); // Integer division
	y = 2 * (pixelNumber % (hor / 2)); // Remainder
	//Load analysis area (window)
	for (a = 0; a<(k + 1); a++)
	{
		for (b = 0; b<(k + 1); b++)
		{
			areaA[a*(k + 1) + b] = ImBaseCuda[(x + a)*(horp)+y + b];
		}
	}
```

```
//Take common area
for (a = 0; a<(k - 1); a++)
{
	for (b = 0; b<(k - 1); b++)
	{
		areaT[a*(k - 1) + b] = areaA[(a + 1)*(k + 1) + b + 1];
	}
}
//Find max and min from min numbers needed
//14
for (i = 0; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
}
for (i = 0; i < (minAreaN - 4); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
}
MinMaxSW(areaT[0], areaT[4], temp);	//min
MinMaxSW(areaT[3], areaT[7], temp);	//max
MinMaxSW(areaT[8], areaT[12], temp);//min
MinMaxSW(areaT[11], areaT[13], temp);	//max
MinMaxSW(areaT[0], areaT[8], temp);	//min
MinMaxSW(areaT[7], areaT[13], temp);	//max

//extra pixel 1 of common area
areaT[minAreaN - 1] = areaT[minAreaN];
//13
for (i = 1; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
}
for (i = 1; i < (minAreaN - 4); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
}
MinMaxSW(areaT[1], areaT[5], temp);	//min
MinMaxSW(areaT[4], areaT[8], temp);	//max
MinMaxSW(areaT[9], areaT[13], temp);//min
MinMaxSW(areaT[12], areaT[13], temp);	//max
MinMaxSW(areaT[1], areaT[9], temp);	//min
MinMaxSW(areaT[8], areaT[13], temp);	//max

//extra pixel 2 of common area
areaT[minAreaN - 1] = areaT[minAreaN + 1];
//12
for (i = 2; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
}
for (i = 2; i < (minAreaN - 3); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
}
MinMaxSW(areaT[2], areaT[6], temp);	//min
MinMaxSW(areaT[5], areaT[9], temp);	//max
MinMaxSW(areaT[2], areaT[10], temp);//min
MinMaxSW(areaT[9], areaT[13], temp);	//max
//
```

```
//Multy pixel per thread
for (a = 0; a<(k - 1); a++)
{
	for (b = 0; b<(k - 1); b++)
	{
		areaD[a*(k - 1) + b] = areaT[a*(k - 1) + b];
	}
}
//upper and low
//pixel1
areaT[minAreaN - 1] = areaA[1];
areaD[minAreaN - 1] = areaA[5 * (k + 1) + 1];
//11
for (i = 3; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
	MinMaxSW(areaD[i], areaD[i + 1], temp);
}
for (i = 3; i < (minAreaN - 4); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
	MinMaxSW(areaD[i], areaD[i + 2], temp);
	MinMaxSW(areaD[i + 1], areaD[i + 3], temp);
}
MinMaxSW(areaT[11], areaT[13], temp);	//min
MinMaxSW(areaT[12], areaT[13], temp);	//max
MinMaxSW(areaT[3], areaT[7], temp);	//min
MinMaxSW(areaT[6], areaT[10], temp);	//max
MinMaxSW(areaT[3], areaT[11], temp);//min
MinMaxSW(areaT[10], areaT[13], temp);	//max
MinMaxSW(areaD[11], areaD[13], temp);	//min
MinMaxSW(areaD[12], areaD[13], temp);	//max
MinMaxSW(areaD[3], areaD[7], temp);	//min
MinMaxSW(areaD[6], areaD[10], temp);	//max
MinMaxSW(areaD[3], areaD[11], temp);//min
MinMaxSW(areaD[10], areaD[13], temp);	//max

//pixel2
areaT[minAreaN - 1] = areaA[2];
areaD[minAreaN - 1] = areaA[5 * (k + 1) + 2];
//10
for (i = 4; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
	MinMaxSW(areaD[i], areaD[i + 1], temp);
}
for (i = 4; i < (minAreaN - 4); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
	MinMaxSW(areaD[i], areaD[i + 2], temp);
	MinMaxSW(areaD[i + 1], areaD[i + 3], temp);
}
MinMaxSW(areaT[4], areaT[8], temp);	//min
MinMaxSW(areaT[7], areaT[11], temp);	//max
MinMaxSW(areaT[4], areaT[12], temp);	//min
MinMaxSW(areaT[11], areaT[13], temp);	//max
MinMaxSW(areaD[4], areaD[8], temp);	//min
MinMaxSW(areaD[7], areaD[11], temp);	//max
MinMaxSW(areaD[4], areaD[12], temp);	//min
```

```
MinMaxSW(areaD[11], areaD[13], temp);    //max

//pixel3
areaT[minAreaN - 1] = areaA[3];
areaD[minAreaN - 1] = areaA[5 * (k + 1) + 3];
//9
for (i = 5; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
	MinMaxSW(areaD[i], areaD[i + 1], temp);
}
for (i = 5; i < (minAreaN - 4); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
	MinMaxSW(areaD[i], areaD[i + 2], temp);
	MinMaxSW(areaD[i + 1], areaD[i + 3], temp);
}
MinMaxSW(areaT[5], areaT[9], temp);      //min
MinMaxSW(areaT[8], areaT[12], temp);     //max
MinMaxSW(areaT[5], areaT[13], temp);     //min
MinMaxSW(areaT[12], areaT[13], temp);    //max
MinMaxSW(areaD[5], areaD[9], temp);      //min
MinMaxSW(areaD[8], areaD[12], temp);     //max
MinMaxSW(areaD[5], areaD[13], temp);     //min
MinMaxSW(areaD[12], areaD[13], temp);    //max

//pixel4
areaT[minAreaN - 1] = areaA[4];
areaD[minAreaN - 1] = areaA[5 * (k + 1) + 4];
//8
for (i = 6; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
	MinMaxSW(areaD[i], areaD[i + 1], temp);
}
for (i = 6; i < (minAreaN - 3); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
	MinMaxSW(areaD[i], areaD[i + 2], temp);
	MinMaxSW(areaD[i + 1], areaD[i + 3], temp);
}
MinMaxSW(areaT[6], areaT[10], temp);     //min
MinMaxSW(areaT[9], areaT[13], temp);     //max
MinMaxSW(areaD[6], areaD[10], temp);     //min
MinMaxSW(areaD[9], areaD[13], temp);     //max

