UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
ESPECIALIZACIÓN EN CÓMPUTO DE ALTO RENDIMIENTO
INSTITUTO DE INVESTIGACIONES EN MATEMÁTICAS APLICADAS Y SISTEMAS UNAM IIMAS

ENTRENAMIENTO DE REDES NEURONALES CONVOLUCIONALES Y MAX POOLING POR MEDIO DE TÉCNICAS DE CÓMPUTO DE ALTO RENDIMIENTO

TESINA PARA OBTENER EL TITULO DE
ESPECIALIZACION EN CÓMPUTO DE ALTO RENDIMIENTO

PRESENTA:
ING. JUAN CARLOS LÓPEZ NÚÑEZ

TUTOR
DR. HECTOR BENITEZ PEREZ
INSTITUTO DE INVESTIGACIONES EN MATEMÁTICAS APLICADAS Y SISTEMAS UNAM IIMAS

COMITÉ TUTORIAL
ING. ADRIAN DURÁN CHAVESTI
DR. ERNESTO RUBIO ACOSTA
INSTITUTO DE INVESTIGACIONES EN MATEMÁTICAS APLICADAS Y SISTEMAS UNAM IIMAS

CIUDAD DE MÉXICO, CIUDAD UNIVERSITARIA, JUNIO 2023

RESUMEN

Las aplicaciones del aprendizaje de máquina en la última década han incrementado las cuales van desde diversas aplicaciones tales como la detección de patrones, el reconocimiento de imágenes o de texto. Debido a que estos programas son computacionalmente demandantes debido a que para el entrenamiento de la red se necesita de una gran cantidad de operaciones de algebra lineal y procesar en ocasiones gran cantidad de información para el entrenamiento de la red se busca utilizar técnicas de cómputo de alto rendimiento que permitan acelerar los tiempos de procesamiento de las operaciones que conforman la red neuronal, así como el entrenamiento de la red.

Para este proyecto se busca comprender las características de las redes convolucionales por medio de la red de arquitectura AlexNet para familiarizarse con las características de esta técnica. Se busca realizar en el presente trabajo un análisis más detallado del funcionamiento interno de las redes convolucionales por medio de la operación de Max Pooling siendo esta una parte importante del diseño de las capas que conforman una red convolucional. Debido a que esta es una operación que se considera computacionalmente intensiva en su procesamiento en el presente trabajo se buscara identificar las secciones del código que sean paralelizables y poder aplicar técnicas de cómputo de alto rendimiento que mejoren su desempeño y tengan una serie de beneficios por el uso de estas técnicas en el procesamiento de la operación de Max Pooling.

AGRADECIMIENTOS

A mis padres por confiar en mí.

Índice General

Índice de figuras

Introducción

En el presente trabajo se busca el desarrollo de programas relacionados con el área de redes convolucionales y la posterior implementación de técnicas de cómputo de alto rendimiento para mejorar su procesamiento debido a que estos problemas se les considera computacionalmente intensivos. Para alcanzar estos objetivos se busca el desarrollo de un programa de Max Pooling la cual se considera una de las 3 operaciones fundamentales que conforman las capas de las redes convolucionales junto la función de activación ReLU y la operación de convolución. La operación de Max Pooling tiene la propiedad de que presenta procesos independientes durante la extracción de características de la imagen por lo que este programa puede ser un candidato para que este programa sea paralelizable y se puedan utilizar técnicas de CAR para mejorar su desempeño. Para este proyecto también se busca familiarizarse con las técnicas de redes convolucionales las cuales tienen actualmente un papel importante tanto en la investigación como la industria.

Actualmente las redes convolucionales tienen gran cantidad de aplicaciones donde su principal uso es el de la clasificación de imágenes a partir de un conjunto de entrenamiento dividido en clases. Desde el éxito de AlexNet en la competencia ILSVRC en 2012 se han logrado avances considerables siendo esta una aportación considerable al campo del aprendizaje profundo adoptando muchas características de este diseño tal como el uso de la función de activación ReLU. Esta red se compone de 8 capas para las funciones de extracción de características y clasificación donde las primeras 5 son convolucionales mientras las ultimas 3 son capas completamente conectadas, la primeras dos de estas capas conectada a través de Dropout mientras la última conectada con una capa Softmax.

