UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

PROGRAMACIÓN DE ALTO DESEMPEÑO DEL FILTRO C-MACE EN UNA UNIDAD DE PROCESAMIENTO GRÁFICO

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELECTRICISTA

DANIEL OSCAR BAEZA MARTÍNEZ

PROFESOR GUÍA:
PABLO ESTÉVEZ VALENCIA

MIEMBROS DE LA COMISIÓN:
HÉCTOR AGUSTO ALEGRÍA
PABLO ZEGERS FERNÁNDEZ

Este trabajo ha sido parcialmente financiado por FONDECYT 1110701

SANTIAGO DE CHILE
MARZO 2013

Resumen

En el presente trabajo de memoria se realiza una implementación del filtro CMACE (*Correntropy-Minimum Average Correlation Energy*) en una Unidad de Procesamiento Gráfico o GPU (*Graphics Processor Unit*), con el objetivo de reducir los costos computacionales asociados con la ejecución de este filtro. Los grandes tiempos de espera que tiene el filtro CMACE en su implementación en serie hace que esta herramienta sea poco práctica en la gran mayoría de los problemas de clasificación y reconocimiento de imágenes. La reducción de los costos computacionales mediante la implementación en GPU del filtro CMACE pretende hacer de este filtro una herramienta mucho más útil en problemas de ingeniería orientados al procesamiento de imágenes. La supercomputación mediante GPU está haciendo posible usar herramientas computacionalmente costosas en tiempos mucho más reducidos sin sacrificar las cualidades de los algoritmos.

La implementación se realiza en una tarjeta Nvidia Tesla C2050, utilizando el lenguajes de programación C y la extensión CUDA hecha por Nvidia para la programación de GPU. La implementación final del filtro es híbrida, en donde se mezclan implementaciones en GPU y CPU para aprovechar las características de cada uno de estos dispositivos en distintos tipos de procesamiento de datos. Además de la implementación, de realizan comparaciones de desempeño frente a las implementaciones tradicionales en CPU y pruebas de validación para ver el poder de clasificación que tiene el filtro CMACE. Además se realizan pruebas con preprocesamiento de las imágenes mediante una reducción dimensional.,con el fin de reducir la carga de los procesos que tienen los dispositivos.

Los resultados obtenidos en la implementación del filtro CMACE en GPU muestran que esta implementación es 16 veces más rápida que la implementación tradicional en CPU .

En conclusión, la GPU da una solución a los tiempos de espera que se tienen en el uso del filtro CMACE, haciendo de este filtro una herramienta mucho más útil y práctica.

Dedicada a la memoria de mi abuela y madre Edith ...

Agradecimientos

Aprovechando esta instancia, quisiera agradecer a las personas que me han acompañado en este proceso.
En primer lugar, quisiera agradecer a mi familia. Gracias a mis viejos por apoyarme, confiar en mi siempre, incluso en momentos que ni yo lo hacía y quererme con la fuerza que siento que lo hacen. Gracias a mi tío Claudio, un tipo admirable que siempre ha estado para darme consejos(instrucciones), recordarme y enseñarme a perceberar hasta alcanzar los objetivos. Hermanos, tíos, primos, a todos gracias por ser mi familia, darme el cariño y los valores que me han acompañado hasta ahora.

En segundo lugar quisiera agradecer a mis amigos, esa pequeña familia que uno elije a lo largo de la vida. Gracias Pelao, Wenche, Caro, Pauly, Nicolás, Joaco e Italo. El círculo de hierro, amigos casi hermanos que siempre están ahí para todo.

Agradecer también a la gente del laboratorio de imágenes e inteligencia computacional por haber hecho del tiempo que pasé ahí una etapa inolvidable, en especial a Pablo, Rafa y Jorge.

Para finalizar quisiera agradecer a dos profesores que han sido muy importantes. Gracias a mi profesor guía Dr. Pablo Estévez por darme un espacio en el laboratorio que dirige y por otro lado, ser un apoyo importante en esta última etapa de la entrega final del trabajo de memoria. Y por último, agradecer a mi actual jefe el profesor Dr. Julián Ortiz, del cual también he tenido un apoyo muy grande tanto en lo laboral como en el hecho de cerrar este proceso y poder titularme.

Gracias a todos y sepan que siempre pueden contar conmigo.

Daniel

Tabla de contenido

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

Uno de los problemas más comunes en algunas áreas de la ingeniería es el procesamiento masivo de datos, por ejemplo Procesamiento de Imágenes, Minería de Datos, Control, Automatización, Modelamiento, etc, se necesita resolver algoritmos complejos en bajos tiempos de ejecución para dar respuestas en tiempo real. La super computación da una solución a este problema, a través del uso de computadores especializados, con grandes bancos de memorias y decenas de procesadores capaces de trabajar en forma paralela.

Debido principalmente a la industria del video juego, las tarjetas gráficas han tenido un crecimiento sorprendente, mostrando en las últimas versiones cientos de procesadores en paralelo capaces de hacer operaciones sobre conjuntos masivos de datos, en donde no solo se cuentan las estructuras y tipos relacionados con el procesamiento de gráficos, sino además los pensados para el procesamiento de programas ligados a cualquier área de la ingeniería (mecánica, minería, eléctrica, hidráulica, etc).

El Filtro C-MACE (Correntropy Minimum Average Correlation Energy), es un método de clasificación inspirado en el filtro MACE que incorpora nuevos conocimientos entregados por el área de Teoría de Información. Si bien su poder de clasificación muestra mejoras respecto al filtro MACE, los costos computacionales son tan elevados que resulta impráctico en problemas de clasificación en donde se necesite una respuesta rápida ante una señal de entrada.

El presente trabajo "Programación de Alto Desempeño del Filtro C-MACE en una Unidad de Procesamiento Gráfico" consiste en la implementación eficiente del filtro CMACE en una tarjeta de procesamiento gráfico, con el fin de reducir los costos computacionales asociados al uso de este filtro.

1.1. Objetivo General

- Implementar en forma eficiente el filtro C-MACE en una unidad de procesamiento gráfico.

1.2. Objetivos Específicos

- Mejorar el costo computacional del filtro C-MACE mediante el uso de una unidad de procesamiento gráfico con respecto a las implementaciones tradicionales en una CPU.
- Optimizar el uso de recursos computacionales de una unidad de procesamiento gráfico.
- Establecer limitaciones de la implementación del algoritmo CMACE en una unidad de procesamiento gráfico.
- Reducir las limitaciones impuestas por el uso de una Unidad de procesamiento gráfico mediante el preprocesamiento de los datos.
- Probar la capacidad de clasificación y reconocimiento de rostros del filtro C-MACE programado en una unidad de procesamiento gráfico.

1.3. Estructura de la Memoria

La organización del presente trabajo de título "Programación de Alto Desempeño del Filtro C-MACE en una Unidad de Procesamiento Gráfico" es como sigue. En la sección 2 se dan a conocer los antecedentes en los cuales se fundamenta este trabajo, explicando los conceptos de GPU, filtro MACE, filtro CMACE y métricas de computación paralela. En la sección 3 se muestra la metodología usada en el desarrollo de los experimentos, describiendo: las características de los dispositivos, la base de datos de imágenes, el diseño de la paralelización del filtro CMACE, las limitaciones de la paralelización, las pruebas hechas para la validación del filtro CMACE en GPU y las pruebas hechas para comparar el desempeño de la implementación en paralelo con la implementación tradicional en serie. En la sección 4 se muestran los resultados de los experimentos de comparación de desempeño de la implementación en paralelo versus la implementación en serie y las pruebas de validación del filtro CMACE en GPU mediante pruebas de clasificación y el reconocmiento de rostros en imágenes. En la sección 5 se dan a conocer las conclusiones obtenidas del desarrollo del presente trabajo.

Capítulo 2

Antecedentes

2.1. Unidad de Procesamiento Gráfico - GPU

La unidad de procesamiento gráfico o GPU (acrónimo del inglés *Graphics Processing Units*) es un procesador dedicado al tratamiento de gráficos y cómputo de punto flotante, esto con el fin de aligerar la carga de procesos en la CPU de un computador.

En los primeros años, finales de los 80's y principio de los 90's, fue diseñada sólo como un complemento de la CPU en el área gráfica. A finales de los 90's empezó a tener mayor flexibilidad hacia el programador, dando la posiblidad de programarla con algoritmos básicos. En 1999 se construye la primera GPU Nvidia y en menos de un año, el término GPU empieza a ser utilizado ampliamente por artistas gráficos y desarrolladores de juegos. Poco tiempo después se enciende el interés de la comunidad científica por este dispositivo debido a su impresionante rendimiento en el procesamiento de punto flotante. Es aquí en donde nace el desarrollo de GPU con propósito general GPGPU (*General Purpose GPU*).

En los comienzos, la GPGPU estaba fuera del alcance del mundo científico, a menos que se conociera prácticamente de memoria las últimas versiones de APIs gráficas. Aún en estos casos el desarrollo de propósitos generales tenía limitaciones de comunicación al tener que traducir todos los problemas a triángulos y polígonos, ya que en esos tiempos todavía era un dispositivo diseñado para resolver problemas gráficos.

La primera solución al problema de programación con fines genéricos lo dio la Universidad de Stanford en el 2003 con el nombre Brook, el primer modelo de programación con el fin de ampliar la programación en C hacia datos y estructuras en paralelo. Finalmente, en el 2006 Nvidia da a conocer NVIDIA CUDA [ND, NVI], la primera solución para computación general en GPU, en donde lenguaje y arquitectura son diseñados de manera conjunta, haciendo posible la interacción con el dispositivo desde el lenguaje C, un lenguaje amigable y conocido ampliamente por la comunidad científica.

Debido a una gran exigencia en reproducciones en tiempo real y alta definición en

gráficos 3D, las GPU han evolucionado llegando a tener en estos días un alto nivel de paralelismo, procesamiento en varios hilos de ejecución (en inglés *multithreads*), decenas de nucleos (cores) con un gran poder de procesamiento (Figura 2.1) y acceso a memoria con ancho de bandas bastante grandes como lo muestra la Figura 2.2. En el caso de la GPU usada en el presente trabajo, el ancho de banda es de 144 GB/s.

Todas estas características hacen de este procesador una buena alternativa para computación de alto desempeño destinada a la programación en paralelo.

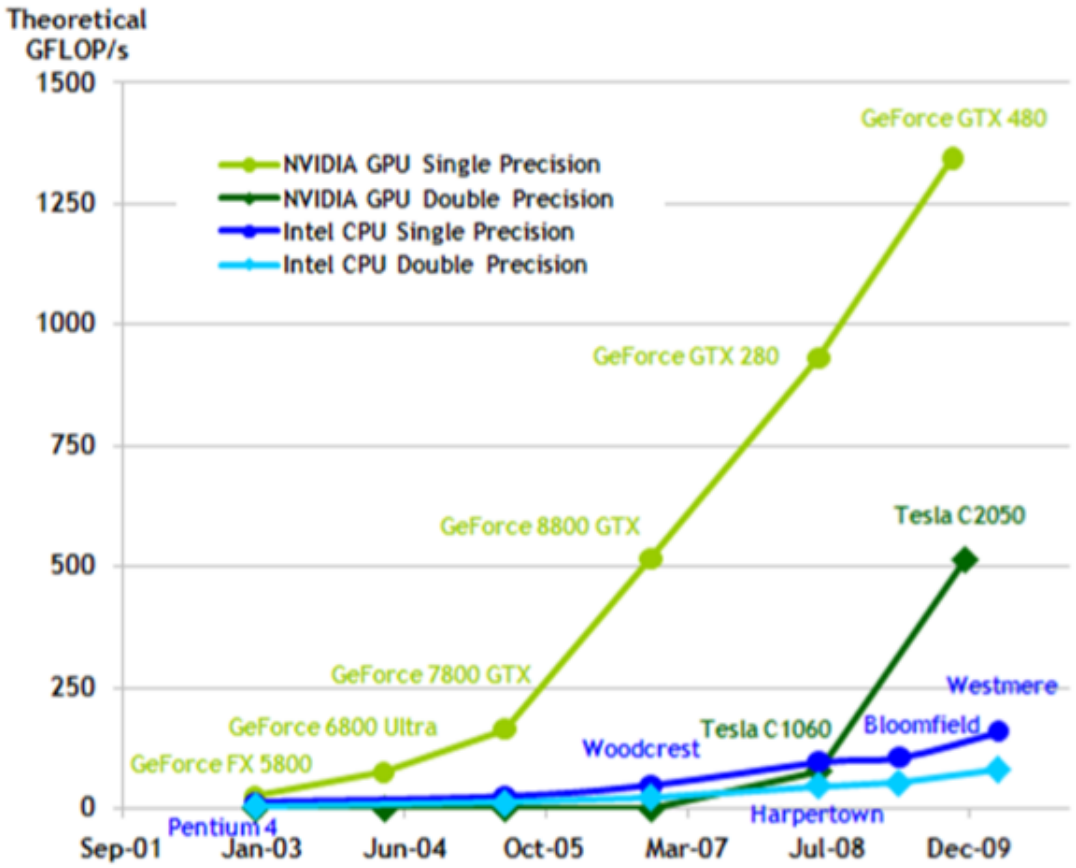


Figura 2.1: Evolución del poder de cómputo de las GPU y de CPU en los últimos años.

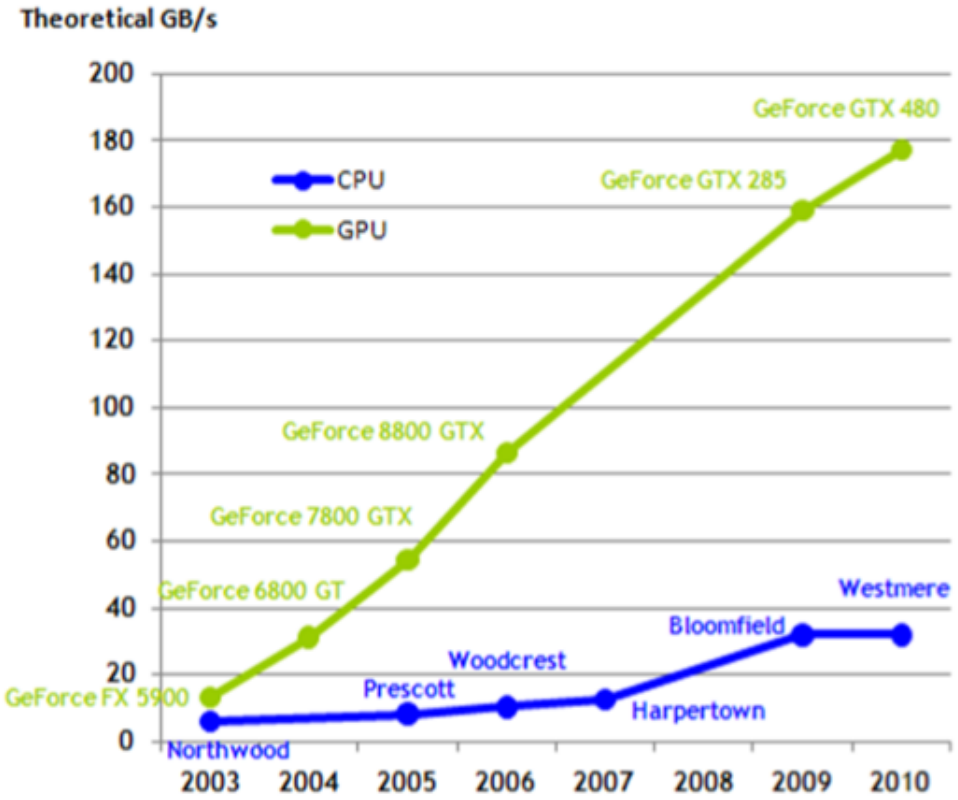


Figura 2.2: Mejora del Ancho de Banda con las memorias de GPU y CPU en los últimos años.