//left and right
for (a = 0; a<k - 1; a++)
{
	for (b = 0; b<k - 1; b++)
	{
		areaTD[a*(k - 1) + b] = areaT[a*(k - 1) + b];
		areaDD[a*(k - 1) + b] = areaD[a*(k - 1) + b];
	}
}
//p1
areaT[minAreaN - 1] = areaA[0];
areaTD[minAreaN - 1] = areaA[(k + 1) - 1];
areaD[minAreaN - 1] = areaA[k + 1];
areaDD[minAreaN - 1] = areaA[2 * (k + 1) - 1];
```

```
//7
for (i = 7; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
	MinMaxSW(areaD[i], areaD[i + 1], temp);
	MinMaxSW(areaTD[i], areaTD[i + 1], temp);
	MinMaxSW(areaDD[i], areaDD[i + 1], temp);
}

MinMaxSW(areaT[7], areaT[9], temp);	//min
MinMaxSW(areaT[8], areaT[10], temp);	//max
MinMaxSW(areaT[11], areaT[13], temp);	//min
MinMaxSW(areaT[12], areaT[13], temp);	//max
MinMaxSW(areaT[7], areaT[11], temp);	//min
MinMaxSW(areaT[10], areaT[13], temp);	//max
MinMaxSW(areaD[7], areaD[9], temp);	//min
MinMaxSW(areaD[8], areaD[10], temp);	//max
MinMaxSW(areaD[11], areaD[13], temp);	//min
MinMaxSW(areaD[12], areaD[13], temp);	//max
MinMaxSW(areaD[7], areaD[11], temp);	//min
MinMaxSW(areaD[10], areaD[13], temp);	//max
MinMaxSW(areaTD[7], areaTD[9], temp);	//min
MinMaxSW(areaTD[8], areaTD[10], temp);	//max
MinMaxSW(areaTD[11], areaTD[13], temp);	//min
MinMaxSW(areaTD[12], areaTD[13], temp);	//max
MinMaxSW(areaTD[7], areaTD[11], temp);	//min
MinMaxSW(areaTD[10], areaTD[13], temp);	//max
MinMaxSW(areaDD[7], areaDD[9], temp);	//min
MinMaxSW(areaDD[8], areaDD[10], temp);	//max
MinMaxSW(areaDD[11], areaDD[13], temp);	//min
MinMaxSW(areaDD[12], areaDD[13], temp);	//max
MinMaxSW(areaDD[7], areaDD[11], temp);	//min
MinMaxSW(areaDD[10], areaDD[13], temp);	//max

//p2
areaT[minAreaN - 1] = areaA[(k + 1)];
areaTD[minAreaN - 1] = areaA[2 * (k + 1) - 1];
areaD[minAreaN - 1] = areaA[2 * (k + 1)];
areaDD[minAreaN - 1] = areaA[3 * (k + 1) - 1];
//6
for (i = 8; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
	MinMaxSW(areaD[i], areaD[i + 1], temp);
	MinMaxSW(areaTD[i], areaTD[i + 1], temp);
	MinMaxSW(areaDD[i], areaDD[i + 1], temp);
}
MinMaxSW(areaT[8], areaT[10], temp);	//min
MinMaxSW(areaT[9], areaT[11], temp);	//max
MinMaxSW(areaT[8], areaT[12], temp);	//min
MinMaxSW(areaT[11], areaT[13], temp);	//max
MinMaxSW(areaD[8], areaD[10], temp);	//min
MinMaxSW(areaD[9], areaD[11], temp);	//max
MinMaxSW(areaD[8], areaD[12], temp);	//min
MinMaxSW(areaD[11], areaD[13], temp);	//max
MinMaxSW(areaTD[8], areaTD[10], temp);	//min
MinMaxSW(areaTD[9], areaTD[11], temp);	//max
MinMaxSW(areaTD[8], areaTD[12], temp);	//min
MinMaxSW(areaTD[11], areaTD[13], temp);	//max
MinMaxSW(areaDD[8], areaDD[10], temp);	//min
MinMaxSW(areaDD[9], areaDD[11], temp);	//max
MinMaxSW(areaDD[8], areaDD[12], temp);	//min
```

```
MinMaxSW(areaDD[11], areaDD[13], temp);  //max

//p3
areaT[minAreaN - 1] = areaA[2 * (k + 1)];
areaTD[minAreaN - 1] = areaA[3 * (k + 1) - 1];
areaD[minAreaN - 1] = areaA[3 * (k + 1)];
areaDD[minAreaN - 1] = areaA[4 * (k + 1) - 1];
//5
for (i = 9; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
	MinMaxSW(areaD[i], areaD[i + 1], temp);
	MinMaxSW(areaTD[i], areaTD[i + 1], temp);
	MinMaxSW(areaDD[i], areaDD[i + 1], temp);
}
MinMaxSW(areaT[9], areaT[11], temp);     //min
MinMaxSW(areaT[10], areaT[12], temp);    //max
MinMaxSW(areaT[9], areaT[13], temp);     //min
MinMaxSW(areaT[12], areaT[13], temp);    //max
MinMaxSW(areaD[9], areaD[11], temp);     //min
MinMaxSW(areaD[10], areaD[12], temp);    //max
MinMaxSW(areaD[9], areaD[13], temp);     //min
MinMaxSW(areaD[12], areaD[13], temp);    //max
MinMaxSW(areaTD[9], areaTD[11], temp);   //min
MinMaxSW(areaTD[10], areaTD[12], temp);  //max
MinMaxSW(areaTD[9], areaTD[13], temp);   //min
MinMaxSW(areaTD[12], areaTD[13], temp);  //max
MinMaxSW(areaDD[9], areaDD[11], temp);   //min
MinMaxSW(areaDD[10], areaDD[12], temp);  //max
MinMaxSW(areaDD[9], areaDD[13], temp);   //min
MinMaxSW(areaDD[12], areaDD[13], temp);  //max

//p4
areaT[minAreaN - 1] = areaA[3 * (k + 1)];
areaTD[minAreaN - 1] = areaA[4 * (k + 1) - 1];
areaD[minAreaN - 1] = areaA[4 * (k + 1)];
areaDD[minAreaN - 1] = areaA[5 * (k + 1) - 1];
//4
MinMaxSW(areaT[10], areaT[11], temp);
MinMaxSW(areaT[12], areaT[13], temp);
MinMaxSW(areaT[10], areaT[12], temp);    //min
MinMaxSW(areaT[11], areaT[13], temp);    //max
MinMaxSW(areaD[10], areaD[11], temp);
MinMaxSW(areaD[12], areaD[13], temp);
MinMaxSW(areaD[10], areaD[12], temp);    //min
MinMaxSW(areaD[11], areaD[13], temp);    //max
MinMaxSW(areaTD[10], areaTD[11], temp);
MinMaxSW(areaTD[12], areaTD[13], temp);
MinMaxSW(areaTD[10], areaTD[12], temp);  //min
MinMaxSW(areaTD[11], areaTD[13], temp);  //max
MinMaxSW(areaDD[10], areaDD[11], temp);
MinMaxSW(areaDD[12], areaDD[13], temp);
MinMaxSW(areaDD[10], areaDD[12], temp);  //min
MinMaxSW(areaDD[11], areaDD[13], temp);  //max