La red AlexNet tiene gran diversidad de aplicaciones en tareas de clasificación entre las que se puede mencionar como herramienta de diagnóstico temprano y exploratorio preventivo contra la detección de cáncer para que los pacientes al tener un diagnóstico oportuno les permita tener mayores posibilidades de recuperación de dicho padecimiento. Para este trabajo se utilizó una red de la arquitectura previamente mencionada y su entrenamiento se realizó con un conjunto de imágenes cerebrales obtenidos de una base de datos médicas.

Descripción del conjunto de datos datasetMRI

El conjunto de datos consiste en 3 clases cada una con 91 imágenes en formato JPG de las cuales fueron previamente convertidas de formato NIFTI por medio de la aplicación med2image en Linux Ubuntu 20.04.

https://github.com/FNNDSC/med2image

Las imágenes están con el nombre de la clase seguido de un numero con una numeración que va del 0 hasta el 90. Este conjunto de datos originalmente se piensa usar para clasificación por medio de redes convolucionales. Las dimensiones de las imágenes son de 91x109 (ancho X alto) en formato JPG. Se incluye archivos de valores en forma de archivos CSV los cuales pueden funcionar como etiquetas para tareas de clasificación por medio de métodos supervisados.

OBJETIVOS

La red neuronal implementada en PyTorch es una red AlexNet con modificaciones para el entrenamiento y clasificación del conjunto de imágenes cerebrales. El objetivo de este proyecto es acerca de aplicar técnicas de cómputo de alto rendimiento para el entrenamiento de redes neuronales convolucionales, así como familiarizarse con el funcionamiento de las capas que lo conforman por medio de la implementación de la operación de Max Pooling. El entrenamiento de la red neuronal se busca que se haga con una red de arquitectura AlexNet en CPU y en GPU.

Para el conjunto de prueba solo se toman las primeras 20 imágenes de cada una de las clases, mientras que para poder visualizar las imágenes en formato NIFTI se utilizó el visor de imágenes de software libre Mango.

HIPÓTESIS

La operación de Max Pooling es un componente que se encuentra presente en diferentes arquitecturas de redes neuronales actualmente incluyendo las redes convolucionales siendo este uno de sus 3 componentes básicos que lo conforman. En el presente trabajo se busca mejorar su procesamiento a través de técnicas de cómputo de alto rendimiento tomando en cuenta que esta operación tiene independencia en sus procesos, razón por la cual es posible paralelizar para el procesamiento de una operación que se considera computacionalmente intensiva.

METAS

Se busca en el presente trabajo desarrollar un programa que realice la operación de Max Pooling para poder obtener de esta forma beneficios de la paralelización que beneficien el cálculo de esta operación tales como mejorar la velocidad de procesamiento, repartir la carga de procesamiento entre los núcleos disponibles haciendo operaciones más sencillas por cada procesador reduciendo la carga computacional por núcleo y una mejor distribución de los datos que representan el tamaño de la entrada de datos el cual crece considerablemente conforme aumenta el tamaño de la imagen de entrada. Estos beneficios se esperan que consigan una reducción de los tiempos de ejecución, así como un incremento de Speedup.

ALCANCES

Los alcances de este proyecto se basan en el mejoramiento del procesamiento de la operación de Max Pooling por medio de técnicas de CAR como una forma de mejorar el procesamiento de las redes neuronales las cuales actualmente en el momento de la realización de este trabajo se consideran computacionalmente intensivas. Existen actualmente otras arquitecturas que forman parte del estado del arte tales como las redes recurrentes, aprendizaje por reforzamiento, entre otras arquitecturas recientes en el campo de aprendizaje profundo por lo que no se busca en este trabajo proponer una nueva arquitectura ni resolver problemas de relevancia para el aprendizaje profundo como el problema del desvanecimiento del gradiente, o mejorar los porcentajes de error en la detección de los patrones para tareas de clasificación, por lo que estos problemas se encuentran fuera del alcance del presente trabajo.