2.2. CUDA

CUDA (*Compute Unified Device Architecture*)es una plataforma de desarrollo y modelo de programación en paralelo creada por Nvidia, principal compañía dedicada a la fabricación de GPUs.

La principal preocupación en la computación paralela es mejorar el rendimiento, es decir reducir los tiempos computacionales requeridos para resolver algún problema en particular.

Mientras la computación paralela clásica tiene una curva de aprendizaje muy empinada, CUDA presenta una solución mucho más cercana al programador, en donde es posible enviar a la GPU un código de C o C++ mediante un conjunto simple de instrucciones. Con esto es posible destinar el tiempo principalmente a diseñar el algoritmo paralelo y no gastar mucho tiempo en aprender a usar una plataforma que permita la comunicación con el dispositivo.

Por medio de un lenguaje de alto nivel, es posible distribuir y dirigir el trabajo secuencial a la CPU, y acelerar los procesos paralelizables con el uso de GPU, optimizando el algoritmo final con el procesamiento en paralelo.

2.3. Filtro MACE

El filtro MACE [LWZ04] (*Minimum Average Correlation Energy*), tal como lo dice su nombre, minimiza la energía de correlación entre la salida del filtro y las imágenes de entrenamiento. El efecto generado por esta minimización son planos muy cercanos a cero, excepto en objetos de la base de entrenamiento que tengan semejanza con la entrada del filtro, en los cuales se producirán fuertes impulsos.

Sea un vector d $\times$ 1 la representación de una imagen bidimensional, en donde d representa la cantidad de pixeles de la imagen ordenados de manera lexicográfica como se muestra en la Figura 2.3.

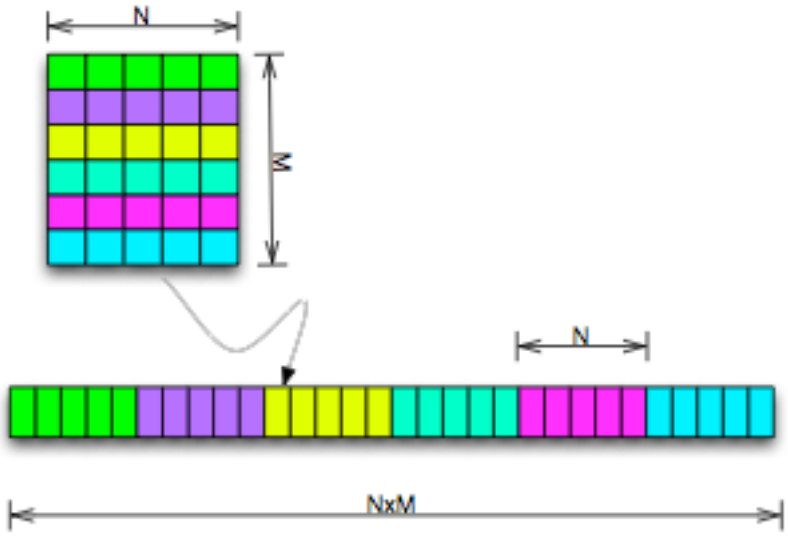


Figura 2.3: Representación de un ordenamiento lexicográfico.

Vamos a denotar la i-ésima imagen de entrenamiento como x_{i} y aplicándole la transformada discreta de Fourier o DFT (acrónimo de *Discrete Fourier Transform*), denotaremos su representación en el dominio de frecuencia como X_{i}. De este modo el conjunto de entrenamiento está dado por la matriz

$$X = \{X_1, X_2, X_3, \ldots, X_N\} \tag{2.1}$$

con dimensión d $\times$ N, en donde N es el número de imágenes de entrenamiento y d son la cantidad de pixeles por cada imagen.

Llamaremos h a la salida del filtro en el espacio de imágenes y H a la salida del filtro en el dominio de la frecuencia. La correlación $\oplus$ entre la i-ésima imagen $x_{\mathrm{i}}(n)$ y el filtro $h(n)$ es:

$$g_{\mathrm{i}}(n) = x_{\mathrm{i}}(n) \oplus h(n) \tag{2.2}$$

Por el teorema de Parseval, la energía de correlación de la i-ésima imagen con el filtro H (dominio de frecuencia) puede ser escrita como:

$$E_{\mathrm{i}} = H^{\mathcal{H}} D_{\mathrm{i}} H \tag{2.3}$$

en donde D_{i} es una matriz diagonal de d $\times$ d en que los elementos de la diagonal representan el cuadrado de la magnitud de cada elemento de X_{i} y el superíndice $\mathcal{H}$ indica la transposición Hermítica de H. El objetivo del filtro MACE es minimizar el promedio de la energía de correlación sobre la imagen, satisfaciendo al mismo tiempo las restricciones de intensidad en el origen de cada imagen.

El valor de la correlación en el origen de cada imagen puede ser expresado como:

$$g_{\mathrm{i}}(0) = X_{\mathrm{i}}^{\mathcal{H}} H = c_{\mathrm{i}} \tag{2.4}$$

para i $= 1, 2, \ldots, N$ imágenes de entrenamiento, en donde c_{i} es la correlación de salida en el origen especificada por el usuario. La energía promedio sobre todas las imágenes de entrenamiento se puede expresar como:

$$E_{prom} = H^{\mathcal{H}} D H \tag{2.5}$$

donde $D = \frac{1}{N} \sum_{\mathrm{i}=1}^{N} D_{\mathrm{i}}$. De esta forma el problema del filtro MACE se resume en minimizar E_{prom}, mientras se satisface la restricción, $X^{\mathcal{H}} H = c$ donde $c = c_1, c_2, \ldots, c_N$ es un vector N-dimensional. Este problema de optimización se puede resolver mediante multiplicadores de Lagrange, con lo que la solución queda expresada con la siguiente ecuación:

$$H = D^{-1} X \left(X^{\mathcal{H}} D^{-1} X \right)^{-1} c \tag{2.6}$$

El filtro h en el espacio se obtiene calculando la DFT inversa de H.

2.4. Correntropía

2.4.1. Definición

Correntropía Cruzada [LPP, JP08] es la medida de semejanza entre dos vectores de variables aleatorias definida como:

$$V(X,Y) = E[k_\sigma(X-Y)]$$

en donde $E[\cdot]$ es la esperanza matemática sobre un proceso estocástico y $k(X-Y)$ es un Kernel que satisface las condiciones del teorema de Mercer [Aro50]. En el presente trabajo se usa un kernel Gaussiano como el que se muestra en la siguiente ecuación:

$$k_\sigma(X-Y) = \frac{1}{\sqrt{2\pi}\sigma}\exp - \left(\frac{\|X-Y\|^2}{2\sigma^2}\right) \tag{2.7}$$

en donde σ es el tamaño del kernel.

En la práctica, para un número finito de muestras $\{x_\mathrm{i}, y_\mathrm{i}\}_{\mathrm{i}=1}^{\mathrm{d}}$, el estimador de correntropía está dado por:

$$\hat{V}(X,Y) = \frac{1}{\mathrm{d}}\sum_{\mathrm{i}=1}^{\mathrm{d}} k_\sigma(x_\mathrm{i} - y_\mathrm{i})$$

2.4.2. Propiedades de la Correntropía

Las demostraciones de las siguientes propiedades son desarrolladas por Gunduz y Weilfeng en [GP09, LPP].

Propiedad 1

Correntropía es una medida de semejanza entre X e Y que incorpora información de los momentos de mayor orden de las variables aleatorias X e Y.

El kernel Gaussiano, expandiéndolo en serie de Taylor, se puede reescribir la función Correntropía como

$$V(X,Y) = \frac{1}{\sqrt{2\pi}\sigma}\sum_{n=0}^{\infty}\frac{(-1)^n}{2^n\sigma^{2n}n!}E\left[\|X-Y\|^{2n}\right] \tag{2.8}$$

Esta última ecuación muestra que usando el kernel Gaussiano, la Función Correntropía involucra a los momentos de mayor orden de la variable aleatoria $\|X-Y\|$. Para $n=1$ en la ecuación 2.8 el término corresponde a:

$$E\left[\|X\|^2\right] + E\left[\|Y\|^2\right] - 2E\left[\langle X, Y\rangle\right] = \sigma^2_{x_{t_1}} + \sigma^2_{x_{t_2}} - 2R_x\left(X, Y\right) \tag{2.9}$$

donde $R_x\left(X, Y\right)$ es la función de covarianza del proceso estocástico, de la ecuación 2.9 se aprecia que la información dada por la función de covarianza está incluida en la función Correntropía.

El tamaño del kernel Gaussiano controla la importancia que se le da a los momentos de mayor orden respecto al de segundo orden. A medida que σ aumenta, la función Correntropía tiende a la función de correlación clásica. Es por esto que el tamaño del kernel Gaussiano se tiene que elegir según la aplicación y se sugiere utilizar la regla de Silverman [Sil86] para la estimación de estadísticos y análisis de datos.

Propiedad 2

Sea $\{x_{\mathrm{i}}, \mathrm{i}\,\varepsilon\, T\}$ un proceso estocástico con T el conjunto de entrada, la función correntropía del proceso estocástico $V\left(\mathrm{i}, j\right) = E\left[k_\sigma\left(x_{\mathrm{i}} - x_j\right)\right]$ es una función simétrica y definida positiva.

Propiedad 3

Sea $f_{X,Y}\left(x, y\right)$ la función de la función de densidad de probabilidad conjunta de las muestras $\{x_{\mathrm{i}}, y_{\mathrm{i}}\}_{\mathrm{i}=1}^{\mathrm{d}}$ y $\hat{f}_{X,Y,\sigma}\left(x, y\right)$ su estimador de Parzen con tamaño de kernel σ. El estimador de la función de Correntropía con tamaño de kernel $\sigma' = \sqrt{2}\sigma$ es la integral de $\hat{f}_{X,Y,\sigma}\left(x, y\right)$ a lo largo de la recta $x = y$.

$$\hat{V}_{\sqrt{2}\sigma}\left(X, Y\right) = \int_{-\infty}^{+\infty} \hat{f}_{X,Y,\sigma}\left(x, y\right)|_{x=y=u}\, \mathrm{d}u$$

Propiedad 4

Dado dos vectores $X = [x_1, x_2, \ldots, x_N]$ e $Y = [y_1, y_2, \ldots, y_N]$, se define una métrica para el espacio de muestras como $CIM\left(X, Y\right) = \left(k_\sigma\left(0\right) - V\left(X, Y\right)\right)^{1/2}$ (CIM acrónimo de *Correntropy Induce Metric* - Métrica Inducida por Correntropía) , donde k_σ es el kernel Gaussiano de tamaño σ .
Cuando el tamaño del kernel es muy grande, CIM tiende a la norma 0 y si el tamaño de kernel tiende a 0, la métrica tiende a la norma 2 o euclidiana.

$$norma\,0\left(X, Y\right) = \begin{cases} 0 & si\ \|X - Y\| = 0 \\ 1 & si\ \|X - Y\| \neq 0 \end{cases}$$

$$norma\,2\left(X, Y\right) = \|X - Y\|^2$$

Propiedad 5

Dado un conjunto de datos $\{x_{\mathrm{i}}\}_{\mathrm{i}=1}^{\mathrm{d}}$, la función Correntropía crea otro conjunto de datos $\left\{f\left(x_{\mathrm{i}}\right)_{\mathrm{i}=1}^{\mathrm{d}}\right\}$, que preserva las medidas de semejanza dadas por la métrica inducida por Correntropía.

$$V\left(X, Y\right) = E\left[k_\sigma\left(x_{\mathrm{i}} - x_j\right)\right] = E\left[f\left(x_{\mathrm{i}}\right) f\left(x_j\right)\right] \tag{2.10}$$

2.5. Filtro C-MACE

El diseño del filtro C-MACE usado en el presente trabajo, es el diseño propuesto por Jeong en [JLH+09, JP08].

Notemos el i-ésimo vector de imagen como $x_{\mathrm{i}} = [x_{\mathrm{i}}(1), x_{\mathrm{i}}(2), \ldots, x_{\mathrm{i}}(\mathrm{d})]^T$ y al filtro $h = [h(1), h(2), \ldots, h(\mathrm{d})]^T$, en donde T significa la traspuesta.

Denotaremos a la matriz de imágenes de entrenamiento y al vector del filtro de dimensiones $\mathrm{d} \times N$ y $\mathrm{d} \times 1$ respectivamente, luego de aplicarles correntropía como

$$Fx = [\mathtt{f}_{x1}, \mathtt{f}_{\mathtt{x2}}, \mathtt{f}_{\mathtt{x3}}, \ldots, \mathtt{f}_{\mathtt{xN}}] \tag{2.11}$$

$$\mathtt{f}_h = [f(h(1)), f(h(2)), f(h(3)), \ldots, f(h(\mathrm{d}))]^T \tag{2.12}$$

en donde

$$\mathtt{f}_{x\mathrm{i}} = [f(x_{\mathrm{i}}(1)), f(x_{\mathrm{i}}(2)), f(x_{\mathrm{i}}(3)), \ldots, f(x_{\mathrm{i}}(\mathrm{d}))]^T \tag{2.13}$$

para $\mathrm{i} = 1 \ldots N$.