//p5
areaT[minAreaN - 1] = areaA[4 * (k + 1)];
areaTD[minAreaN - 1] = areaA[5 * (k + 1) - 1];
areaD[minAreaN - 1] = areaA[5 * (k + 1)];
areaDD[minAreaN - 1] = areaA[6 * (k + 1) - 1];
//3
MinMaxSW(areaT[11], areaT[13], temp);    //min
```

```
	MinMaxSW(areaT[12], areaT[13], temp);	//max
	MinMaxSW(areaT[11], areaT[12], temp);	//min
	MinMaxSW(areaD[11], areaD[13], temp);	//min
	MinMaxSW(areaD[12], areaD[13], temp);	//max
	MinMaxSW(areaD[11], areaD[12], temp);	//min
	MinMaxSW(areaTD[11], areaTD[13], temp);	//min
	MinMaxSW(areaTD[12], areaTD[13], temp);	//max
	MinMaxSW(areaTD[11], areaTD[12], temp);	//min
	MinMaxSW(areaDD[11], areaDD[13], temp);	//min
	MinMaxSW(areaDD[12], areaDD[13], temp);	//max
	MinMaxSW(areaDD[11], areaDD[12], temp);	//min

	/////////////////
	//Colocar valor de Mediana
	filmedCuda[x*hor + y] = areaT[minAreaN - 2];
	filmedCuda[x*hor + y + 1] = areaTD[minAreaN - 2];
	filmedCuda[(x + 1)*hor + y] = areaD[minAreaN - 2];
	filmedCuda[(x + 1)*hor + y + 1] = areaDD[minAreaN - 2];
}

// CUDA kernel L7
__global__ void SorNet_GPU_L7(float *filmedCuda, float *ImBaseCuda)
{
	float areaA[64], areaT[36], areaD[36], areaTD[36], areaDD[36];
	float temp;
	unsigned int a, b, x, y, i, pixelNumber;
	unsigned int minAreaN = 26;

	pixelNumber = (blockIdx.x*blockDim.x) + threadIdx.x; // Pixel position
in the 1D array

	// Remmaping to a (x,y) format. Not optimal
	x = 2 * (pixelNumber / (hor / 2)); // Integer division
	y = 2 * (pixelNumber % (hor / 2)); // Remainder

	//Load analysis area (window)
	for (a = 0; a<(k + 1); a++)
	{
		for (b = 0; b<(k + 1); b++)
		{
			areaA[a*(k + 1) + b] = ImBaseCuda[(x + a)*(horp)+y + b];
		}
	}
	//Take common area
	for (a = 0; a<(k - 1); a++)
	{
		for (b = 0; b<(k - 1); b++)
		{
			areaT[a*(k - 1) + b] = areaA[(a + 1)*(k + 1) + b + 1];
		}
	}
	//Find max and min from min numbers needed
	//26
	for (i = 0; i < (minAreaN - 1); i = i + 2)//eliminate min max for min
numbers needed
	{
		MinMaxSW(areaT[i], areaT[i + 1], temp);
	}
	for (i = 0; i < (minAreaN - 3); i = i + 4)
	{
		MinMaxSW(areaT[i], areaT[i + 2], temp);
		MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
```

```
}
for (i = 0; i < (minAreaN - 3); i = i + 8)
{
	MinMaxSW(areaT[i], areaT[i + 4], temp);
	MinMaxSW(areaT[i + 3], areaT[i + 7], temp);
}
MinMaxSW(areaT[0], areaT[8], temp);			//min
MinMaxSW(areaT[7], areaT[15], temp);	//max
MinMaxSW(areaT[16], areaT[24], temp);	//min
MinMaxSW(areaT[23], areaT[25], temp);	//max
MinMaxSW(areaT[0], areaT[16], temp);	//min
MinMaxSW(areaT[15], areaT[25], temp);	//max

//extra pixel 1 of common area
areaT[minAreaN - 1] = areaT[minAreaN];
//25
for (i = 1; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
}
for (i = 1; i < (minAreaN - 3); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
}
for (i = 1; i < (minAreaN - 3); i = i + 8)
{
	MinMaxSW(areaT[i], areaT[i + 4], temp);
	MinMaxSW(areaT[i + 3], areaT[i + 7], temp);
}
MinMaxSW(areaT[1], areaT[9], temp);			//min
MinMaxSW(areaT[8], areaT[16], temp);	//max
MinMaxSW(areaT[17], areaT[25], temp);	//min
MinMaxSW(areaT[24], areaT[25], temp);	//max
MinMaxSW(areaT[1], areaT[17], temp);	//min
MinMaxSW(areaT[16], areaT[25], temp);	//max

//extra pixel 2 of common area
areaT[minAreaN - 1] = areaT[minAreaN + 1];
//24
for (i = 2; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
}
for (i = 2; i < (minAreaN - 3); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
}
for (i = 2; i < (minAreaN - 3); i = i + 8)
{
	MinMaxSW(areaT[i], areaT[i + 4], temp);
	MinMaxSW(areaT[i + 3], areaT[i + 7], temp);
}
MinMaxSW(areaT[2], areaT[10], temp);	//min
MinMaxSW(areaT[9], areaT[17], temp);	//max
MinMaxSW(areaT[2], areaT[18], temp);	//min
MinMaxSW(areaT[17], areaT[25], temp);	//max

//extra pixel 3 of common area
areaT[minAreaN - 1] = areaT[minAreaN + 2];
//23
```

```
for (i = 3; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
}
for (i = 3; i < (minAreaN - 4); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
}
MinMaxSW(areaT[23], areaT[25], temp);	//min
MinMaxSW(areaT[24], areaT[25], temp);	//max
for (i = 3; i < (minAreaN - 8); i = i + 8)
{
	MinMaxSW(areaT[i], areaT[i + 4], temp);
	MinMaxSW(areaT[i + 3], areaT[i + 7], temp);
}
MinMaxSW(areaT[19], areaT[23], temp);	//min
MinMaxSW(areaT[22], areaT[25], temp);	//max

MinMaxSW(areaT[3], areaT[11], temp);	//min
MinMaxSW(areaT[10], areaT[18], temp);	//max
MinMaxSW(areaT[3], areaT[19], temp);	//min
MinMaxSW(areaT[18], areaT[25], temp);	//max