En este trabajo se concentrará en comprender la operación de Max Pooling y las características que la conforman para poder aplicar técnicas de alto rendimiento que mejoren su procesamiento, así como experimentos con la red AlexNet para poder realizar el entrenamiento tanto en CPU como en GPU para buscar mejores tiempos de aceleración de la red y familiarizarse con las características de la red neuronal y el papel que juega el Max Pooling en las redes convolucionales.

Para este proyecto, aunque se deben comprender los principios de las redes neuronales convolucionales los alcances de este proyecto se enfocan en las técnicas de cómputo de alto rendimiento para mejorar los tiempos de entrenamiento por lo que podemos ver este trabajo desde dos puntos de vista, el de inteligencia artificial por las operaciones de Max Pooling y las características de la red AlexNet y las técnicas de cómputo de alto rendimiento utilizadas sobre estos programas para su procesamiento.

Las redes neuronales son programas que tienen una carga computacional intensiva debido a la gran cantidad de datos que se necesitan para los conjuntos de entrenamiento. Las operaciones matriciales que están presentes en las operaciones fundamentales de las redes neuronales tales como el Max Pooling y la convolución, así como las funciones de activación para su procesamiento requiere de gran cantidad de ciclos por lo que las tareas de estas operaciones pueden ser paralelizables por técnicas tales como el uso de las GPUs que permiten paralelización y dividir la tarea en una gran cantidad de núcleos que conforman la GPU.

PROPUESTA

Como propuesta para este trabajo se busca la implementación de la técnica de Max Pooling la cual a partir de extraer elementos de la imagen recibida obtiene un sumario estadístico de la imagen produciendo un mapa de características con los valores más alto de cada kernel o ventana obtenida. Cada uno estos kernels tiene la propiedad de ser un proceso independiente incluso si las ventanas se solapan debido a la configuración de los hiper parámetros. Esta propiedad de independencia de procesos permite que estos procesos de multiplicación puedan ser paralelizables por medio de distintas técnicas de paralelización.

La implementación del programa de Max Pooling se realiza por medio de Python debido a que este es un lenguaje de alto nivel con muchas librerías disponibles que permiten un desarrollo más ágil del programa además de ser el lenguaje actualmente más utilizado para las redes neuronales, técnicas de aprendizaje de maquina y ciencia de datos.

Este trabajo se realiza con fines académicos con el objetivo de que por medio de la operación de Max Pooling pueda obtenerse beneficios por parte de la implementación de las técnicas de paralelización, así como la familiarización en las técnicas de CAR que actualmente se utilizan en distintos programas relacionados con el área de aprendizaje profundo.

CONCEPTOS DE APRENDIZAJE PROFUNDO

Definición de tensores

Los tensores son objetos algebraicos que su objetivo es describir una relación multilineal entre conjuntos de objetos relacionados en un espacio vectorial. Desde el punto de vista computacional los tensores se pueden representar como una matriz de n dimensiones, la cual se puede ver como una generalización de los vectores y matrices. En los tensores la suma y la resta entre elementos de las mismas dimensiones está definida y dicha suma debe ser de elemento a elemento.

Desde el punto de vista computacional un tensor es una matriz de n dimensiones donde cada capa representa un canal de información de la imagen. Para el caso de una imagen RGB de 3 colores se representa por medio de un tensor de 3 canales donde cada uno de estos canales consiste en una representación matricial donde se encuentra 1 matriz por cada uno de los 3 colores de la imagen. Para una imagen en escala de grises la representación es una matriz de un solo canal. Esta representación de un solo canal será la utilizada para el presente trabajo como imagen de entrada sobre el cual se aplicará la imagen de Max Pooling por medio de una matriz bidimensional con valores aleatorios que van desde el 0 hasta el 9.

FUNCIONES DE ACTIVACIÓN

La función de activación permite que la red neuronal actúe ante un estímulo dado. Actualmente la más utilizada es la función ReLU. Desde que se utilizó en la red AlexNet se ha convertido en una función muy utilizada en las redes neuronales ya que permite un aprendizaje más rápido, otra de las ventajas de la función ReLU es que no requiere normalización de la entrada para prevenir la saturación. Otras funciones como funciones de activación son la tangente hiperbólica y la función sigmoide.