Basados en la ecuación 2.10, podemos estimar la correntropía cruzada entre la i-ésima imagen de entrenamiento y el filtro como:

$$v_{o\mathrm{i}}[m] = \frac{1}{\mathrm{d}} \sum_{n=1}^{\mathrm{d}} f(h(n)) f(x_{\mathrm{i}}(n-m)) \tag{2.14}$$

donde $m = \mathrm{d}-1, \mathrm{d}-2, \ldots, 0$ es la distancia entre dos píxeles de la i-ésima imagen. Entonces se obtiene una expresión para el vector de la correntropía cruzada para todas las m-distancias posibles entre píxeles $v_{o\mathrm{i}}[m]$ como:

$$v_{o\mathrm{i}} = S_{\mathrm{i}} \mathtt{f}_h \tag{2.15}$$

En donde S_{i} es una matriz de dimensión $(2\mathrm{d}-1) \times \mathrm{d}$

$$S_{\mathrm{i}} = \begin{pmatrix} f\left(x_{\mathrm{i}}\left(\mathrm{d}\right)\right) & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ f\left(x_{\mathrm{i}}\left(\mathrm{d}-1\right)\right) & f\left(x_{\mathrm{i}}\left(\mathrm{d}\right)\right) & 0 & \cdots & \cdots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ f\left(x_{\mathrm{i}}\left(1\right)\right) & f\left(x_{\mathrm{i}}\left(2\right)\right) & \cdots & \cdots & \cdots & \cdots & f\left(x_{\mathrm{i}}\left(\mathrm{d}\right)\right) \\ 0 & f\left(x_{\mathrm{i}}\left(1\right)\right) & \cdots & \cdots & \cdots & \cdots & f\left(x_{\mathrm{i}}\left(\mathrm{d}-1\right)\right) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & 0 & \cdots & 0 & f\left(x_{\mathrm{i}}\left(1\right)\right) \end{pmatrix} \tag{2.16}$$

De esta forma la energía de correntropía de la i-ésima imagen está dada por:

$$E_{\mathrm{i}} = v_{o\mathrm{i}}^{T} v_{o\mathrm{i}} = \mathtt{f}_{h}^{T} S_{\mathrm{i}}^{T} S_{\mathrm{i}} \mathtt{f}_{h} \tag{2.17}$$

Reemplazando $S_{\mathrm{i}}^{T} S_{\mathrm{i}} = V_{x\mathrm{i}}$, se obtiene la matriz de correntropía de dimensión $\mathrm{d} \times \mathrm{d}$ como:

$$V_{x\mathrm{i}} = \begin{pmatrix} v_{x\mathrm{i}}\left(0\right) & v_{x\mathrm{i}}\left(1\right) & \cdots & v_{x\mathrm{i}}\left(\mathrm{d}-1\right) \\ v_{x\mathrm{i}}\left(1\right) & v_{x\mathrm{i}}\left(0\right) & \cdots & v_{x\mathrm{i}}\left(\mathrm{d}-2\right) \\ \vdots & \vdots & \ddots & \vdots \\ v_{x\mathrm{i}}\left(\mathrm{d}-1\right) & v_{x\mathrm{i}}\left(\mathrm{d}-2\right) & \cdots & v_{x\mathrm{i}}\left(0\right) \end{pmatrix} \tag{2.18}$$

en donde cada elemento de la matriz está dado por:

$$v_{x\mathrm{i}}\left(l\right) = \sum_{n=1}^{\mathrm{d}} k_{\sigma}\left(x_{\mathrm{i}}\left(n\right) - x_{\mathrm{i}}\left(n+l\right)\right) \tag{2.19}$$

para $l = 1, \ldots, \mathrm{d}-1$

Luego, la energía promedio de correntropía para todo el conjunto de imágenes de entrenamiento se puede escribir como:

$$E_{prom} = \frac{1}{N} \sum E_{\mathrm{i}} = \mathtt{f}_{h}^{T} V_{X} \mathtt{f}_{h} \tag{2.20}$$

en donde

$$V_{x} = \frac{1}{N} \sum_{\mathrm{i}=1}^{n} v_{x\mathrm{i}} \tag{2.21}$$

El objetivo es minimizar la energía de correntropía (ecuación 2.20). Este problema de optimización se puede formular como:

$$min\ \mathtt{f}_h^T V_X \mathtt{f}_h \tag{2.22}$$

$$s.a.\ F_x^T \mathtt{f}_h = c \tag{2.23}$$

Como la matriz de correntropía V_x es definida positiva, entonces existe una solución analítica a este problema dada por:

$$\mathtt{f}_h = V_x^{-1} F_x \left(F_x^T V_x^{-1} F_x\right)^{-1} c \tag{2.24}$$

Sea Z la matriz de prueba de P imágenes, La salida del filtro es un vector de dimensión $P \times 1$ que está dado por:

$$y = F_z^T V_x^{-1} F_x \left(F_x^T V_x^{-1} F_x\right)^{-1} c \tag{2.25}$$

Para simplificar la ecuación 2.25 se introduce la siguiente notación,

$$T_{zx} = F_z^T V_x^{-1} F_x \tag{2.26}$$

$$T_{xx} = F_x^T V_x^{-1} F_x \tag{2.27}$$

En donde T_{zx} y T_{xx} son dos matrices con dimensión $P \times N$ y $N \times N$ respectivamente, y cada uno de sus elementos están dados por:

$$T_{zx}\left(\mathrm{i}, j\right) = \sum_{k=1}^{P} \sum_{l=1}^{N} w_{lk} k_\sigma \left(z_\mathrm{i}\left(k\right) - x_j\left(l\right)\right) \tag{2.28}$$

$$T_{xx}\left(\mathrm{i}, j\right) = \sum_{K=1}^{N} \sum_{l=1}^{N} w_{lk} k_\sigma \left(x_\mathrm{i}\left(k\right) - x_j\left(l\right)\right) \tag{2.29}$$

Con esta notación la salida del filtro C-MACE [JP08, JLH+09] para un conjunto de imágenes Z queda como:

$$\Longrightarrow y = T_{zx} \left(T_{xx}\right)^{-1} c \tag{2.30}$$

en donde w_{lk} es el elemento (l, k) de V_x^{-1}.

2.6. Reducción Dimensional con Proyección Aleatoria

El filtro C-MACE está hecho para trabajar en el dominio de los pixeles, en donde las imágenes poseen una dimensionalidad muy alta. La reducción aleatoria es un método para reducción dimensional y es utilizada en el presente trabajo para reducir los costos computacionales del filtro C-MACE en base a la reducción dimensional de los datos de entrada del filtro.

Sea un conjunto de tamaño N de datos y de dimensión d. La proyección de los datos en un sub-espacio de dimensión k ($k \lll \mathrm{d}$) usando una matriz aleatoria R de $k \times \mathrm{d}$ está dada por

$$X_{k\times N}^{RP} = R_{k\times \mathrm{d}} X_{\mathrm{d}\times N}$$

en donde $X_{k\times N}^{RP}$ es la proyección aleatoria de los N datos en el espacio de dimensión k.

El fundamento teórico de la proyección aleatoria o RP (*Random Projection*) nace del lema de Johnson-Lindenstrauss [BM01].

Lema Johnson-Lindenstrauss
Para cualquier $0 < \varepsilon < 1$ y cualquier entero N, sea k un entero positivo tal que

$$k \geq 4 \left(\frac{\varepsilon^2}{2} - \frac{\varepsilon^3}{3} \right)^{-1} \ln N$$

entonces para cualquier conjunto C de N puntos en R^{d}, hay una proyección $f : R^{\mathrm{d}} \to R^k$ tal que para todo $u, v \, \varepsilon \, C$,

$$(1-\varepsilon) \left\| u - v \right\|^2 \leq \left\| f(u) - f(v) \right\|^2 \leq (1+\varepsilon) \left\| u - v \right\|^2$$

En términos simples, este lema dice que si un vector es proyectado en un sub espacio aleatorio elegido adecuadamente, entonces las distancias entre los puntos originales aproximadamente se conservan.

Una matriz R generada por un conjunto de datos Gaussiano con media cero y varianza unitaria satisface el lema anterior, por lo que es el usado en la mayoria de los casos.

La ventaja principal de este método es su baja complejidad computacional: Construir la matriz R y proyectar los datos de $R^{\mathrm{d}} \to R^k$ es de orden $\mathcal{O}(\mathrm{d}kN)$, la cual es despreciable cuando se compara con método de Análisis de Componentes Principales de orden $\mathcal{O}(\mathrm{d}^2 N + \mathrm{d}^3)$. De esta forma la reducción aleatoria es un método recomendado para algoritmos que necesiten costos computacionales bajos.

2.7. Estimación de Silverman para σ en kernel Gaussiano

Para un conjunto de datos aleatorios $X = \{x_1, x_2, ..., x_{n-1}, x_n\}$, Silverman [Sil86] propone que el ancho de banda óptimo para el kernel Gaussiano está dado por:

$$\sigma_{Silverman} = \left(\frac{4\hat{\sigma}^5}{3n}\right)^{\frac{1}{5}} \simeq 1{,}06\hat{\sigma}n^{-\frac{1}{5}}$$

en donde $\hat{\sigma}$ es la desviación estandar del conjunto de datos aleatorios y n es el número de datos.

2.8. Computación Paralela

2.8.1. Velocidad de Procesamiento

La velocidad de procesamiento es la velocidad en que el *hardware* realiza una operación o procesar un dato.

Si se conoce el tiempo que se demora un proceso y el número de operaciones que se realizan en el mismo, la velocidad de procesamiento es

$$Velocidad\ de\ Procesamiento = \frac{número\ de\ Operaciones}{tiempo\ de\ procesamiento}$$

La unidad clásica de velocidad de procesamiento son los FLOPS (acrónimo de FLoating-point Operations Per Second - operaciones de punto de coma flotante por segundo). Hay distintas formas de estimar la velocidad de procesamiento y que dependen principalmente de la información que se pueda obtener de los procesos internos de cada dispisitivo. En el presente trabajo se calcularán los pixeles procesados por segundo como una medida de velocidad de procesamiento.

2.8.2. Aceleración (speed-up)

La aceleración de un proceso es la razón entre el tiempo que demora un procesador y el tiempo que demoran p procesadores. Sea T_1 el tiempo que demora 1 procesador en ejecutar un proceso y T_P el tiempo de ejecución del mismo proceso entre p procesadores, la aceleración del proceso S_p es:

$$S_P = \frac{T_1}{T_p}$$

Capítulo 3

Metodología

3.1. Características de GPU y CPU

En los experimentos de esta memoria se usan dispositivos cuyas características se muestran en las tabla 3.1 para la CPU y en la tabla 3.2 para la GPU. En ambos casos, los dispositivos son parte de un computador de escritorio. Esto quiere decir que en el caso de la CPU, además de ejecutar el Filtro CMACE, mantiene un porcentaje de cómputo para ejecutar el resto de los programas del ordenador y en el caso de la GPU, parte de su poder de cómputo es usado para el control y cálculo de gráficos en el computador.

Modelo	**Intel Core 2 Duo E7300**
Número de Núcleos	4
Frecuencia	2.67 [GHz]
Memoria RAM	2 [GB]

Tabla 3.1: Características de CPU.

Modelo	**Tesla C2050**
Clock del Núcleo	575 MHz
Número de Procesadores Threads	448
Clock de Procesadores Threads	1150 MHz
Acceso a Memoria Global	144 GB/s
Tipo de Bus Memoria Global	GDDR5
Tamaño de Memoria Global	3072 MB
Clock de Memoria Global	3000 MHz
Velocidad de Ejecución	1030 GFLOPS
Potencia de Diseño Térmico (disipación de calor de la GPU)	238 watts

Tabla 3.2: Características GPU utilizada.

3.2. Bases de Datos

Para las simulaciones y análisis se usa la base de expresiones faciales del Laboratorio Avanzado de Procesamiento Multimedial del Departamento de Ingeniería Eléctrica y Computación de la Universidad Carnegie Mellon (*Advanced Multimedia Processing Laboratory, Carnegie Mellon University*). Esta base consta de 6 sujetos con 74 imágenes diferentes de cada uno, con distintas expresiones faciales (444 imágenes en total). En la figura 3.1 se muestran ejemplos de rotros y distintas expresiones de la base de datos utilizada en los experimentos.

Figura 3.1: Rostros y expresiones faciales de la base de datos de imágenes usada en las simulaciones.

La tamaño de cada imagen es de 64×64 (dimensión d = 4096) pixeles y vienen en formato bmp en escala de grises (cada pixel está caracterizado por un entero entre 0 y 255, que es el valor de su representación de gris) .

Con el uso de esta base de datos de imágenes, se realizan experimentos de aceleración y validación de la implementación de CMACE en GPU.

Para los experimentos de aceleración se realizan pruebas variando de 1 a 200 el número de imágenes en la etapa de entrenamiento y con 1 imagen a clasificar.

Para las pruebas de validación se escoge una base de datos de imágenes para entrenamiento con 4 de los 6 sujetos incluidos en [amplaeceC] y a su vez se escogen 5 imágenes al azar de cada uno para la etapa de entrenamiento y luego se mide el poder de clasificación de cada filtro (MACE y CMACE) usando el resto de las base de datos [amplaeceC] como conjunto de test. Esto se realiza para cada experiencia y configuración de las pruebas de validación.

3.3. Diseño del filtro C-MACE en GPU

El filtro CMACE tiene un mejor poder de clasificación en relación al filtro MACE, pero sus grandes costos computacionales lo hacen una herramienta poco práctica. El diseño en

GPU pretende dar una solución a los tiempos de espera relacionados con la ejecución de este filtro.

El diseño de la implementación en GPU del filtro CMACE propuesto en esta memoria es enfocado a la clasificación de rostros y expresiones faciales en imágenes.

En el funcionamiento del filtro C-MACE se distinguen dos etapas:

- Entrenamiento: corresponde a la etapa de aprendizaje y construcción del filtro (construcción de las matrices ω_{lk} y T_{xx}) mediante el procesamiento de un conjunto de imágenes. Los resultados obtenidos de esta etapa se usan en la fase en línea sin tener que repetir los cálculos para cada clasificación.
- En línea: Etapa en la cual el filtro es capaz de clasificar y distinguir imágenes relacionadas con las imágenes usadas en la etapa de entrenamiento.

Invertir una matriz, al ser un problema con una paralelización baja, no es ventajoso programarlo directamente en la GPU por lo que, en la etapa de entrenamiento del filtro C-MACE, invertir las matrices se realizó directamente en la CPU. Esto deja como resultado la ejecución de forma híbrida en la etapa de entrenamiento mientras que la etapa en línea es completamente ejecutada en la GPU.

El diagrama de flujo final del funcionamiento del filtro C-MACE se muestra en la Figura 3.2.

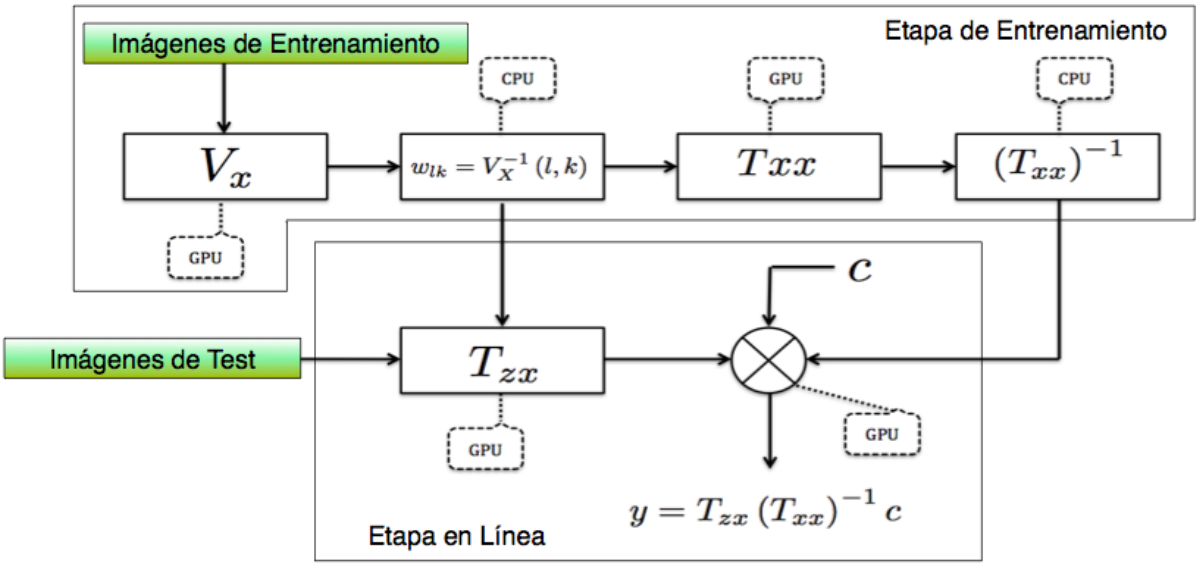


Figura 3.2: Diagrama de Flujo del Filtro C-MACE.