//extra pixel 4 of common area
areaT[minAreaN - 1] = areaT[minAreaN + 3];
//22
for (i = 4; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
}
for (i = 4; i < (minAreaN - 4); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
}
for (i = 4; i < (minAreaN - 8); i = i + 8)
{
	MinMaxSW(areaT[i], areaT[i + 4], temp);
	MinMaxSW(areaT[i + 3], areaT[i + 7], temp);
}
MinMaxSW(areaT[20], areaT[24], temp);	//min
MinMaxSW(areaT[23], areaT[25], temp);	//max

MinMaxSW(areaT[4], areaT[12], temp);	//min
MinMaxSW(areaT[11], areaT[19], temp);	//max
MinMaxSW(areaT[4], areaT[20], temp);	//min
MinMaxSW(areaT[19], areaT[25], temp);	//max

//extra pixel 5 of common area
areaT[minAreaN - 1] = areaT[minAreaN + 4];
//21
for (i = 5; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
}
for (i = 5; i < (minAreaN - 4); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
}
for (i = 5; i < (minAreaN - 8); i = i + 8)
```

```
{
	MinMaxSW(areaT[i], areaT[i + 4], temp);
	MinMaxSW(areaT[i + 3], areaT[i + 7], temp);
}
MinMaxSW(areaT[21], areaT[25], temp);	//min
MinMaxSW(areaT[24], areaT[25], temp);	//max

MinMaxSW(areaT[5], areaT[13], temp);	//min
MinMaxSW(areaT[12], areaT[20], temp);	//max
MinMaxSW(areaT[5], areaT[21], temp);	//min
MinMaxSW(areaT[20], areaT[25], temp);	//max

//extra pixel 6 of common area
areaT[minAreaN - 1] = areaT[minAreaN + 5];
//20
for (i = 6; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
}
for (i = 6; i < (minAreaN - 3); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
}
for (i = 6; i < (minAreaN - 8); i = i + 8)
{
	MinMaxSW(areaT[i], areaT[i + 4], temp);
	MinMaxSW(areaT[i + 3], areaT[i + 7], temp);
}
MinMaxSW(areaT[6], areaT[14], temp);	//min
MinMaxSW(areaT[13], areaT[21], temp);	//max
MinMaxSW(areaT[6], areaT[22], temp);	//min
MinMaxSW(areaT[21], areaT[25], temp);	//max

//extra pixel 7 of common area
areaT[minAreaN - 1] = areaT[minAreaN + 6];
//19
for (i = 7; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
}
for (i = 7; i < (minAreaN - 4); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
}
MinMaxSW(areaT[23], areaT[25], temp);	//min
MinMaxSW(areaT[24], areaT[25], temp);	//max
for (i = 7; i < (minAreaN - 8); i = i + 8)
{
	MinMaxSW(areaT[i], areaT[i + 4], temp);
	MinMaxSW(areaT[i + 3], areaT[i + 7], temp);
}
MinMaxSW(areaT[7], areaT[15], temp);	//min
MinMaxSW(areaT[14], areaT[22], temp);	//max
MinMaxSW(areaT[7], areaT[23], temp);	//min
MinMaxSW(areaT[22], areaT[25], temp);	//max

//extra pixel 8 of common area
areaT[minAreaN - 1] = areaT[minAreaN + 7];
//18
for (i = 8; i<(minAreaN - 1); i = i + 2)
```

```
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
}
for (i = 8; i < (minAreaN - 4); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
}
for (i = 8; i < (minAreaN - 8); i = i + 8)
{
	MinMaxSW(areaT[i], areaT[i + 4], temp);
	MinMaxSW(areaT[i + 3], areaT[i + 7], temp);
}
MinMaxSW(areaT[8], areaT[16], temp);	//min
MinMaxSW(areaT[15], areaT[23], temp);	//max
MinMaxSW(areaT[8], areaT[24], temp);	//min
MinMaxSW(areaT[23], areaT[25], temp);	//max

//extra pixel 9 of common area
areaT[minAreaN - 1] = areaT[minAreaN + 8];
//17
for (i = 9; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
}
for (i = 9; i < (minAreaN - 4); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
}
for (i = 9; i < (minAreaN - 8); i = i + 8)
{
	MinMaxSW(areaT[i], areaT[i + 4], temp);
	MinMaxSW(areaT[i + 3], areaT[i + 7], temp);
}
MinMaxSW(areaT[9], areaT[17], temp);	//min
MinMaxSW(areaT[16], areaT[24], temp);	//max
MinMaxSW(areaT[9], areaT[25], temp);	//min
MinMaxSW(areaT[24], areaT[25], temp);	//max

//extra pixel 10 of common area
areaT[minAreaN - 1] = areaT[minAreaN + 9];
//16
for (i = 10; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
}
for (i = 10; i < (minAreaN - 3); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
}
for (i = 10; i < (minAreaN - 7); i = i + 8)
{
	MinMaxSW(areaT[i], areaT[i + 4], temp);
	MinMaxSW(areaT[i + 3], areaT[i + 7], temp);
}
MinMaxSW(areaT[10], areaT[18], temp);	//min
MinMaxSW(areaT[17], areaT[25], temp);	//max

//Multy pixel per thread
for (a = 0; a<(k - 1); a++)
```

```
{
	for (b = 0; b<(k - 1); b++)
	{
		areaD[a*(k - 1) + b] = areaT[a*(k - 1) + b];
	}
}
//upper and low
//pixel 1
areaT[minAreaN - 1] = areaA[1];
areaD[minAreaN - 1] = areaA[k*(k + 1) + 1];
//15
for (i = 11; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
	MinMaxSW(areaD[i], areaD[i + 1], temp);
}
for (i = 11; i < (minAreaN - 4); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
	MinMaxSW(areaD[i], areaD[i + 2], temp);
	MinMaxSW(areaD[i + 1], areaD[i + 3], temp);
}
MinMaxSW(areaT[23], areaT[25], temp);    //min
MinMaxSW(areaT[24], areaT[25], temp);    //max
MinMaxSW(areaD[23], areaD[25], temp);    //min
MinMaxSW(areaD[24], areaD[25], temp);    //max

MinMaxSW(areaT[11], areaT[15], temp);    //min
MinMaxSW(areaT[14], areaT[18], temp);    //max
MinMaxSW(areaT[19], areaT[23], temp);    //min
MinMaxSW(areaT[12], areaT[25], temp);    //max
MinMaxSW(areaT[11], areaT[19], temp);    //min
MinMaxSW(areaT[18], areaT[25], temp);    //max
MinMaxSW(areaD[11], areaD[15], temp);    //min
MinMaxSW(areaD[14], areaD[18], temp);    //max
MinMaxSW(areaD[19], areaD[23], temp);    //min
MinMaxSW(areaD[12], areaD[25], temp);    //max
MinMaxSW(areaD[11], areaD[19], temp);    //min
MinMaxSW(areaD[18], areaD[25], temp);    //max