GRADIENTE DESCENDIENTE

El algoritmo de gradiente descendiente minimiza la función dada conocida también como función de costo. El algoritmo consiste en los siguientes pasos.

1.-Calcular el gradiente. La primera derivada de primer orden de la función.

2.-Hacer el paso en la dirección opuesta al gradiente. La pendiente empieza a incrementarse hasta llegar al punto correspondiente por alfa veces el gradiente a ese punto donde alfa representa la tasa de aprendizaje.

Se debe tener un ajuste adecuado en la tasa de aprendizaje ya que puede llegar a afectar el comportamiento del gradiente descendiente. Una tasa de aprendizaje pequeña puede llevar a una convergencia baja debido a que se requerirán muchos pasos del gradiente. En caso de que la tasa de aprendizaje sea demasiado alta la búsqueda tiene un sobre impulso y con oscilaciones alrededor del mínimo. La función puede llegar a tener muchos mínimos locales por lo que se considera una solución adecuada colocarse en un mínimo local, aunque no se llegue al mínimo absoluto. La optimización puede llegar a converger a diferentes puntos con distintos puntos de inicio y tasa de aprendizaje. Alguna de las tareas más comunes en el aprendizaje de maquina son las siguientes.

CLASIFICACIÓN

En este tipo de tarea, un programa de computadora nos lleva a especificar cuáles de las k categorías algunas entradas puede llegar a pertenecer. Cuando se tiene y=f(x), el modelo asigna una entrada descrita por un vector x a una categoría identificada por un código numérico y. Un ejemplo de las tareas de clasificación es el reconocimiento de objetos, donde la entrada es una imagen, y la salida es un código numérico identificando el objeto en la imagen. A continuación, se mencionan brevemente algunos conceptos relacionados con las tareas de clasificación.

Algoritmos no supervisados

Experimentan un conjunto de datos conteniendo muchas características y aprender propiedades de la estructura del conjunto de datos. Dentro del contexto del aprendizaje profundo se quiere aprender la distribución de probabilidad entrante que genero el conjunto de datos.

Algoritmos supervisados

Experimentan un conjunto de datos conteniendo características, pero cada ejemplo es asociado con una etiqueta o un objetivo. Los algoritmos no supervisados involucran muchos ejemplos de un vector aleatorio x y tratan de leer la distribución de probabilidad p(x).

Modelos no paramétricos.

Los modelos paramétricos aprenden una función descrita por un vector de parámetro. Los modelos no paramétricos son abstracciones teóricas que no pueden ser implementadas en la práctica. El error de entrenamiento y generalización varia conforme el tamaño de los conjuntos de entrenamiento varia. Los errores de generalización esperados no pueden nunca incrementar conforme el número de ejemplos de entrenamiento se incrementa. Para el caso de modelos no paramétricos más datos llevan a mejor generalización hasta que el mejor error posible es alcanzado.

El teorema de lunch no gratis

La teoría de aprendizaje que un algoritmo de aprendizaje de maquina puede generalizar de un conjunto de entrenamiento de ejemplos. El teorema del lunch no gratis establece que promediado sobre todas las distribuciones de generación de datos cada algoritmo de clasificación tiene la misma tasa de error cuando se clasifico previamente puntos no observables. Para consultar esta información e información adicional acerca de clasificación referirse a (Goodfelow, 2016, 113).

Hiper parámetros y conjuntos de validación.

Los hiper parámetros son ajustes de los cuales podemos tener control para ajustar el comportamiento del algoritmo. Se necesita de un conjunto de validación de ejemplos que el algoritmo de entrenamiento no pueda observar. Los otros conjuntos tal como el de validación usado para estimar el error de generalización durante o después del entrenamiento, permitiendo los hiper

parámetros actualizarse acordemente. El subconjunto de datos usado para aprender los parámetros es llamado comúnmente como el conjunto de entrenamiento.