Para mejorar el desempeño del filtro, se programó directamente en la GPU una función para calcular la correntropía entre dos pixeles. Esto permite que cualquier programa Kernel usado en la GPU pueda llamar esta función para hacer los cálculos respectivos.

Esta función (ecuación 3.1) permanece en línea durante todo el funcionamiento del filtro y en ambas etapas (entrenamiento y en línea).

$$f(x,y) = \frac{1}{\sqrt{\pi}\sigma} \mathrm{e}^{\left(\frac{-\|x-y\|}{\sigma^2}\right)} \tag{3.1}$$

3.4. Descripción de los programas Kernel de GPU

Kernel es el nombre técnico que tienen los programas que son implementados en la GPU. A continuación se hace una descripción de los programas Kernel usados para la implementación del Filtro CMACE en GPU y se muestra además, sus limitaciones de uso en función del uso de memoria global de la tarjeta gráfica.

3.4.1. Cálculo de V_X

V_X es una matriz de dimensión d × d (ecuación 2.21), en donde se representa el promedio de la matriz de correntropía (ecuación 2.18) de todas las imágenes de la etapa de entrenamiento.

La paralelización de este cálculo se realiza generando una grilla de d×d hilos de cálculo, en donde cada uno tiene la responsabilidad de dar como resultado $k_\sigma(x_\mathrm{i}(n) - x_\mathrm{i}(n+l))$ de la ecuación 3.2, para un n y un l dados por la fila y columna que se encuentren dentro de la grilla.

$$v_{x\mathrm{i}}(l) = \sum_{n=1}^{\mathrm{d}} k_\sigma(x_\mathrm{i}(n) - x_\mathrm{i}(n+l)) \tag{3.2}$$

Luego se suman las columnas de la grilla, quedando un vector en donde cada elemento representa la suma de la correntropía entre píxeles de igual distancia entre ellos.

Después de esto, se crea una nueva grilla de d × d hilos de ejecución, en donde se rellena la matriz V_X con los elemenos del vector $v_{x\mathrm{i}}(l)$ según el algoritmo de decisión que se muestra a continuación,

$$V_X(k,j)_{\mathrm{d}\times\mathrm{d}} = \begin{cases} V_X(k,j) + \frac{v_{x\mathrm{i}}(k-j)}{N} & \text{si } k \geq j \\ V_X(k,j) + \frac{v_{x\mathrm{i}}(j-k)}{N} & \text{si } j > k \end{cases} \tag{3.3}$$

Finalmente se obtiene la matriz $V_x(k,j)$ tal como se muestra en la ecuación 3.4

$$V_x(k,j) = \frac{1}{N}\begin{pmatrix} \sum_{\mathrm{i}=1}^{N} v_{x\mathrm{i}}(0) & \sum_{\mathrm{i}=1}^{N} v_{x\mathrm{i}}(1) & \cdots & \sum_{\mathrm{i}=1}^{N} v_{x\mathrm{i}}(\mathrm{d}-1) \\ \sum_{\mathrm{i}=1}^{N} v_{x\mathrm{i}}(1) & \sum_{\mathrm{i}=1}^{N} v_{x\mathrm{i}}(0) & \cdots & \sum_{\mathrm{i}=1}^{N} v_{x\mathrm{i}}(\mathrm{d}-2) \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{\mathrm{i}=1}^{N} v_{x\mathrm{i}}(\mathrm{d}-1) & \sum_{\mathrm{i}=1}^{N} v_{x\mathrm{i}}(\mathrm{d}-2) & \cdots & \sum_{\mathrm{i}=1}^{N} v_{x\mathrm{i}}(0) \end{pmatrix}_{\mathrm{d}\times\mathrm{d}} \tag{3.4}$$

La figura 3.3 muestra un diagrama de flujo en donde se ilustra el funcionamiento del kernel V_x en GPU.

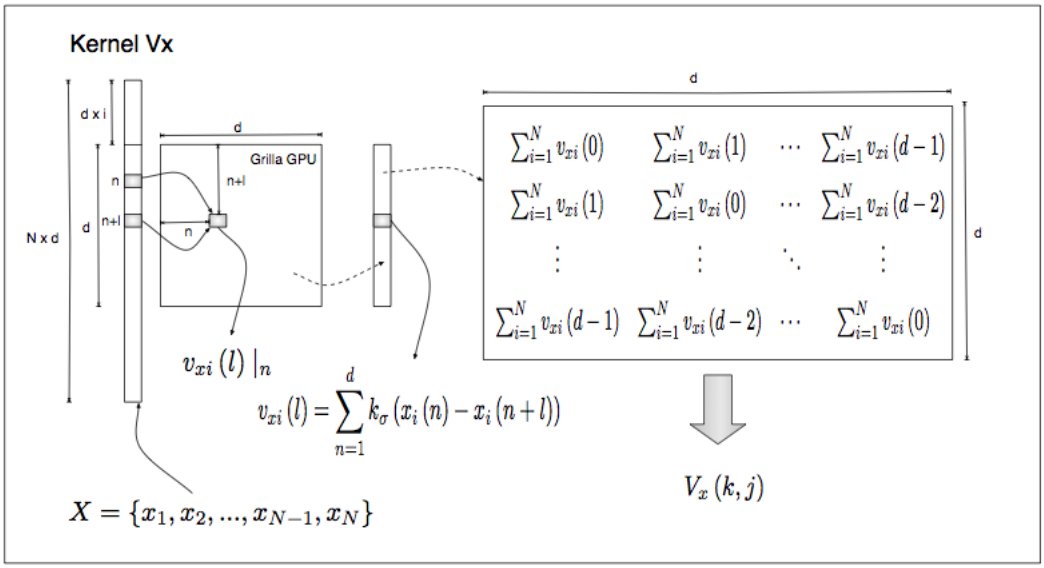


Figura 3.3: Diagrama de flujo del kernel Vx.

La complejidad del cálculo de V_x es $\mathcal{O}(N\mathrm{d}^2)$.

Al tener la matriz $V_x(k,j)$, se envían los valores de la matriz a la CPU para calcular su inversa y seguir con el resto del entrenamiento.

$$w_{lk} = V_X^{-1}(l,k)$$

3.4.2. Cálculo de T_{XX}

Se toma un par de imágenes (i,j) del conjunto de entrenamiento para calcular el término $T_{xx}(\mathrm{i},j)$ de la matriz T_{XX}.

$$T_{xx}(\mathrm{i},j) = \sum_{k=1}^{\mathrm{d}}\sum_{l=1}^{\mathrm{d}} w_{lk} k_\sigma(x_{\mathrm{i}}(k) - x_j(l)) \tag{3.5}$$

El cálculo de 3.5 se paraleliza haciendo que en una grilla de d × d, cada hilo de ejecución calcule el término $w_{lk}k_{\sigma}\left(x_{\mathrm{i}}\left(k\right)-x_{j}\left(l\right)\right)$ asignado dependiendo de la columna l y la fila k en que esté ubicado en la grilla.

Este proceso se repite N^2 veces, en donde N es el número de imágenes de entrenamiento.

Después de procesar todo el conjunto de entramiento se obtiene la matriz T_{XX}, como se muestra en la ecuación 3.6.

$$Txx = \begin{pmatrix} Txx\left(1,1\right) & \cdots & Txx\left(1,N\right) \\ Txx\left(2,1\right) & \cdots & Txx\left(2,N\right) \\ \vdots & \vdots & \vdots \\ Txx\left(N-1,1\right) & \cdots & Txx\left(N-1,N\right) \\ Txx\left(N,1\right) & \cdots & Txx\left(N,N\right) \end{pmatrix}_{N\times N} \tag{3.6}$$

La figura 3.4 muestra un diagrama de flujo en donde se ilustra el funcionamiento del Kernel T_{xx} en GPU.

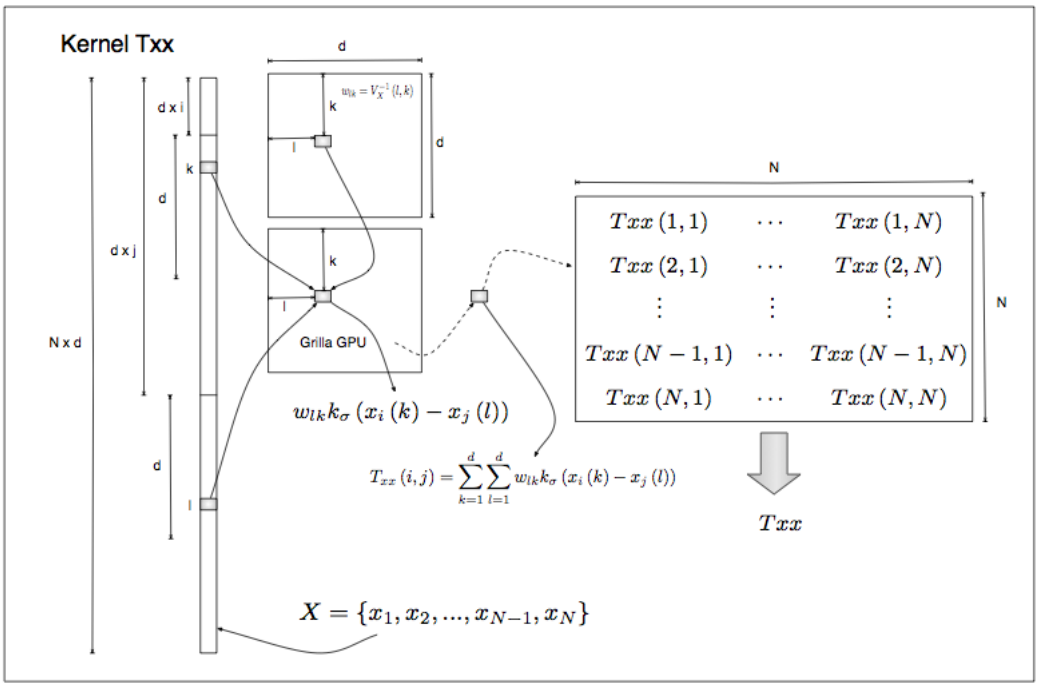


Figura 3.4: Diagrama de flujo del kernel Txx.

La complejidad del cálculo de T_{XX} es $\mathcal{O}\left(N^2\mathrm{d}^2\right)$.

La matriz T_{XX} se envía a la CPU para calcular su inversa. El cálculo de la inversa de T_{XX} da por finalizada la etapa de entrenamiento del filtro.

3.4.3. Cálculo de T_{ZX}

El cálculo de T_{ZX} es el procesamiento en línea de las imágenes de test. Para obtener esta matriz, se toma un par (i, j) de imágenes, en donde i es una imagen que pertenece al conjunto de test y j es una imagen del conjunto de entrenamiento. La ecuación 3.7 muestra la expresión para calcular el elemento (i, j) de la matriz T_{zx}.

$$T_{zx}(\mathrm{i}, j) = \sum_{k=1}^{\mathrm{d}} \sum_{l=1}^{\mathrm{d}} w_{lk} k_\sigma \left(z_{\mathrm{i}}(k) - x_j(l) \right) \tag{3.7}$$

La paralelización de $Tzx(\mathrm{i}, j)$ se realiza de manera similar que en el caso del cálculo de $Txx(\mathrm{i}, j)$, construyendo una grilla de $\mathrm{d} \times \mathrm{d}$ hilos de ejecución, para que cada hilo calcule un elemento $w_{lk} k_\sigma \left(z_{\mathrm{i}}(k) - x_j(l) \right)$ dependiendo de su fila k y su columna l dentro de la grilla.

Luego de esto se reduce la matriz $Tzx(\mathrm{i}, j)$ en ambas dimensiones y el escalar resultante es el valor del término (i, j) de la matriz T_{ZX} (la reducción es la misma para el cálculo Txx). Este proceso se repite ZN veces, en donde N es el número de imágenes de entrenamiento y Z es el número de imágenes de clasificación. Después de procesar todo el conjunto de clasificación se obtiene la matriz T_{ZX}, como muestra la ecuación 3.8.

$$Tzx = \begin{pmatrix} Tzx(1,1) & \cdots & Tzx(1,N) \\ Tzx(2,1) & \cdots & Tzx(2,N) \\ \vdots & \vdots & \vdots \\ Tzx(Z-1,1) & \cdots & Tzx(Z-1,N) \\ Tzx(Z,1) & \cdots & Tzx(Z,N) \end{pmatrix}_{Z \times N} \tag{3.8}$$

La complejidad del cálculo de T_{ZX} es $\mathcal{O}(ZN\mathrm{d}^2)$.

Luego de calcular la matriz T_{ZX}, se multiplica con el valor de T_{XX} calculado en la etapa de entrenamiento y se obtiene el resultado de CMACE para el conjunto de imágenes de test.

La figura 3.5 muestra el diagrama de flujos del Kernel T_{zx} para ilustrar su funcionamiento.

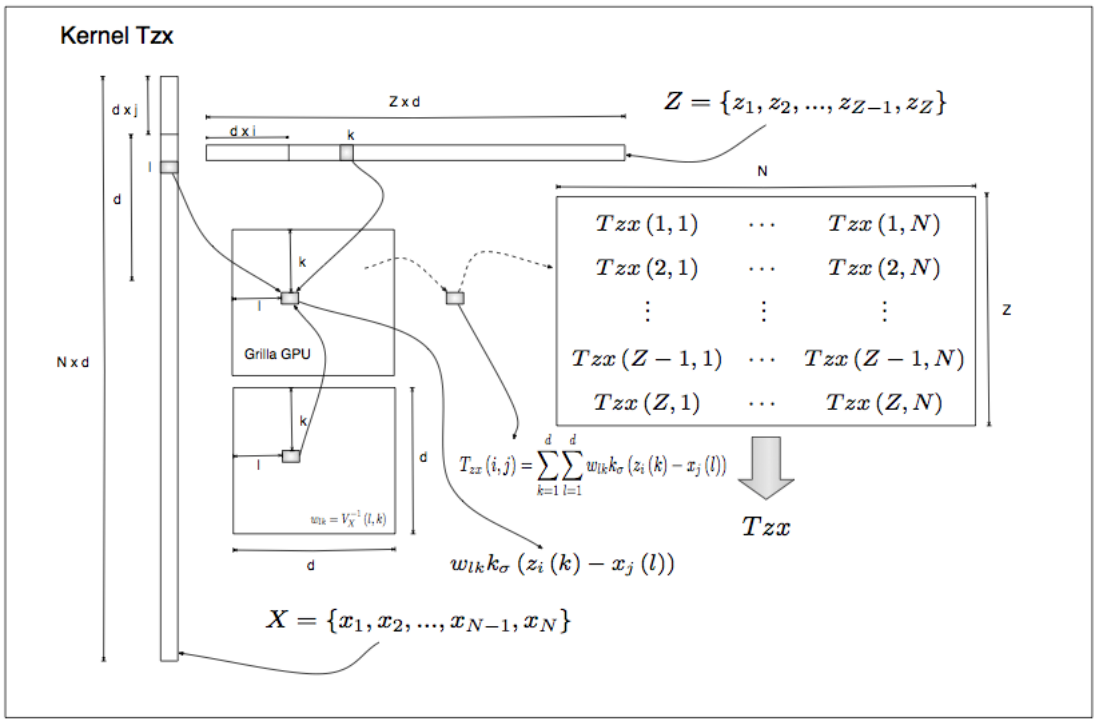


Figura 3.5: Diagrama de flujo del kernel Tzx.