//pixel 2
areaT[minAreaN - 1] = areaA[2];
areaD[minAreaN - 1] = areaA[k*(k + 1) + 2];
//14
for (i = 12; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
	MinMaxSW(areaD[i], areaD[i + 1], temp);
}
for (i = 12; i < (minAreaN - 4); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
	MinMaxSW(areaD[i], areaD[i + 2], temp);
	MinMaxSW(areaD[i + 1], areaD[i + 3], temp);
}
MinMaxSW(areaT[12], areaT[16], temp);    //min
MinMaxSW(areaT[15], areaT[19], temp);    //max
MinMaxSW(areaT[20], areaT[24], temp);    //min
MinMaxSW(areaT[23], areaT[25], temp);    //max
MinMaxSW(areaT[12], areaT[20], temp);    //min
```

```
MinMaxSW(areaT[19], areaT[25], temp);    //max
MinMaxSW(areaD[12], areaD[16], temp);    //min
MinMaxSW(areaD[15], areaD[19], temp);    //max
MinMaxSW(areaD[20], areaD[24], temp);    //min
MinMaxSW(areaD[23], areaD[25], temp);    //max
MinMaxSW(areaD[12], areaD[20], temp);    //min
MinMaxSW(areaD[19], areaD[25], temp);    //max

//pixel 3
areaT[minAreaN - 1] = areaA[3];
areaD[minAreaN - 1] = areaA[k*(k + 1) + 3];
//13
for (i = 13; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
	MinMaxSW(areaD[i], areaD[i + 1], temp);
}
for (i = 13; i < (minAreaN - 4); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
	MinMaxSW(areaD[i], areaD[i + 2], temp);
	MinMaxSW(areaD[i + 1], areaD[i + 3], temp);
}
MinMaxSW(areaT[13], areaT[17], temp);    //min
MinMaxSW(areaT[16], areaT[20], temp);    //max
MinMaxSW(areaT[21], areaT[25], temp);    //min
MinMaxSW(areaT[24], areaT[25], temp);    //max
MinMaxSW(areaT[13], areaT[21], temp);    //min
MinMaxSW(areaT[20], areaT[25], temp);    //max
MinMaxSW(areaD[13], areaD[17], temp);    //min
MinMaxSW(areaD[16], areaD[20], temp);    //max
MinMaxSW(areaD[21], areaD[25], temp);    //min
MinMaxSW(areaD[24], areaD[25], temp);    //max
MinMaxSW(areaD[13], areaD[21], temp);    //min
MinMaxSW(areaD[20], areaD[25], temp);    //max

//pixel 4
areaT[minAreaN - 1] = areaA[4];
areaD[minAreaN - 1] = areaA[k*(k + 1) + 4];
//12
for (i = 14; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
	MinMaxSW(areaD[i], areaD[i + 1], temp);
}
for (i = 14; i < (minAreaN - 3); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
	MinMaxSW(areaD[i], areaD[i + 2], temp);
	MinMaxSW(areaD[i + 1], areaD[i + 3], temp);
}
MinMaxSW(areaT[14], areaT[18], temp);    //min
MinMaxSW(areaT[17], areaT[21], temp);    //max
MinMaxSW(areaT[14], areaT[22], temp);    //min
MinMaxSW(areaT[21], areaT[25], temp);    //max
MinMaxSW(areaD[14], areaD[18], temp);    //min
MinMaxSW(areaD[17], areaD[21], temp);    //max
MinMaxSW(areaD[14], areaD[22], temp);    //min
MinMaxSW(areaD[21], areaD[25], temp);    //max
```

```
//pixel 5
areaT[minAreaN - 1] = areaA[5];
areaD[minAreaN - 1] = areaA[k*(k + 1) + 5];
//11
for (i = 15; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
	MinMaxSW(areaD[i], areaD[i + 1], temp);
}
for (i = 15; i < (minAreaN - 4); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
	MinMaxSW(areaD[i], areaD[i + 2], temp);
	MinMaxSW(areaD[i + 1], areaD[i + 3], temp);
}
MinMaxSW(areaT[23], areaT[25], temp);    //min
MinMaxSW(areaT[24], areaT[25], temp);    //max
MinMaxSW(areaD[23], areaD[25], temp);    //min
MinMaxSW(areaD[24], areaD[25], temp);    //max

MinMaxSW(areaT[15], areaT[19], temp);    //min
MinMaxSW(areaT[18], areaT[22], temp);    //max
MinMaxSW(areaT[15], areaT[23], temp);    //min
MinMaxSW(areaT[22], areaT[25], temp);    //max
MinMaxSW(areaD[15], areaD[19], temp);    //min
MinMaxSW(areaD[18], areaD[22], temp);    //max
MinMaxSW(areaD[15], areaD[23], temp);    //min
MinMaxSW(areaD[22], areaD[25], temp);    //max

//pixel 6
areaT[minAreaN - 1] = areaA[6];
areaD[minAreaN - 1] = areaA[k*(k + 1) + 6];
//10
for (i = 16; i<(minAreaN - 1); i = i + 2)
{
	MinMaxSW(areaT[i], areaT[i + 1], temp);
	MinMaxSW(areaD[i], areaD[i + 1], temp);
}
for (i = 16; i < (minAreaN - 4); i = i + 4)
{
	MinMaxSW(areaT[i], areaT[i + 2], temp);
	MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
	MinMaxSW(areaD[i], areaD[i + 2], temp);
	MinMaxSW(areaD[i + 1], areaD[i + 3], temp);
}
MinMaxSW(areaT[16], areaT[20], temp);    //min
MinMaxSW(areaT[19], areaT[23], temp);    //max
MinMaxSW(areaT[16], areaT[24], temp);    //min
MinMaxSW(areaT[23], areaT[25], temp);    //max
MinMaxSW(areaD[16], areaD[20], temp);    //min
MinMaxSW(areaD[19], areaD[23], temp);    //max
MinMaxSW(areaD[16], areaD[24], temp);    //min
MinMaxSW(areaD[23], areaD[25], temp);    //max


for (a = 0; a<k - 1; a++)

{
	for (b = 0; b<k - 1; b++)
	{
		areaTD[a*(k - 1) + b] = areaT[a*(k - 1) + b];
```