Nos preocupamos en el comportamiento de un estimador conforme el monto de entrenamiento crece. En particular deseamos que mientras el número de puntos dado m en el conjunto de datos se incrementa, los estimadores de puntos m aumenten el conjunto de datos se incrementa, los puntos de estimación convergen al valor verdadero de los parámetros correspondientes. El símbolo plim indica convergencia en probabilidad, significando que por cada E > 0. La consistencia asegura que el bias inducido por el estimador disminuye conforme los ejemplos del número de datos crece. Se puede usar la primera muestra x(1) del conjunto de datos es un estimador no basado sin importar cuantos puntos de datos están vistos. El más común de los estimadores es el principio de similitud máximo.

Propiedades de la máxima similitud

Sobre condiciones apropiadas el estimador de similitud máxima tiene la propiedad de consistencia significando que conforme el número de ejemplos de entrenamiento va acercándose al infinito, la similitud máxima de un parámetro converge al verdadero valor del parámetro. Para más información referirse a (Goodfellow et al, 2016, 113).

La maldición de la dimensionalidad

Los problemas de aprendizaje de maquina se incrementan conforme el número de dimensiones de los datos es alto. Un problema sucede cuando el número de distintas posibles configuraciones de un conjunto de variables se incrementa exponencialmente conforme el número de variables aumenta. Al mismo tiempo cuando aumenta el número de dimensiones es más difícil distinguir valores diferentes de cada variable.

REDES NEURONALES CONVOLUCIONALES

Las redes neuronales convolucionales son una forma específica de red neuronal para procesar datos que tienen una topología de malla. La red convolucional se compone básicamente de 3 operaciones principales. La convolución, el pooling y una función de activación.

La convolución es una forma especializada de operación lineal. Las redes convolucionales son simplemente redes neuronales que usan la convolución en lugar de la multiplicación general de matrices en al menos una de las capas. La operación de convolución se denota típicamente por medio de un asterisco.

$s(t) = (x * w)(t)$

Donde w es una función de densidad de probabilidad valida, o la salida no es un promedio ponderado. W necesita ser 0 para todos los argumentos negativos. Este argumento se conoce como kernel. La función x se refiere como la entrada a la convolución. La salida se conoce como mapa de características. El índice de tiempo t puede tomar únicamente valores enteros. Si se asumen que x

y w se definen solamente sobre un entero t, definimos la convolución discreta. En aplicaciones de aprendizaje de máquina la entrada es por lo general un arreglo multidimensional de datos, y el kernel es usualmente un arreglo multidimensional de datos. El kernel es un arreglo multidimensional de parámetros que son adaptados del algoritmo de aprendizaje conocidos estos como tensores. Para más información consultar las siguientes referencias. (Goodfellow et al, 2016) (Agrawall, 2019).

Pooling

Una capa típica de una red convolucional consiste en 3 fases. La primera capa hace muchas convoluciones en paralelo para producir un conjunto de activaciones lineales. En la segunda etapa, cada activación lineal funciona a través de una función de activación no lineal esta etapa se conoce como etapa detectora. En la tercera etapa se usa una función de pooling para cambiar la salida de la capa.

La operación de Max Pooling reporta la salida máxima junto con una vecindad rectangular. Esta operación usa el concepto de invarianza; es decir, las unidades de esta capa dentro de la red convolucional solamente son sensibles al valor máximo en el vecindario, no su localidad exacta. Pooling hace que la representación se vuelva aproximadamente invariante a pequeñas traslaciones de la entrada. La función de pooling reemplaza la salida de la red en una cierta localidad con un resumen estadístico de las salidas cercanas. Existen diferentes tipos de pooling dentro de los más comunes tenemos max pooling y average pooling. Para el caso de max pooling la operación reporta la máxima salida junto a una vecindad rectangular.

La idea de la operación de Max Pooling trata acerca de extraer características de una región particular de la imagen. Es posible aplicar una operación sobre dicha región una vez realizado el proceso de multiplicación de matrices. En el caso de Max Pooling se extrae el valor de mayor peso y se coloca en el mapa de características que conforma la imagen final. Esta operación disminuye el costo computacional al reducir el tamaño de la imagen original. Otra característica importante es la invarianza de la traslación, lo cual permite el reconocimiento de las características de la imagen sin importar la posición en la cual se encuentren dichas características. Esta operación puede llegar a presentar reducciones de la imagen, así como perder información durante la extracción de características en el momento de sustraer los kernels de la imagen de entrada. La reducción de la imagen por parte de esta operación permite mantener el control de la imagen conforme se aumentan las capas de la red neuronal en el caso de las redes convolucionales.