3.4.4. Limitación de los Kernels en el Uso de Memoria Global de la GPU

La tabla 3.3 muestra el uso de memoria Global de la GPU para cada Kernel. La memoria Global es de tipo RAM y es de uso exclusivo de la GPU.

Kernel	*Tamaño*
V_x	$(N\mathrm{d} + 3\mathrm{d}^2)$
T_{xx}	$(N^2 + N\mathrm{d} + 3\mathrm{d}^2)$
T_{zx}	$(N\mathrm{d} + Z\mathrm{d} + 3\mathrm{d}^2 + ZN)$

Tabla 3.3: Memoria global de la GPU usada en cada kernel.

Según la tabla 3.2, la GPU Tesla C2050 cuenta con 3 GigaBytes de memoria Global. Tomando en cuenta la dimensión de las imágenes (d = 4096 en este caso), podemos calcular el número máximo de imágenes que la GPU podría procesar de una sola vez sin tener transferencia de datos entre CPU y GPU.

Para el cálculo se toma una imágen a clasificar ($Z = 1$) y cada imagen se considera de dimensión 4096 Pixeles.

Para el caso de V_x, el número máximo de imágenes de entrenamiento es dado por la ecuación 3.9.

$$N_{max} = \left(\frac{\frac{MEM_{GLOBAL}}{4} - 3\mathrm{d}^2}{\mathrm{d}} \right) \tag{3.9}$$

El factor 4 por el cual se dividide la memoria global está relacionado con el tamaño en Bytes del tipo de dato punto flotante, con el cual se realiza el cálculo. Reemplazando los datos en 3.9 se obtiene que el número máximo en esta etapa es de 170000 imágenes.

Para la etapa T_{XX}, el número máximo de imágenes está dado por la ecuación 3.10. Reemplazando los datos en del problema y las caraterísticas de la GPU en 3.10, el número máximo en esta etapa es de 9000 imágenes de entrenamiento.

$$N_{max} = \left(\frac{-\mathrm{d} + \sqrt{\frac{MEM_{GLOBAL}}{4} - 11\mathrm{d}^2}}{2} \right) \tag{3.10}$$

Para la etapa T_{ZX}, el número máximo de imágenes de entrenamiento está dado por la ecuación 3.11. Reemplazando los datos en 3.11 se obtiene que el número máximo de imágenes de entrenamiento en esta etapa es de 170000.

$$N_{max} = \left(\frac{\frac{MEM_{GLOBAL}}{4} - (\mathrm{d} + 3\mathrm{d}^2)}{\mathrm{d} - 1} \right) \tag{3.11}$$

Por lo tanto, el número máximo que el Filtro CMACE en GPU puede procesar a la vez es el mínimo de los máximos en cada etapa, es decir 9000 imágenes. Este valor depende directamente de la memoria global disponible en la GPU y de la dimensionalidad que tengan las imágenes a procesar.

Este es un cálculo teórico que puede diferir un poco de los valores reales debido a los usos de memoria en registros y variables auxiliares usadas en el cómputo de cada Kernel.

3.5. Pruebas de Validación

Las pruebas de validación consisten en la construcción de curvas ROC (acrónimo de *Receiver Operating Characteristic*, o Característica Operativa del Receptor) en distintas

configuraciones, para comprobar el poder de clasificación y las diferencias que hay con el método tradicional (Filtro MACE).

Para esto, se toma un conjunto de imágenes de entrenamiento y dos conjuntos de prueba, uno correspondiente a la base de casos positivos y otra correspondiente a una base de casos negativos. Para cada conjunto de entrenamiento se hace una prueba de clasificación con los dos conjuntos de prueba, variando el umbral de clasificación del filtro y midiendo la tasa de aciertos que tiene cada vez que se varía este parámetro. La tabla 3.4 muestra como se agrupan los distintos resultados posibles en este experimento.

Tipo de Acierto	Entrada al Filtro	Salida del Filtro (clasificación)
Verdadero Positivo (VP)	Positivo	Positivo
Verdadero Negativo (VN)	Negativo	Negativo
Falso Positivo (FP)	Negativo	Positivo
Falso Negativo (FN)	Positivo	Negativo

Tabla 3.4: Tipos de Acierto Para la Construcción de Curvas ROC (matriz de confusión).

Respecto a los tipos de acierto (tabla 3.4) se definen tres indicadores.

La curva ROC se obtiene graficando para cada umbral de corte, la Sensibilidad en ordenadas y la Fracción de Falsos Positivos (FFP) en abscisas.

$$Sensibilidad = \frac{VP}{VP + FN} \tag{3.12}$$

$$Especificidad = \frac{VN}{VN + FP} \tag{3.13}$$

$$FFP = 1 - Especificidad \tag{3.14}$$

Los resultados de las pruebas son el promedio de 10 realizaciones por cada configuración y con bases de prueba sin ruido y con ruido blanco como las imágenes que se muestran en la figura 3.6.

Figura 3.6: De izquierda a derecha: Imagen original, imagen con ruido blanco de intensidad 10dB y una imagen con ruido blanco de intensidad 20dB.

Capítulo 4

Resultados de Simulaciones

4.1. Paralelización

La aceleración del algoritmo es la principal preocupación que se tiene cuando se usa supercomputación en ingeniería. A continuación se muestran los resultados de los costos computacionales y medidaes de desempeño obtenidos en las implementaciones del filtro CMACE en CPU y GPU.

4.1.1. Perfil de Desempeño del Filtro CMACE en CPU

La figura 4.1 muestra el tiempo de ejecución del filtro CMACE en CPU tomando en cuenta todas las etapas de construcción y de clasificación en línea que hay involucradas.

Es importante notar como aumenta el tiempo a medida que crece la base de entrenamiento, llegando por sobre los 200 minutos cuando tenemos 200 imágenes.

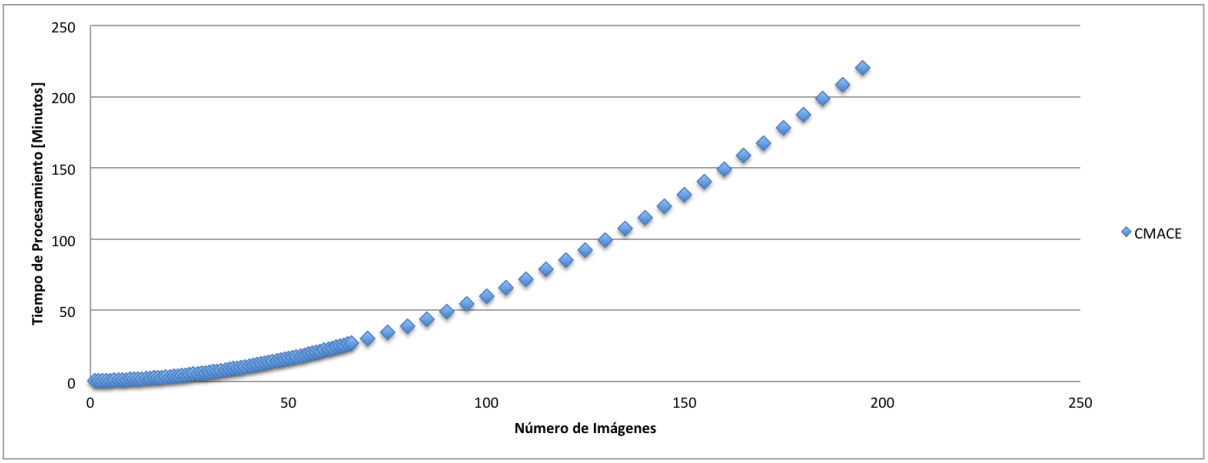


Figura 4.1: Tiempo Total de Ejecución CMACE en CPU.

A continuación se hace una división del algoritmo para analizar el comportamiento de cada sección a paralelizar y su contribución al tiempo total del filtro.

La figura 4.2 muestra que el comportamiento de la etapa de construcción de V_x es lineal en función del número de imágenes, lo cual concuerda con la complejidad estimada para esta sección. El tiempo de la construcción de V_x es superior a los 3 minutos en el límite de 200 imágenes que se toma para el desarrollo de estas pruebas. Al crecer la base de datos el tiempo de construcción de V_x es poco relevante en relación al tiempo total que toma la ejecución del filtro.

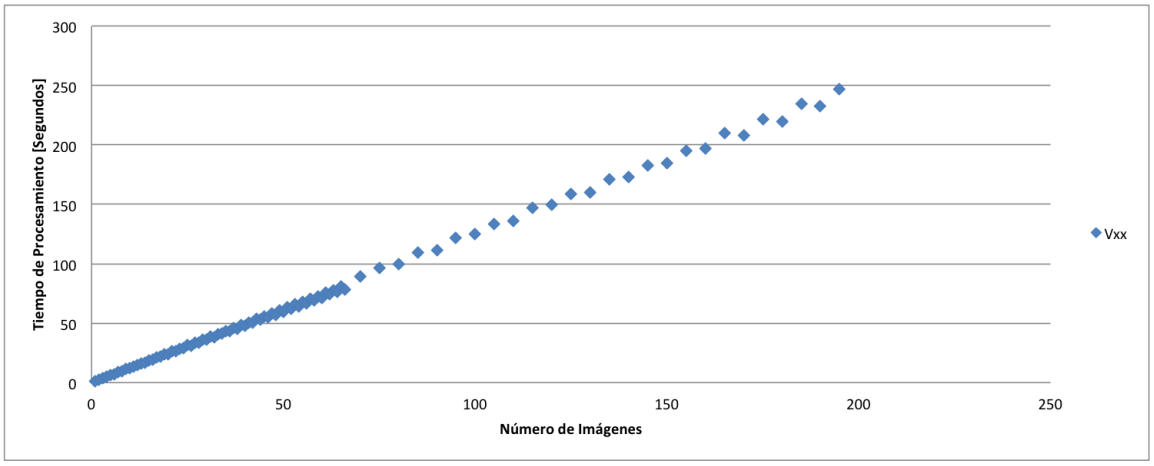


Figura 4.2: Tiempo de Ejecución [minutos] de Vx en CPU en Función del Número de Imágenes de Entrenamiento.

La figura 4.3 muestra el comportamiento de la etapa de construcción de T_{xx} en función del número de imágenes. Esta es sin duda la etapa mas costosa del algoritmo, siendo la contribución más grande al tiempo total del filtro.

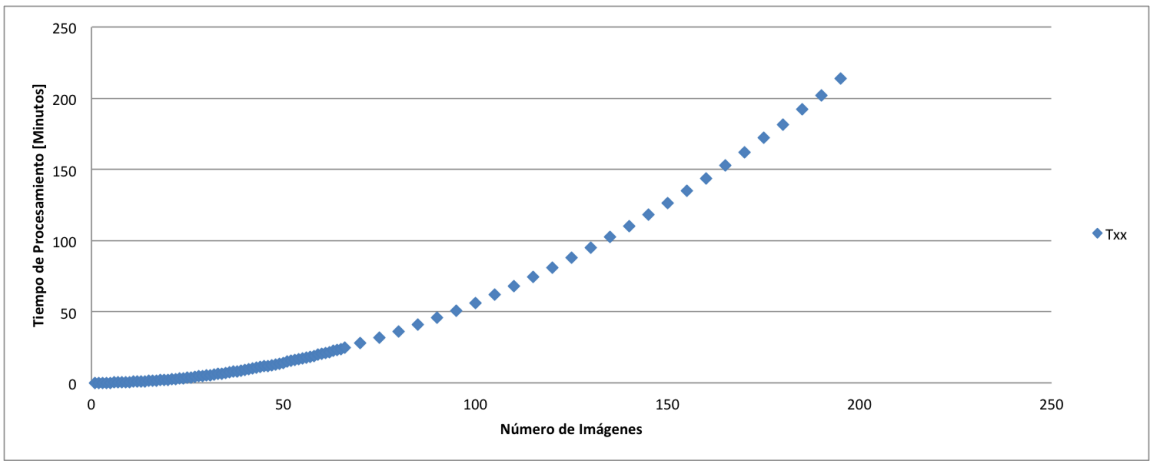


Figura 4.3: Tiempo de Ejecución [minutos] de Txx en CPU en Función del Número de Imágenes de Entrenamiento.

La figura 4.4 muestra el costo de tiempo de la construcción de T_{xx} y el tiempo total del filtro CMACE, con lo que se puede ver la similitud de ambas curvas, y además T_{xx} es la etapa principal y más compleja computacionalmente que tiene el filtro en relación a al tiempo.

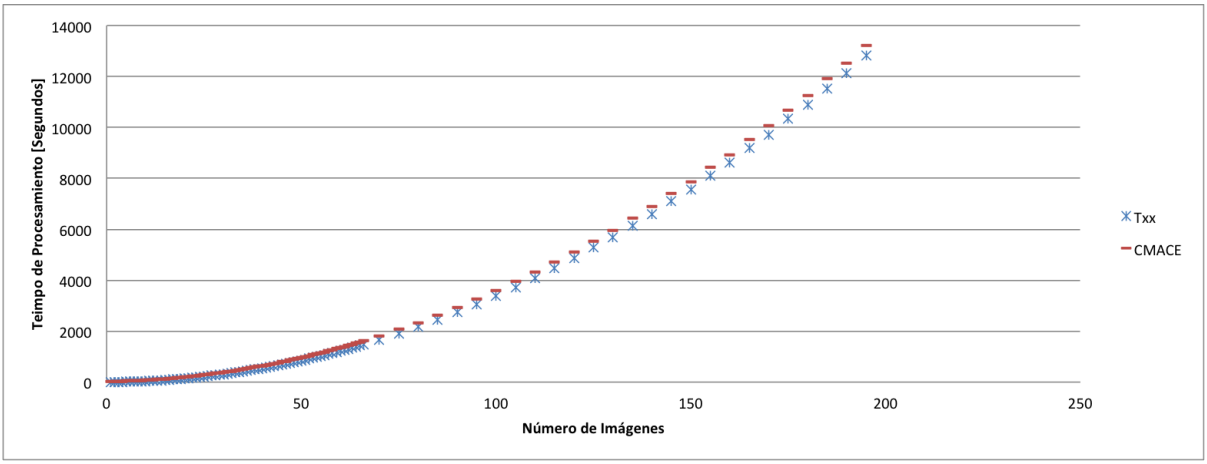


Figura 4.4: Comparación entre los tiempos de CMACE y Txx en CPU.