```
                areaDD[a*(k - 1) + b] = areaD[a*(k - 1) + b];
        }
}
/////p1
areaT[minAreaN - 1] = areaA[0];
areaTD[minAreaN - 1] = areaA[(k + 1) - 1];
areaD[minAreaN - 1] = areaA[k + 1];
areaDD[minAreaN - 1] = areaA[2 * (k + 1) - 1];
//9
for (i = 17; i<(minAreaN - 1); i = i + 2)
{
        MinMaxSW(areaT[i], areaT[i + 1], temp);
        MinMaxSW(areaD[i], areaD[i + 1], temp);
        MinMaxSW(areaTD[i], areaTD[i + 1], temp);
        MinMaxSW(areaDD[i], areaDD[i + 1], temp);
}
for (i = 17; i < (minAreaN - 4); i = i + 4)
{
        MinMaxSW(areaT[i], areaT[i + 2], temp);
        MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
        MinMaxSW(areaD[i], areaD[i + 2], temp);
        MinMaxSW(areaD[i + 1], areaD[i + 3], temp);
        MinMaxSW(areaTD[i], areaTD[i + 2], temp);
        MinMaxSW(areaTD[i + 1], areaTD[i + 3], temp);
        MinMaxSW(areaDD[i], areaDD[i + 2], temp);
        MinMaxSW(areaDD[i + 1], areaDD[i + 3], temp);
}
MinMaxSW(areaT[17], areaT[21], temp);    //min
MinMaxSW(areaT[20], areaT[24], temp);    //max
MinMaxSW(areaT[17], areaT[25], temp);    //min
MinMaxSW(areaT[24], areaT[25], temp);    //max
MinMaxSW(areaD[17], areaD[21], temp);    //min
MinMaxSW(areaD[20], areaD[24], temp);    //max
MinMaxSW(areaD[17], areaD[25], temp);    //min
MinMaxSW(areaD[24], areaD[25], temp);    //max
MinMaxSW(areaTD[17], areaTD[21], temp);  //min
MinMaxSW(areaTD[20], areaTD[24], temp);  //max
MinMaxSW(areaTD[17], areaTD[25], temp);  //min
MinMaxSW(areaTD[24], areaTD[25], temp);  //max
MinMaxSW(areaDD[17], areaDD[21], temp);  //min
MinMaxSW(areaDD[20], areaDD[24], temp);  //max
MinMaxSW(areaDD[17], areaDD[25], temp);  //min
MinMaxSW(areaDD[24], areaDD[25], temp);  //max

//p2
areaT[minAreaN - 1] = areaA[(k + 1)];
areaTD[minAreaN - 1] = areaA[2 * (k + 1) - 1];
areaD[minAreaN - 1] = areaA[2 * (k + 1)];
areaDD[minAreaN - 1] = areaA[3 * (k + 1) - 1];
//8
for (i = 18; i<(minAreaN - 1); i = i + 2)
{
        MinMaxSW(areaT[i], areaT[i + 1], temp);
        MinMaxSW(areaD[i], areaD[i + 1], temp);
        MinMaxSW(areaTD[i], areaTD[i + 1], temp);
        MinMaxSW(areaDD[i], areaDD[i + 1], temp);
}
for (i = 18; i < (minAreaN - 3); i = i + 4)
{
        MinMaxSW(areaT[i], areaT[i + 2], temp);
        MinMaxSW(areaT[i + 1], areaT[i + 3], temp);
        MinMaxSW(areaD[i], areaD[i + 2], temp);
```

```
        MinMaxSW(areaD[i + 1], areaD[i + 3], temp);
        MinMaxSW(areaTD[i], areaTD[i + 2], temp);
        MinMaxSW(areaTD[i + 1], areaTD[i + 3], temp);
        MinMaxSW(areaDD[i], areaDD[i + 2], temp);
        MinMaxSW(areaDD[i + 1], areaDD[i + 3], temp);
}
MinMaxSW(areaT[18], areaT[22], temp);    //min
MinMaxSW(areaT[21], areaT[25], temp);    //max
MinMaxSW(areaD[18], areaD[22], temp);    //min
MinMaxSW(areaD[21], areaD[25], temp);    //max
MinMaxSW(areaTD[18], areaTD[22], temp);  //min
MinMaxSW(areaTD[21], areaTD[25], temp);  //max
MinMaxSW(areaDD[18], areaDD[22], temp);  //min
MinMaxSW(areaDD[21], areaDD[25], temp);  //max

//p3
areaT[minAreaN - 1] = areaA[2 * (k + 1)];
areaTD[minAreaN - 1] = areaA[3 * (k + 1) - 1];
areaD[minAreaN - 1] = areaA[3 * (k + 1)];
areaDD[minAreaN - 1] = areaA[4 * (k + 1) - 1];
//7
for (i = 19; i<(minAreaN - 1); i = i + 2)
{
        MinMaxSW(areaT[i], areaT[i + 1], temp);
        MinMaxSW(areaD[i], areaD[i + 1], temp);
        MinMaxSW(areaTD[i], areaTD[i + 1], temp);
        MinMaxSW(areaDD[i], areaDD[i + 1], temp);
}
MinMaxSW(areaT[19], areaT[21], temp);    //min
MinMaxSW(areaT[20], areaT[22], temp);    //max
MinMaxSW(areaT[23], areaT[25], temp);    //min
MinMaxSW(areaT[24], areaT[25], temp);    //max
MinMaxSW(areaT[19], areaT[23], temp);    //min
MinMaxSW(areaT[22], areaT[25], temp);    //max
MinMaxSW(areaD[19], areaD[21], temp);    //min
MinMaxSW(areaD[20], areaD[22], temp);    //max
MinMaxSW(areaD[23], areaD[25], temp);    //min
MinMaxSW(areaD[24], areaD[25], temp);    //max
MinMaxSW(areaD[19], areaD[23], temp);    //min
MinMaxSW(areaD[22], areaD[25], temp);    //max
MinMaxSW(areaTD[19], areaTD[21], temp);  //min
MinMaxSW(areaTD[20], areaTD[22], temp);  //max
MinMaxSW(areaTD[23], areaTD[25], temp);  //min
MinMaxSW(areaTD[24], areaTD[25], temp);  //max
MinMaxSW(areaTD[19], areaTD[23], temp);  //min
MinMaxSW(areaTD[22], areaTD[25], temp);  //max
MinMaxSW(areaDD[19], areaDD[21], temp);  //min
MinMaxSW(areaDD[20], areaDD[22], temp);  //max
MinMaxSW(areaDD[23], areaDD[25], temp);  //min
MinMaxSW(areaDD[24], areaDD[25], temp);  //max
MinMaxSW(areaDD[19], areaDD[23], temp);  //min
MinMaxSW(areaDD[22], areaDD[25], temp);  //max