Normalmente se utilizan en Max Pooling matrices bidimensionales para el caso de imágenes de un solo color que se puede considerar como un tensor de un solo canal o de 3 canales en caso de imágenes RGB con los hiperparametros como un desplazamiento (stride) de (2,2) y un kernel también conocido como ventana de 2X2 sin uso de padding, por lo que será la forma de trabajar para los experimentos. Por lo regular se utilizan kernels de orden 2 o 3 y strides > 1 para evitar solapamiento de los kernels los cuales permiten ajustes en la imagen durante el proceso de Max Pooling. Al incrementar el kernel se disminuye la resolución y se tiene una mayor pérdida de información, pero se tiene una mayor extracción de características de las regiones extraídas.

Max pooling e independencia de procesos

La selección de los hiper parámetros influyen en el resultado del mapa de características al momento de aplicar la operación de Max Pooling. Los hiperparámetros de esta operación son el desplazamiento y el tamaño del kernel, así como el padding influye en el número de kernels que pueden ser extraídos. Cada uno de estos kernels son independiente uno del otro incluso en el caso de que los kernels tengan un solapamiento con elementos en común de otro kernel en el caso de que se tenga un desplazamiento (stride) de 1. Esta propiedad de la operación de pooling en la independencia de procesos es lo que permite que esta operación pueda ser paralelizable incluso si los kernels tienen solapamiento. La operación de Max Pooling se basa en una serie de multiplicaciones de matrices por cada kernel extraído. La multiplicación de matrices cuadradas tiene las propiedades del producto de Schur las cuales son conmutativas, distributivas y asociativas.

La complejidad de la multiplicación de matrices es de n^3 multiplicaciones en el peor caso y $(n-1)n^2$ sumas de escalares para el cálculo de 2 matrices cuadradas nxn. En la notación de big O esta operación tiene complejidad de $O(n^3)$. Esta carga computacional a través de la paralelización por medio de hilos se reparte entre el número de procesadores disponibles. La multiplicación de matrices es una operación que puede ser paralelizable ya que cada una de las multiplicaciones renglón-columna es un proceso independiente. Esta operación paralelizable sigue el modelo SPMD (Single Program Multiple Data). La operación de multiplicación de matrices es una operación fundamental en aplicaciones computacionales de algebra lineal, matemáticas aplicadas y física.

La independencia de los procesos al extraer el kernel de la imagen de entrada incluso aun con solapamiento entre los kernels cuando se tiene un stride =1 es lo que permite la paralelización de las multiplicaciones de matrices debido a que el número de operaciones generadas son independientes de los datos. En la multiplicación de matrices cada elemento calculado en la matriz resultado C_{ij} es independiente de los demás elementos. En el siguiente ejemplo se ve el proceso de calcular el Max Pooling a partir de la siguiente imagen de entrada con los siguientes parámetros establecidos.

- Tamaño de kernel = 2
- Stride = 2
- Padding = 0

Para las multiplicaciones de matrices se utilizó la matriz identidad como filtro para ser un elemento neutro en la multiplicación de matrices por medio de la propiedad A = I * A. Con la configuración de hiperparámetros establecida se extraerán para este ejemplo 4 kernels donde cada uno de estos se obtendrá a partir de su multiplicación correspondiente como se puede ver en el siguiente ejemplo.