La figura 4.5 muestra los costos de construcción de T_{zx}. En relación a las etapas anteriores, esta etapa es la menos costosa computacionalmente, pero la más importante de paralelizar ya T_{zx} es calculada en línea, a diferencia de V_x y T_{xx} que son calculadas en la etapa de entrenamiento del filtro CMACE. La construcción de T_{zx} se hace en la etapa de clasificación de CMACE, por lo que es importante que tenga tiempos reducidos de ejecución para poder usar este algoritmo en problemas de clasificación en tiempo real.

El gráfico de la figura 4.5 muestra que a las 200 imágenes T_{zx} tiene costos superiores a los 2 minutos. Esta limitación reduce el campo de utilidad del filtro CMACE y es la principal preocupación por lo que se quiere reducir los tiempos de cómputo. T_{zx} es una matriz única para cada conjunto de test, es decir, cada vez que C-MACE clasifica una imagen de test, es necesario construir la matriz T_{zx} para esa imagen de test en particular. Practicamente todo el tiempo de clasificación (o etapa en línea) del filtro C-MACE es destinado a la construcción de la matriz T_{zx}. Es por este motivo que la reducción del costo computacional relacionado con la construcción de T_{zx}, puede hacer del filtro C-MACE una opción mucho más práctica al momento de tener que usar un filtro para el reconocimiento o clasificación de objetos en imágenes.

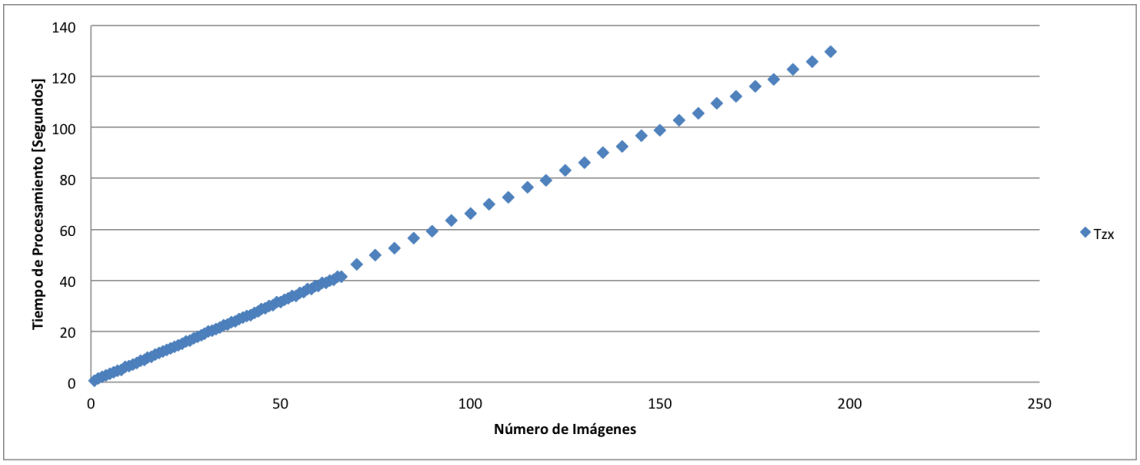


Figura 4.5: Tiempo de Ejecución de Tzx en CPU en Función de las Imágenes de Entrenamiento y con una Imagen de Test.

4.1.2. Tiempos de Ejecución en GPU y Aceleración Vx

La figura 4.6 muestra el tiempo de construcción de V_x en función del número de imágenes luego de paralelizar esta etapa en GPU. El comportamiento sigue siendo lineal, solo que a diferencia de V_x sin paralelizar (figura 4.2), el tiempo máximo de ejecución no supera los 10 segundos para 200 imágenes.

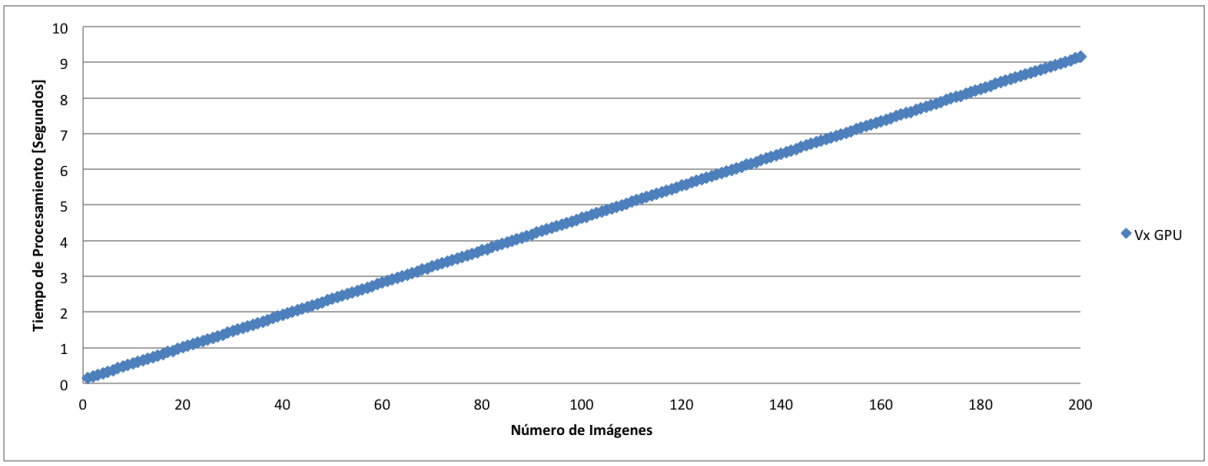


Figura 4.6: Tiempo de Ejecución [segundos] de Vx-GPU en Función del Número de Imágenes de Entrenamiento.

La figura 4.7 muestra la aceleración que se logra al paralelizar la etapa V_x en la GPU.

Para más de 20 imágenes de entrenamiento, la aceleración de V_x tiende a estabilizarse alrededor de 26, es decir V_x en GPU es 26 veces más rápido que el algoritmo implementado en una CPU tradicional.

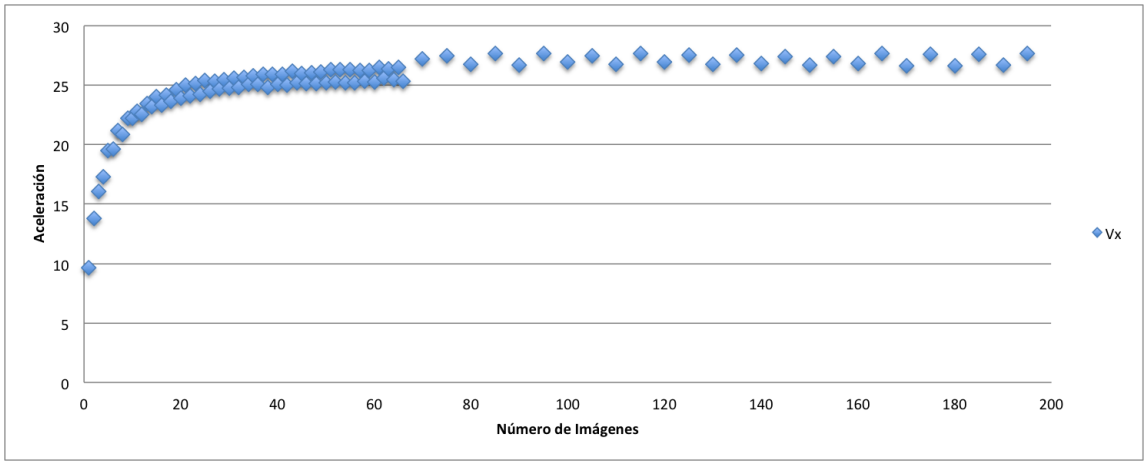


Figura 4.7: Aceleración de Vx en GPU en Función del Número de Imágenes de Entrenamiento.

4.1.3. Tiempos de Ejecución GPU y Aceleración Txx

La figura 4.8 muestra el tiempo de construcción de T_{xx} luego de paralelizar esta etapa en la GPU. El gráfico muestra un comportamiento cuadrático en función de número de imágenes en la base de entrenamiento, al igual que el algoritmo sin paralelizar (figura 4.3), pero los tiempos se reducen llegando a ser menos de 15 minutos en el caso más grande (considerando 200 imágenes de entrenamiento). La paralelización mediante GPU reduce los costos computacionales añadiendo procesadores en las etapas de cómputo, pero no modifica la complejidad de los algoritmos.

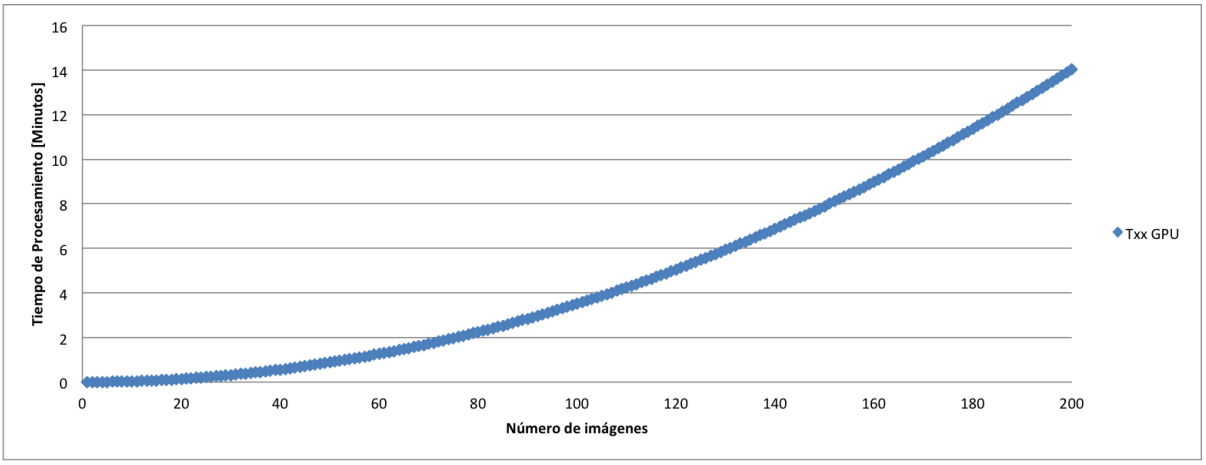


Figura 4.8: Tiempo de Ejecución [minutos] de Txx-GPU en Función del Número de Imágenes de Entrenamiento.

La aceleración de T_{xx} se muestra en la figura 4.9, de la que se deduce que T_{xx} en GPU es 16 veces más rápido que la implementación en CPU.

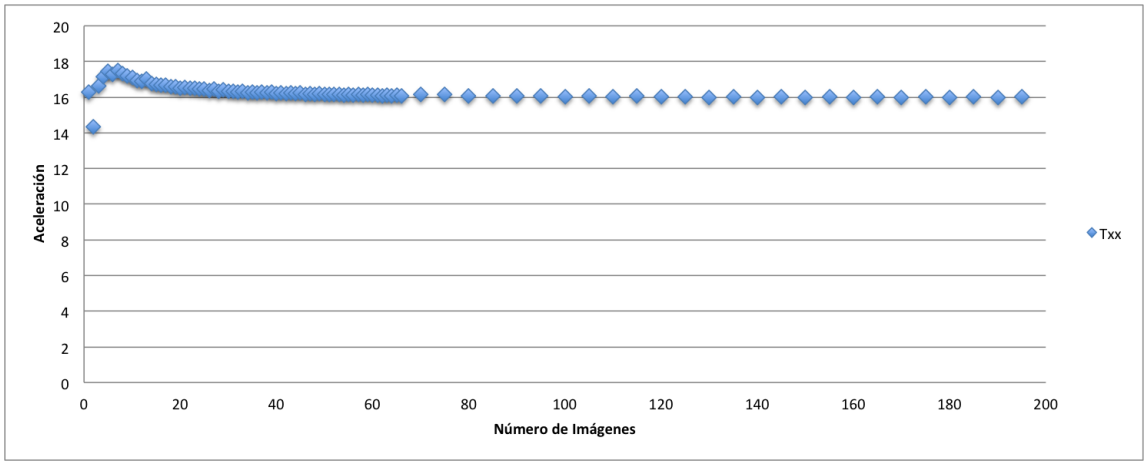


Figura 4.9: Aceleración de Txx en GPU en Función del Número de Imágenes de Entrenamiento.

4.1.4. Tiempos de Ejecución GPU y Aceleración Tzx

La figura 4.10 muestra el tiempo que toma la construcción de T_{zx} paralelizado por la GPU en función del número de imágenes de entrenamiento. El gráfico muestra un comportamiento lineal, al igual que el mostrado en la figura 4.5, con la diferencia de que el tiempo máximo (con 200 imágenes de entrenamiento) es de menos de 4 segundos. Los bajos tiempos obtenidos en esta etapa luego de paralelizarla en la GPU abren la opción de usar el filtro en problemas de clasificación en tiempo real y aumentan así sus posibles usos.

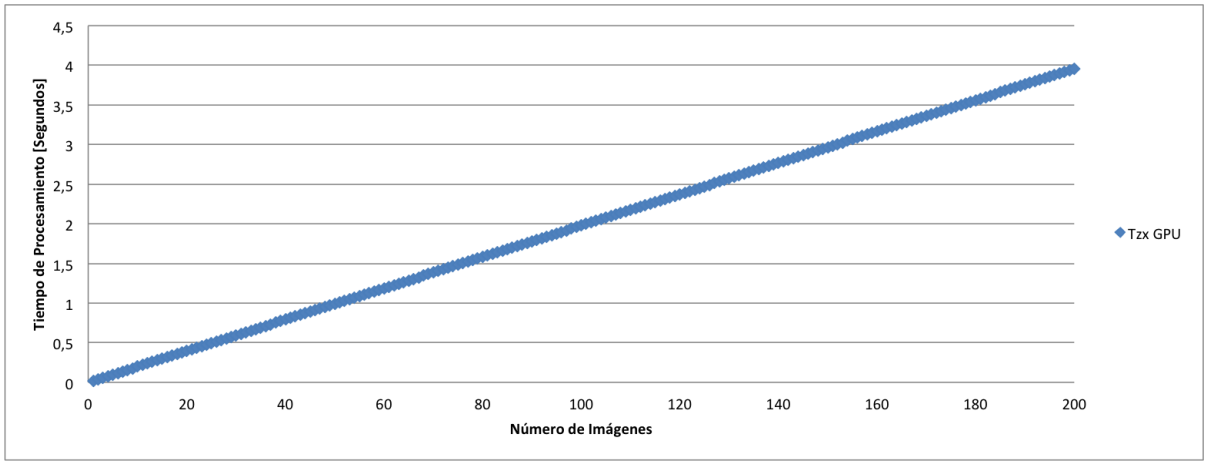


Figura 4.10: Tiempo de Ejecución [minutos] de Tzx-GPU en Función del Número de Imágenes de Entrenamiento y con una Imagen de test.