//p4
areaT[minAreaN - 1] = areaA[3 * (k + 1)];
areaTD[minAreaN - 1] = areaA[4 * (k + 1) - 1];
areaD[minAreaN - 1] = areaA[4 * (k + 1)];
areaDD[minAreaN - 1] = areaA[5 * (k + 1) - 1];
//6
for (i = 20; i<(minAreaN - 1); i = i + 2)
{
        MinMaxSW(areaT[i], areaT[i + 1], temp);
```

```
        MinMaxSW(areaD[i], areaD[i + 1], temp);
        MinMaxSW(areaTD[i], areaTD[i + 1], temp);
        MinMaxSW(areaDD[i], areaDD[i + 1], temp);
}
MinMaxSW(areaT[20], areaT[22], temp);    //min
MinMaxSW(areaT[21], areaT[23], temp);    //max
MinMaxSW(areaT[20], areaT[24], temp);    //min
MinMaxSW(areaT[23], areaT[25], temp);    //max
MinMaxSW(areaD[20], areaD[22], temp);    //min
MinMaxSW(areaD[21], areaD[23], temp);    //max
MinMaxSW(areaD[20], areaD[24], temp);    //min
MinMaxSW(areaD[23], areaD[25], temp);    //max
MinMaxSW(areaTD[20], areaTD[22], temp);  //min
MinMaxSW(areaTD[21], areaTD[23], temp);  //max
MinMaxSW(areaTD[20], areaTD[24], temp);  //min
MinMaxSW(areaTD[23], areaTD[25], temp);  //max
MinMaxSW(areaDD[20], areaDD[22], temp);  //min
MinMaxSW(areaDD[21], areaDD[23], temp);  //max
MinMaxSW(areaDD[20], areaDD[24], temp);  //min
MinMaxSW(areaDD[23], areaDD[25], temp);  //max

//p5
areaT[minAreaN - 1] = areaA[4 * (k + 1)];
areaTD[minAreaN - 1] = areaA[5 * (k + 1) - 1];
areaD[minAreaN - 1] = areaA[5 * (k + 1)];
areaDD[minAreaN - 1] = areaA[6 * (k + 1) - 1];
//5
for (i = 21; i<(minAreaN - 1); i = i + 2)
{
        MinMaxSW(areaT[i], areaT[i + 1], temp);
        MinMaxSW(areaD[i], areaD[i + 1], temp);
        MinMaxSW(areaTD[i], areaTD[i + 1], temp);
        MinMaxSW(areaDD[i], areaDD[i + 1], temp);
}
MinMaxSW(areaT[21], areaT[23], temp);    //min
MinMaxSW(areaT[22], areaT[24], temp);    //max
MinMaxSW(areaT[21], areaT[25], temp);    //min
MinMaxSW(areaT[22], areaT[25], temp);    //max
MinMaxSW(areaD[21], areaD[23], temp);    //min
MinMaxSW(areaD[22], areaD[24], temp);    //max
MinMaxSW(areaD[21], areaD[25], temp);    //min
MinMaxSW(areaD[22], areaD[25], temp);    //max
MinMaxSW(areaTD[21], areaTD[23], temp);  //min
MinMaxSW(areaTD[22], areaTD[24], temp);  //max
MinMaxSW(areaTD[21], areaTD[25], temp);  //min
MinMaxSW(areaTD[22], areaTD[25], temp);  //max
MinMaxSW(areaDD[21], areaDD[23], temp);  //min
MinMaxSW(areaDD[22], areaDD[24], temp);  //max
MinMaxSW(areaDD[21], areaDD[25], temp);  //min
MinMaxSW(areaDD[22], areaDD[25], temp);  //max

//p6
areaT[minAreaN - 1] = areaA[5 * (k + 1)];
areaTD[minAreaN - 1] = areaA[6 * (k + 1) - 1];
areaD[minAreaN - 1] = areaA[6 * (k + 1)];
areaDD[minAreaN - 1] = areaA[7 * (k + 1) - 1];
//4
MinMaxSW(areaT[22], areaT[23], temp);    //min
MinMaxSW(areaT[24], areaT[25], temp);    //max
MinMaxSW(areaT[22], areaT[24], temp);    //min
MinMaxSW(areaT[23], areaT[25], temp);    //max
MinMaxSW(areaD[22], areaD[23], temp);    //min
```

```
        MinMaxSW(areaD[24], areaD[25], temp);    //max
        MinMaxSW(areaD[22], areaD[24], temp);    //min
        MinMaxSW(areaD[23], areaD[25], temp);    //max
        MinMaxSW(areaTD[22], areaTD[23], temp);  //min
        MinMaxSW(areaTD[24], areaTD[25], temp);  //max
        MinMaxSW(areaTD[22], areaTD[24], temp);  //min
        MinMaxSW(areaTD[23], areaTD[25], temp);  //max
        MinMaxSW(areaDD[22], areaDD[23], temp);  //min
        MinMaxSW(areaDD[24], areaDD[25], temp);  //max
        MinMaxSW(areaDD[22], areaDD[24], temp);  //min
        MinMaxSW(areaDD[23], areaDD[25], temp);  //max

        //p7
        areaT[minAreaN - 1] = areaA[6 * (k + 1)];
        areaTD[minAreaN - 1] = areaA[7 * (k + 1) - 1];
        areaD[minAreaN - 1] = areaA[7 * (k + 1)];
        areaDD[minAreaN - 1] = areaA[8 * (k + 1) - 1];
        //3

        MinMaxSW(areaT[23], areaT[24], temp);    //min
        MinMaxSW(areaT[24], areaT[25], temp);    //max
        MinMaxSW(areaT[23], areaT[24], temp);    //min
        MinMaxSW(areaD[23], areaD[24], temp);    //min
        MinMaxSW(areaD[24], areaD[25], temp);    //max
        MinMaxSW(areaD[23], areaD[24], temp);    //min
        MinMaxSW(areaTD[23], areaTD[24], temp);  //min
        MinMaxSW(areaTD[24], areaTD[25], temp);  //max
        MinMaxSW(areaTD[23], areaTD[24], temp);  //min
        MinMaxSW(areaDD[23], areaDD[24], temp);  //min
        MinMaxSW(areaDD[24], areaDD[25], temp);  //max
        MinMaxSW(areaDD[23], areaDD[24], temp);  //min