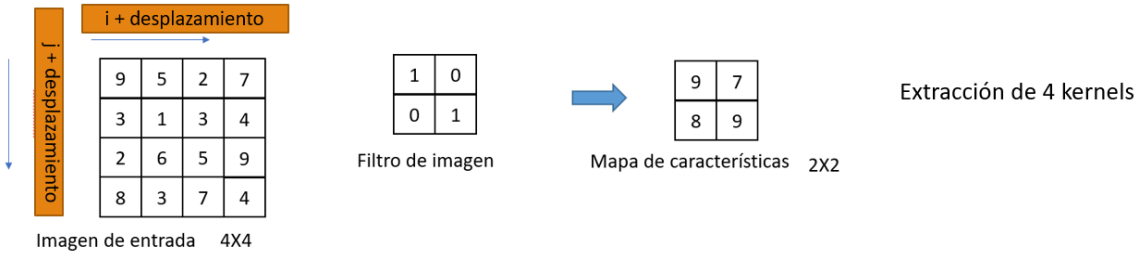

FIGURA 1. IMAGEN DE ENTRADA Y MAPA DE CARACTERISTICAS DESPUES DE APLICAR MAX POOLING

A continuación, se muestra el proceso de extracción de los kernels junto con sus correspondientes multiplicaciones. Cada uno de estos procesos es independiente y esto convierte a este problema como candidato a que la multiplicación de matrices pueda ser paralelizable. Una vez realizada la multiplicación se obtiene el elemento mayor de la matriz resultante, estos valores obtenidos se utilizarán para formar el mapa de características de la operación de Max Pooling.

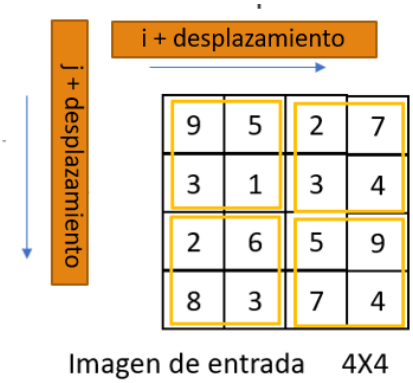

FIGURA 2. MULTIPLICACIONES DE MATRICES A PARTIR DEL MAPA DE CARACTERISTICAS COMO PROCESOS INDEPENDIENTES.

$$\begin{pmatrix}9 & 5\\ 3 & 1\end{pmatrix}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} = \begin{pmatrix}9 & 5\\ 3 & 1\end{pmatrix} \rightarrow max(9,5,3,1) = 9$$

$$\begin{pmatrix}2 & 7\\ 3 & 4\end{pmatrix}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} = \begin{pmatrix}2 & 7\\ 3 & 4\end{pmatrix} \rightarrow max(2,7,3,4) = 7$$

$$\begin{pmatrix}2 & 6\\ 8 & 3\end{pmatrix}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} = \begin{pmatrix}2 & 6\\ 8 & 3\end{pmatrix} \rightarrow max(2,6,8,3) = 8$$

$$\begin{pmatrix}5 & 9\\ 7 & 4\end{pmatrix}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} = \begin{pmatrix}5 & 9\\ 7 & 4\end{pmatrix} \rightarrow max(5,9,7,4) = 9$$

PROCESO DE MULTIPLICACIÓN DE MATRICES Y PARALELIZACIÓN

La multiplicación de matrices es un proceso paralelo donde la multiplicación de cada uno de sus elementos es independiente del otro. Para este programa se utilizó matrices cuadradas en la

multiplicación de matrices para mantener las propiedades del producto de Hadamard o de Schur las cuales son asociativas y distributivas en la suma.

$$A * B = B * A$$

$$A * (B * C) = (A * B) * C$$

$$A * (B + C) = A * B + A * C$$

Antes de la aplicación del proceso de paralelización se necesita identificar la región paralela la cual debido a su independencia de procesos es la sección de código que se puede paralelizar. En la multiplicación de matrices los ciclos for que se mantienen anidados para el proceso de multiplicación de matrices $C_{ij} = a_{i0}b_{0j} + a_{i1}b_{1j} + \cdots + \ a_{i,n-1}b_{n-1,j}$ en la operación de suma de la multiplicación de los elementos renglón-columna de las matrices A y B conforma la región paralela que es posible paralelizar. De acuerdo a la definición matemática de la multiplicación de matrices se define de la siguiente manera:

$$C_{ij} = \ a_{i1}b_{1j} + \ a_{i2}b_{2j} + \cdots + \ a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

Donde i = 1, …, m y j = 1, …, p

Cada una de estas multiplicaciones es un proceso independiente y es posible hacer la paralelización. Debido a que las dimensiones de las matrices coinciden se puede realizar la multiplicación. Tomando en cuenta las dimensiones renglón de la primera matriz y columna de la segunda la matriz resultante tendrá también las mismas dimensiones. Esta multiplicación generara 4 multiplicaciones renglón-columna entre los renglones de la primera matriz con la columna de la segunda donde cada una de estas operaciones se puede considerar como un proceso independiente.