La aceleración de T_{zx} obtenida por su paralelización en GPU se muestra en la figura 4.11. El gráfico muestra que en esta etapa se logra que T_{zx} en GPU sea alrededor de 33 veces más rápido que la implementación tradicional en CPU.

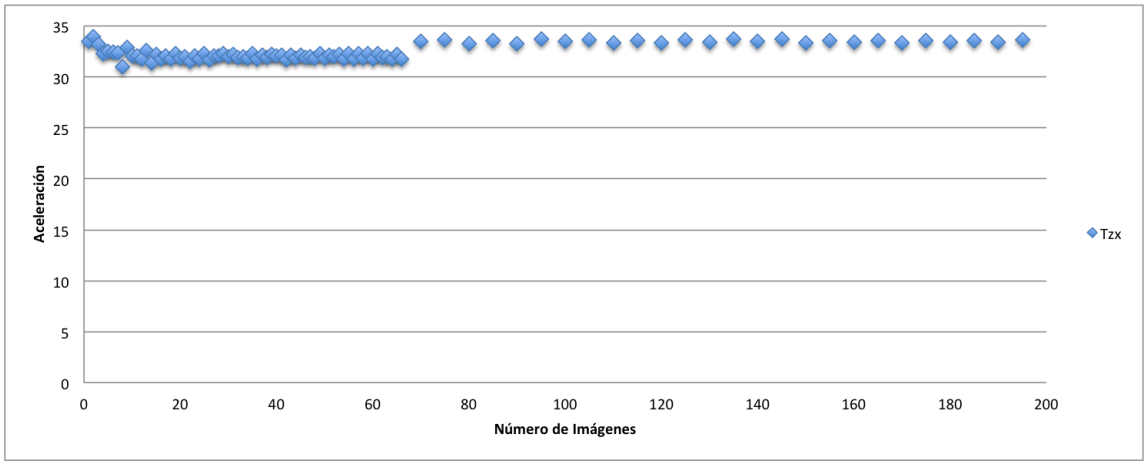


Figura 4.11: Aceleración de Tzx en Función de las Imágenes de Entrenamiento y con una Imagen de Prueba.

4.1.5. Tiempos de Ejecución GPU y Aceleración Total de CMACE

La figura 4.12 muestra el tiempo total de construcción y clasificación del filtro CMACE luego de paralelizar las etapas V_x, T_{xx}, T_{zx} y reemplazarlas en el algoritmo original.

Para el caso de 200 imágenes de entrenamiento (caso máximo en este estudio) el tiempo total se reduce de 220 minutos en CPU a poco menos de 14 minutos en GPU luego de la paralelización.

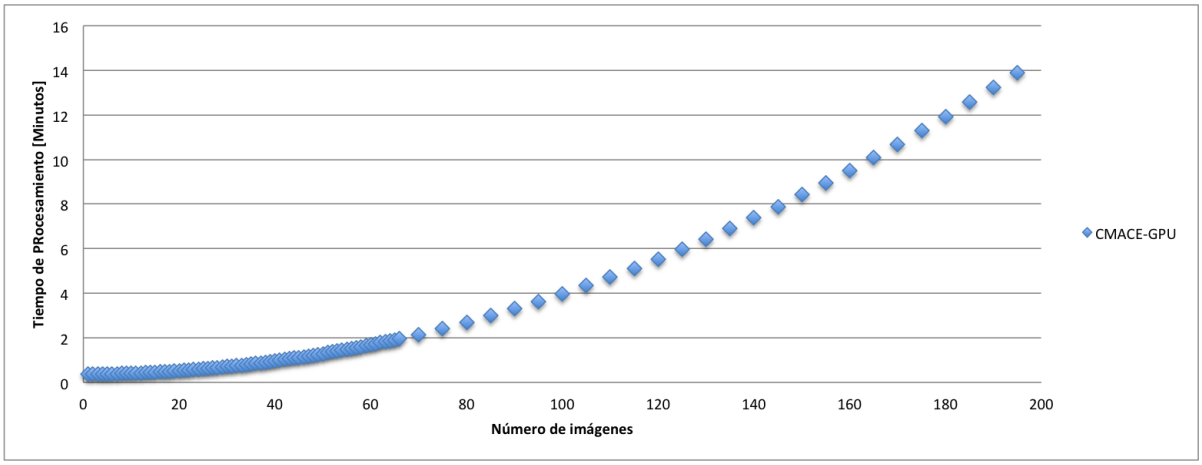
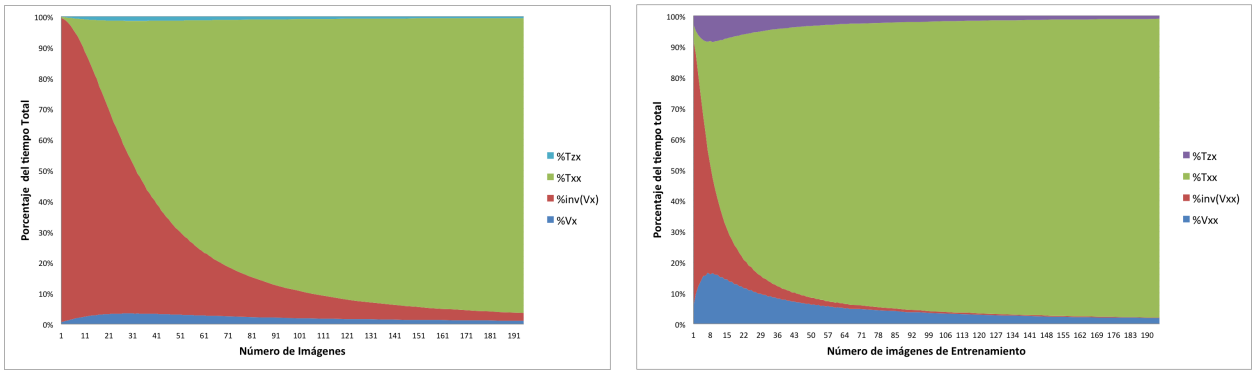


Figura 4.12: Tiempo total [minutos] CMACE en GPU.

La figura 4.13a muestra la contribución porcentual de cada etapa luego de la paralelización y la figura 4.13b muestra la contribución de cada etapa en la implementación en serie. La figura 4.13a muestra que la contribución de las etapas V_x y T_{zx} son practicamente

despreciables en cualquier configuración, mientras que T_{xx} se reduce y es comparable a la etapa de invertir la matriz V_x durante más imágenes de entrenamiento respecto a la etapa T_{xx} en la implementación en serie (figura 4.13b).



a *b*

Figura 4.13: Perfiles de implemntación en serie (b) y paralela (a) del filtro CMACE

La aceleración total del algoritmo se muestra en la figura 4.14. Al ser T_{xx} la etapa más costosa, la aceleración total se estabiliza en el mismo valor obtenido en la aceleración de esta etapa, es decir el filtro CMACE en GPU es 16 veces más rápido que en una implementación tradicional en CPU.

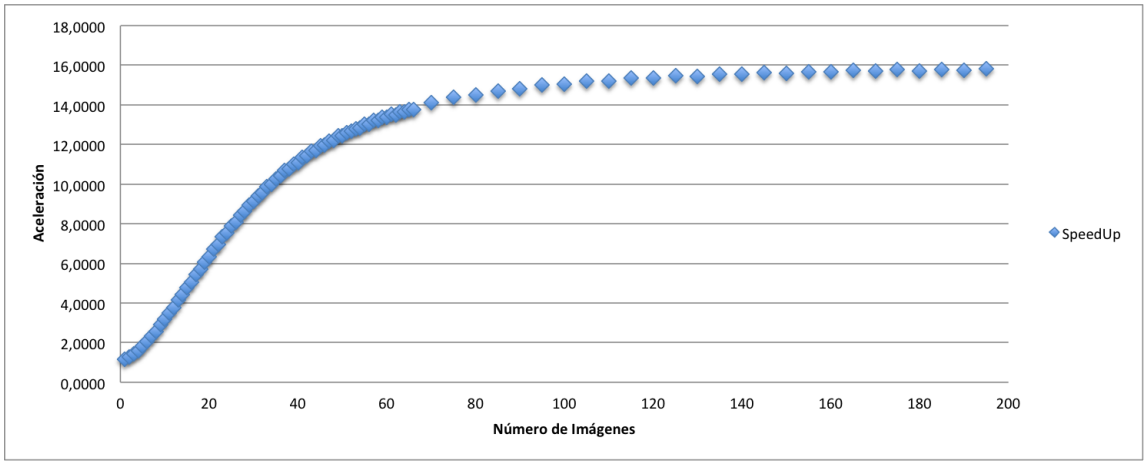


Figura 4.14: Aceleración Total Filtro CMACE en GPU vs CPU.

4.1.6. Tiempos de Ejecución y Aceleración de CMACE en GPU Usando Proyección Aleatoria

La figura 4.15 muestra el tiempo de ejecución que tiene el filtro CMACE, luego reducir la dimensionalidad de las imágenes, de 4096 a 256, mediante el proceso de Proyección Aleatoria.

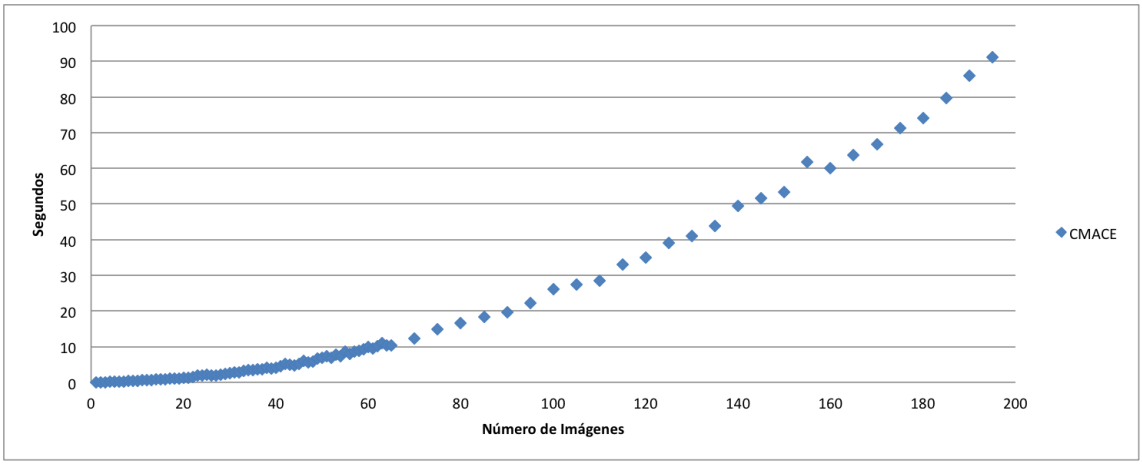


Figura 4.15: Tiempo Total de Ejecución CMACE con d=256.

La aceleración que logra la etapa de construcción de V_x con una dimensión de 256 pixeles es la que se muestra en la figura 4.16. El gráfico muestra que para esta dimensión (d=256), la aceleración de V_x se estabiliza en 38x, lo que significa que esta etapa es 38 veces más rápida en GPU que implementada en una CPU tradicional con una dimension de 256 pixeles.

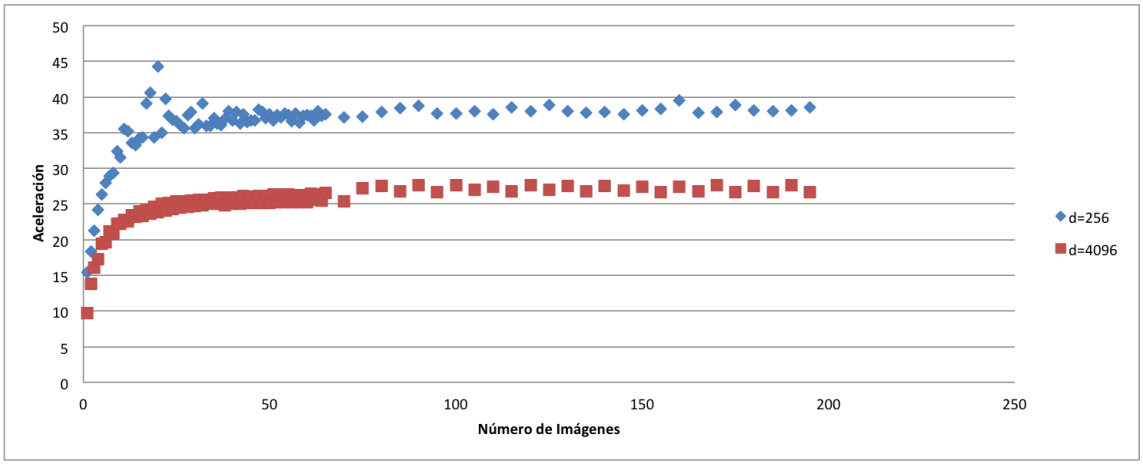


Figura 4.16: Aceleración de Vx con d=256 y d=4096.

La figura 4.17 muestra la aceleración de la etapa de construcción de T_{xx} que se logra en la GPU. Al igual que en la etapa de construcción de V_x , la aceleración crece a medida que la dimensionalidad del problema disminuye, llegando en este caso a ser 27 veces más rápida la implementación en GPU que la usada en una CPU tradicional con una dimension de 256 pixeles.

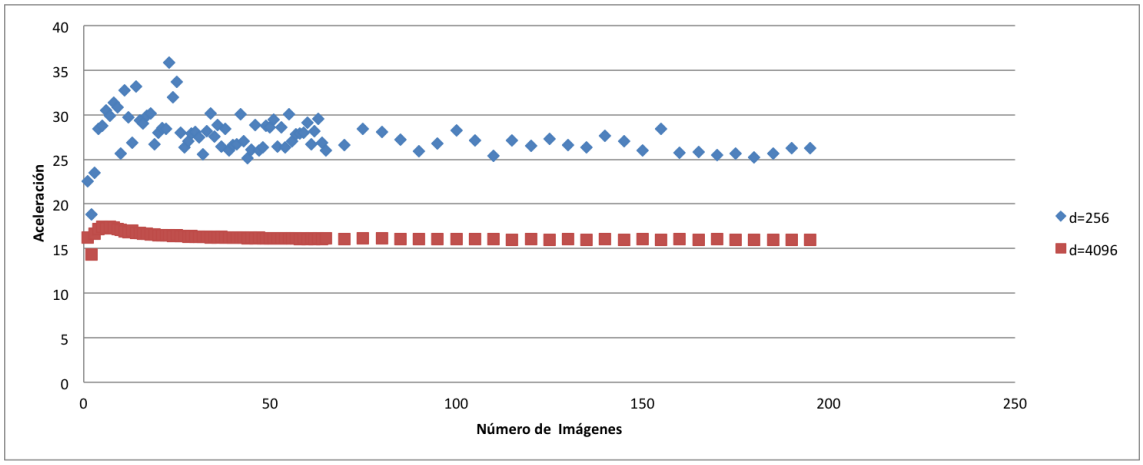


Figura 4.17: Aceleración Txx con d=256 y d=4096.