        /////////////////
        //Colocar valor de Mediana
        filmedCuda[x*hor + y] = areaT[minAreaN - 2];
        filmedCuda[x*hor + y + 1] = areaTD[minAreaN - 2];
        filmedCuda[(x + 1)*hor + y] = areaD[minAreaN - 2];
        filmedCuda[(x + 1)*hor + y + 1] = areaDD[minAreaN - 2];
}

int main(void) {

        printf("Median filter with k=%i\n", k);
        printf("Image size %i x %i\n", ver, hor);
        printf("Padded Image size %i x %i\n", verp, horp);
        //Variables
        FILE *ini;
        float *R, *G, *B, *filmed, *ImBase;
        char archivo[512] = "bmw3840BK01.tiff";
        int i;
        int x, y;     //indice de ubicacion del pixel a operar
        float error;
        int z, error2;

        // CUDA Vars
        cudaError_t cudaerr;
        float *ImBaseCuda, *filmedCuda;
        float *filmedHostFromCuda;
        int numBlocks;
        float mpix;
```

```
	// Timing using cudaEvent
	cudaEvent_t start, stop;
	float gpu_time;

	cudaEventCreate(&start);
	cudaEventCreate(&stop);

	printf("Median filter with k=%i\n", k);

	//Memoria para imagen
	filmed = (float *)malloc(ver*hor * sizeof(float));
	ImBase = (float *)calloc((verp)*(horp), sizeof(float)); // Increase
size to (v+k-1) x (h+k-1) or coalesence


			// Abrir archivo
	if (fopen_s(&ini, archivo, "rb") == 0)
	{
		printf("%s abierto\n", archivo);
	}
	else
	{
		printf("%s fallo al abrirse\n", archivo);
		_getch();
		return(-1);
	}

	// Reserva espacio en memoria para la imagen
	R = (float *)malloc(ver*hor * sizeof(float));
	G = (float *)malloc(ver*hor * sizeof(float));
	B = (float *)malloc(ver*hor * sizeof(float));
	//////////////////////

	// Separa los canales de la imagen
	fseek(ini, 8L, SEEK_SET);

	for (i = 0; i<ver*hor; i++)//decide the channel
	{
		G[i] = (float)fgetc(ini);
	}

	// Fill with zeros the extra borders (v+k-1 x h+k-1)
	for (x = 0; x<ver; x++)
	{
		for (y = 0; y<hor; y++)
		{
			ImBase[(x + (k - 1) / 2)*(horp)+y + (k - 1) / 2] = G[x*hor
+ y];//the letter depends of the channel
		}
	}

	// CUDA processing

	// Allocate memory on the host
	filmedHostFromCuda = (float *)malloc(ver*hor * sizeof(float));

	// Allocate memory in cuda
	cudaerr = cudaMalloc((void**)&ImBaseCuda, sizeof(float)*(verp)*(horp));
// Input image padded
	if (cudaerr != 0)	printf("ERROR allocating memory for ImBaseCuda.
CudaMalloc value=%i\n\r", cudaerr);
```

```
	cudaerr = cudaMalloc((void**)&filmedCuda, sizeof(float)*ver*hor); //
Filtered image
	if (cudaerr != 0)	printf("ERROR allocating memory for filmedCuda.
CudaMalloc value=%i\n\r", cudaerr);

	// Move padded input image from Host to Device
	cudaerr = cudaMemcpy(ImBaseCuda, ImBase, sizeof(float)*(verp)*(horp),
cudaMemcpyHostToDevice);
	if (cudaerr != 0)	printf("ERROR copying ImBase to ImBaseCuda (Host to
Dev). CudaMalloc value=%i\n\r", cudaerr);

	// Launch kernels
	if (hor == ver) numBlocks = (hor * ver / threadsPerBlock) /
pixelsPerThread;//integer
	else numBlocks = (((hor*ver) + threadsPerBlock - 1) / threadsPerBlock)
/ pixelsPerThread; // integer

	switch (k)
	{
	case 3:
		//Cache or Shared
		cudaFuncSetCacheConfig(SorNet_GPU_L3, cudaFuncCachePreferL1);
		//
		SorNet_GPU_L3 << <numBlocks, threadsPerBlock >> >(filmedCuda,
ImBaseCuda);
		//Timing
		cudaEventRecord(start);
		//
		SorNet_GPU_L3 << <numBlocks, threadsPerBlock >> >(filmedCuda,
ImBaseCuda);
		cudaEventRecord(stop);
		cudaEventSynchronize(stop);
		break;
	case 5:
		//Cache or Shared
		cudaFuncSetCacheConfig(SorNet_GPU_L5, cudaFuncCachePreferL1);
		//
		SorNet_GPU_L5 << <numBlocks, threadsPerBlock >> >(filmedCuda,
ImBaseCuda);
		//Timing
		cudaEventRecord(start);
		//
		SorNet_GPU_L5 << <numBlocks, threadsPerBlock >> >(filmedCuda,
ImBaseCuda);
		cudaEventRecord(stop);
		cudaEventSynchronize(stop);
		break;
	case 7:
		//Cache or Shared
		cudaFuncSetCacheConfig(SorNet_GPU_L7, cudaFuncCachePreferL1);
		//
		SorNet_GPU_L7 << <numBlocks, threadsPerBlock >> >(filmedCuda,
ImBaseCuda);
		//Timing
		cudaEventRecord(start);
		//
		//for (i = 0; i < 100; i++) {
			SorNet_GPU_L7 << <numBlocks, threadsPerBlock >> >
(filmedCuda, ImBaseCuda);
		//}
		cudaEventRecord(stop);
		cudaEventSynchronize(stop);
```

```
			break;
		}
		cudaEventElapsedTime(&gpu_time, start, stop);
		mpix = ((ver*hor) / gpu_time) / 1000;
		printf("GPU Time:  %fms\n\r", gpu_time);
		printf("GPU speed:  %fMpix/s\n\r", mpix);
		// Move Array from Device to Host
		cudaerr = cudaMemcpy(filmedHostFromCuda, filmedCuda,
sizeof(float)*ver*hor, cudaMemcpyDeviceToHost);
		if (cudaerr != 0)    printf("ERROR copying filmedCuda to
filmedHostFromCuda (Dev to Host). CudaMalloc value=%i\n\r", cudaerr);

		//Se escribe datos cuda

		FILE *arre2 = fopen("bmw3840BKarre2.tiff", "rb+");
		if (arre2 == NULL)
		{
			printf("error en abrir archivo 2 \n");
		}
		else
		{
			fseek(arre2, 8L, SEEK_SET);// Offset to data
			for (x = 0; x<ver; x++)                    // Write results in file
			{
				for (y = 0; y<hor; y++)
				{
					fputc((unsigned char)filmedHostFromCuda[x*hor + y],
arre2);
				}
			}
		}

		// Finalize
		free(R);
		free(G);
		free(B);
		free(filmed);
		free(ImBase);
		fclose(ini);
		fflush(arre2);
		fclose(arre2);

		_getch(); // Pause to see the console window
		return(0);

}
```