La figura 4.18 muestra la aceleración obtenida en la etapa de construcción de T_{zx}. En este caso la aceleración alcanza una aceleración promedio de 55, lo que quiere decir que la implementación en GPU es 55 veces más rápida que la usada en una CPU tradicional con una dimension de 256 pixeles.

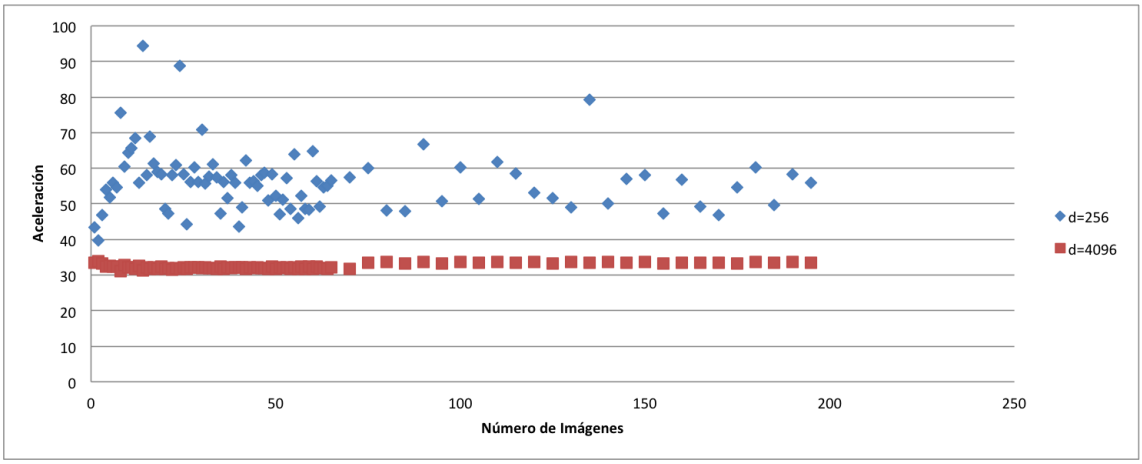


Figura 4.18: Aceleración de Tzx con d=256 y d=4096.

La figura 4.19 muestra el tiempo total del filtro CMACE en GPU, luego de usar proyección aleatoria como preprocesamiento de imágenes(d=256).

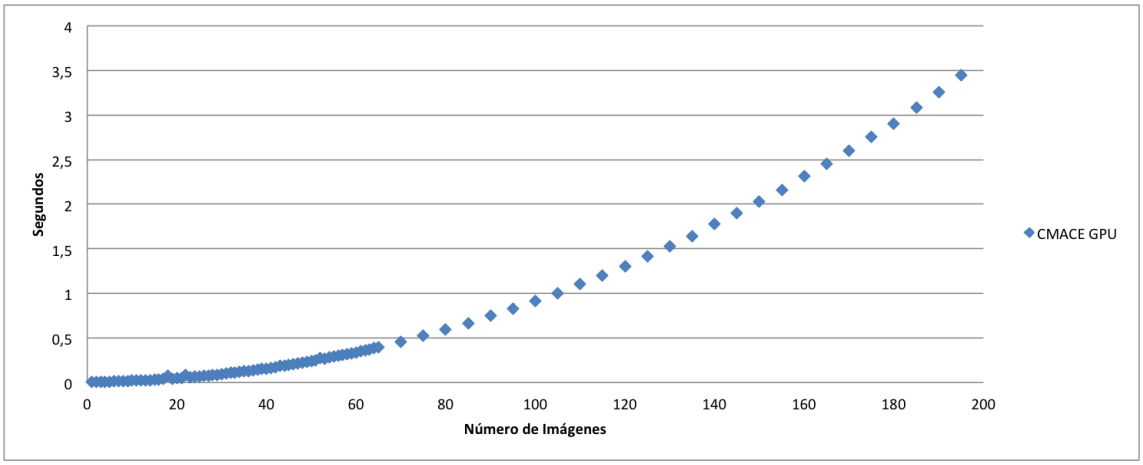


Figura 4.19: Tiempo Total de Ejecución CMACE en GPU con d=256.

La figura 4.20 muestra la aceleración alcanzada por el filtro CMACE en GPU, luego de usar proyección aleatoria como preprocesamiento de imágenes(d=256). El intervalo de tiempo de ejecución llega a ser 26 veces más corto en la GPU que en una CPU tradicional con una dimension de 256 pixeles.

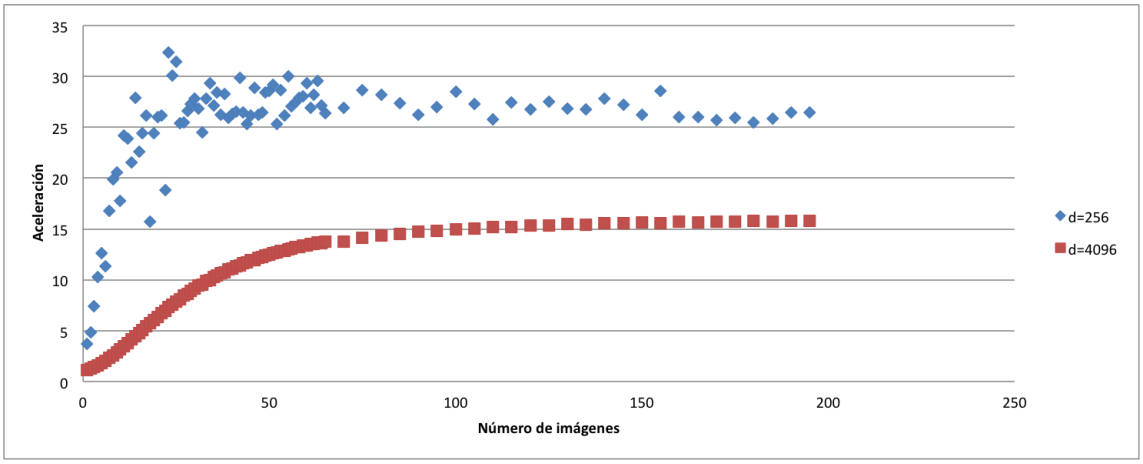


Figura 4.20: Aceleración CMACE con d=256 y d=4096.

Cada etapa paralelizada, tiene una aceleración mayor dimensiones más bajas, por lo que la aceleración completa del filtro CMACE también tiene este comportamiento.

4.1.7. Transferencia de Datos Entre CPU y GPU

La tabla 4.1 muestra las tasas de transferencia de Datos que se tiene entre GPU y CPU al momento de ejecutar el Filtro CMACE.

	$CPU \rightarrow GPU$	$GPU \rightarrow CPU$
Promedio [GigaByte/Seg]	1,6218	0,667
Desviación Estándar [GigaByte/Seg]	0,1063	0,011

Tabla 4.1: Tasa de transferencia de datos entre CPU y GPU, y viceversa.

Las tasas de transferencia de datos se mantienen constantes en todas las configuraciones y decrece su relevancia a medida que la GPU tiene más datos a procesar. Es importante hacer la menor cantidad de llamadas al Bus de datos entre la GPU y la CPU, ya que existe un tiempo de latencia relacionado esta acción. En la implementación del filtro CMACE en GPU, cada Kernel hace una llamada al inicio (transferencia de CPU a GPU) y una llamada al finalizar la ejecución (transferencia de GPU y CPU) para reducir los costos computacionales relacionados con la latencia que tiene el Bus de datos.

La tabla 4.2 muestra el costo computacional relacionado con la transferencia de datos desde la CPU a la GPU como porcentaje del tiempo total de ejecución de cada Kernel.

$CPU \rightarrow GPU$	V_x	T_{xx}	T_{zx}
Promedio	0,021 %	1 %	1,38 %
Desviación Estándar	0,001 %	0,46 %	0,9 %

Tabla 4.2: Transferencia de Datos $CPU \rightarrow GPU$ Relativa al Tiempo Total de Ejecución de cada programa kernel.

La tabla 4.3 muestra el costo computacional relacionado con la transferencia de datos desde la GPU a la CPU como porcentaje del tiempo total de ejecución de cada Kernel.

$GPU \rightarrow CPU$	V_x	T_{xx}	T_{zx}
Promedio	4,7 %	0,00056 %	0,004 %
Desviación Estándar	1,8 %	0,003 %	0,01 %

Tabla 4.3: Transferencia de Datos $GPU \rightarrow CPU$ Relativa al Tiempo Total de Ejecución de cada programa kernel.

En todas las mediciones de tiempo que se hacen al Filtro CMACE para calcular sus costos computacionales y su aceleración respecto a la implementación tradicional en CPU, se toman en cuenta los tiempos asociados a la transferencia de datos.

4.1.8. Velocidad de Procesamiento

La velocidad de procesamiento del procesador es algo muy difícil de medir, ya que en este caso cada programa Kernel tiene varias llamadas de memoria global, tanto para lectura como para escritura que son tiempos muy superiores a los que toma al procesador hacer una operación de punto flotante (alrededor de 30 veces más costoso).

Para estimar la velocidad de procesamiento, se calculó la cantidad de pixeles por segundo que procesa cada etapa en la implementación en CPU y en la implementación en GPU. La tabla 4.4 muestra los pixeles por segundo promedio que procesa cada etapa (V_x ,T_{xx} y T_{zx}), en la implementación en CPU y en la implementación en GPU. El promedio de pixeles por segundo que procesa cada etapa, se calculó promediando la razón entre los pixeles involucrados en cada etapa y el tiempo de ejecución para las diferentes configuraciones. La desviación estandar no supera el 2 % del valor del promedio en ninguno de los tres casos (V_x ,T_{xx} y T_{zx}), esto se debe a que en ambos dispositivos se usan todos los recursos disponibles independiente del tamaño o configuración de los problemas usados en los experimentos.

La tabla 4.4 muestra que mientras la CPU procesa 13, 49 y 26 millones de pixeles por segundos en cada etapa, la GPU es capaz de procesar 350, 795 y 847 millones de pixeles por segundo en cada etapa.

	V_x	T_{xx}	T_{zx}
CPU			
Promedio	13635361	49029144	26123837
Desviación Estándar	301096	823267	588835
GPU			
Promedio	**351877442**	**795313742**	**847979686**
Desviación Estándar	12696009	2694031	131638

Tabla 4.4: Pixeles por segundo en GPU y CPU.

Capítulo 5

Conclusiones

- Se implementó satisfactoriamente una versión del Filtro CMACE usando la GPU en la paralelización de las etapas más importantes.
- Se mejoró el costo computacional del Filtro CMACE mediante el uso de la GPU.
- Los costos computacionales que se usaron en la transferencia de datos entre la GPU y la CPU son despreciables frente a los tiempos de cómputos, lo que significa que se hizo buen uso de los recursos de este dispositivo, usando su tiempo de funcionamiento en la ejecución de los Kernels.
- En el diseño de los Kernels se tuvo en cuenta el tamaño de memoria necesario para cada configuración y con esto las posibles limitaciones en procesamiento de acuerdo a las características de cualquier GPU con arquitectura CUDA.
- Se estimó la velocidad de procesamiento de la GPU y de la CPU para cada etapa implementada en este Filtro.
- La velocidad de procesamiento muestra diferencias significativas entre ambos dispositivos. La implementación en serie procesa 13, 49 y 26 millones de pixeles por segundos en cada etapa, mientras que la implementación en paralelo propuesta en este trabajo es capaz de procesar 350, 795 y 847 millones de pixeles por segundo en cada etapa.
- La velocidad de ejecución aumenta a medida que hay aumentan las operaciones que se realizan en cada Kernel y disminuyen los datos necesarios para el cómputo.
- La versión del Filtro CMACE usando GPU no presenta errores de cómputo al utilizar datos de doble precisión. Esta opción solo es posible en tarjetas gráficas diseñadas para propositos generales y super cómputo.
- Al tener una gran aceleración en la etapa de clasificación, la versión del filtro CMACE en GPU permite el uso de este método en una mayor cantidad de problemas en el que se necesite respuestas a tiempo real.
- Se presentó una mayor aceleración al reducir la dimensión de las imágenes. Esto si bien ayuda en la reducción de los costos computacionales, el poder de clasificación se ve perjudicado.
- El preprocesamiento de los datos mediante Proyección Aleatoria disminuyen las limitaciones de memoria calculadas para los Kernels, aumentando la cantidad de

imágenes máximo que es posible procesar en la GPU.

- Utilizar la GPU en la implementación del Filtro CMACE hizo que los costos computacionales se redujeran sin sacrificar el poder de clasificación que tiene este método (como es el caso del uso de proyección aleatoria) , llegando a ser 16 veces más rápido.
- Combinando Proyección Aleatoria y la paralelización mediante GPU, el filtro llegó a ser 26 veces más rápido.
- Se probó y comparó el poder de clasificación del Filtro CMACE en GPU mediante distintas experiencias.
- Mediante la visualización de las curvas ROC para las distintas experiencias de clasificación, se pudo concluir que el filtro CMACE presenta grandes ventajas por sobre el método convencional MACE.
- Como trabajo futuro se propone trabajar en la paralelización de la etapa de invertir las matrices a través de arquitectura de memoria compartida usando varios núcleos de CPU.
- Se propone además la paralelización de cada programa Kernel usando varias GPU como un dispositivo de memoria distribuida pero usando GPU en vez de CPU para el procesamiento de datos.

Bibliografía

[amplaeceC] advanced multimedia processing lab at electrical and computer eng. CMU. http://www.amp.ece.cmu.edu.

[Aro50] N. Aronszajn. Theory of reproducing kernels. *Transactions of the American Mathematical Society*, 68, 1950.

[BM01] Ella Bingham and Heikki Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '01, pages 245–250, New York, NY, USA, 2001. ACM.

[GP09] Aysegul Gunduz and Jose C. Principe. Correntropy as a novel measure for nonlinearity tests. *Signal Processing*, 89(1):14–23, 2009.

[JLH+09] Kyu-Hwa Jeong, Weifeng Liu, Seungju Han, Erion Hasanbelliu, and Jose C. Principe. The correntropy mace filter. *Pattern Recognition*, 42(5):871 – 885, 2009.

[JP08] Kyu-Hwa Jeong and Jose C. Principe. Enhancing the correntropy mace filter with random projections. *Neurocomputing*, 72(1-3):102 – 111, 2008. Machine Learning for Signal Processing (MLSP 2006) / Life System Modelling, Simulation, and Bio-inspired Computing (LSMS 2007).

[LPP] Weifeng Liu, Puskal P. Pokharel, and José C. Príncipe. Correntropy: Properties and applications in non-gaussian signal processing. *IEEE Trans. on Signal Processing*.

[LWZ04] Yi Li, Zhiyan Wang, and Haizan Zeng. Correlation filter: An accurate approach to detect and locate low contrast character strings in complex table environment. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:1639–1644, 2004.

[ND] NVIDIA-Developer. http://developer.nvidia.com.

[NVI] NVIDIA. http://www.nvidia.com.

[Sil86] B. W. Silverman. *Density estimation: for statistics and data analysis*. Lon-

don, 1986.