$$\begin{pmatrix} 4 & 7 \\ 10 & 2 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 4 & 7 \\ 10 & 2 \end{pmatrix}$$

A continuación, se muestran las multiplicaciones renglón-columna entre los elementos de las matrices. Cada una de estas multiplicaciones son procesos independientes, por lo que esta operación puede ser paralelizable por medio de técnicas de cómputo de alto rendimiento.

$$(4 \quad 7)\begin{pmatrix} 1 \\ 0 \end{pmatrix} = (4)(1) + (7)(0) = 4$$

$$(4 \quad 7)\begin{pmatrix} 0 \\ 1 \end{pmatrix} = (4)(0) + (7)(1) = 7$$

$$(10 \quad 2)\begin{pmatrix}1\\0\end{pmatrix} = (10)(1) + (2)(0) = 10$$

$$(10 \quad 2)\begin{pmatrix}0\\1\end{pmatrix} = (10)(0) + (2)(1) = 2$$

Estas multiplicaciones renglón – columna se calculan por medio de un ciclo utilizado para recorrer los elementos de la multiplicación. Es posible utilizar también una variable auxiliar para esta multiplicación y una vez que termine el ciclo agregarlo a la variable resultado.

$$resultado[i][j] += int(kernel[i][k] * filtro\_imagen[k][j])$$

Consideraciones al momento de aplicar la operación de Max Pooling.

Esta operación reduce considerablemente la imagen original dejando que solamente una pequeña parte de los valores originales de la imagen conforman el mapa de características después de aplicar Max Pooling. En caso de una imagen de 4x4 de 16 elementos con los hiperparámetros de tamaño de kernel o ventana de 2, desplazamiento (stride) de 2 y padding de 0, al aplicar Max Pooling la imagen queda de 2x2 y solo quedan 4 elementos en el mapa de características, es decir se reduce un 75% en el número de elementos que conforman el mapa de características respecto a la imagen previo a aplicar la operación y su tamaño se reduce a la mitad en tanto el eje x como el eje y.

En el momento de hacer la operación de Max Pooling hay un número de kernels extraídos. Dicho número depende del ajuste de hiperparametros antes de aplicar la operación. Por cada ventana extraído se realiza una transferencia de datos entre CPU y GPU donde se envia el kernel obtenido a partir de la imagen original, una vez obtenida la multiplicación de matrices se envía el resultado de vuelta al CPU para agregar el valor máximo en el mapa de características que conforma la operación de Max Pooling.

Programa de multiplicación de matrices

Este programa en base a los principios previamente mencionados lanza un kernel GPU para resolver esta operación considerada computacionalmente intensiva siendo este proceso de multiplicación de matrices la parte que es la región paralelizable del programa.

A continuación, se muestra una serie de experimentos de la multiplicación de matrices en la GPU donde el kernel GPU se conforma de dos ciclos for para recorrer las matrices y el tercer ciclo para poder realizar las multiplicaciones renglón-columna.

Esta multiplicación de matrices toma en cuenta las características que se van a utilizar para obtener la operación de Max Pooling realizando la multiplicación de una matriz aleatoria con la matriz identidad la cual funciona en la multiplicación como elemento neutro. La variable resultado guarda este valor de multiplicación y se compara con una función preestablecida por Python para la verificación de resultados. Para la definición del kernel solamente se aceptan funciones y métodos del lenguaje de Python por lo que la creación de arreglos y métodos de la librería de Numpy no están soportados en la declaración del kernel GPU.

El uso de los índices se encuentra en el kernel GPU de la multiplicación de matrices. Esta variable muestra los índices de la multiplicación de matrices para ver el comportamiento interno de los índices de la GPU los cuales se obtiene a partir de la siguiente instrucción la cual es equivalente a las siguientes variables de la GPU definidas en la sintaxis de Numba CUDA para el acceso a los recursos de la GPU. Estas pruebas se realizaron el 20 de febrero de 2023.

```
I, j = cuda.grid(2)     # selección de índices para un problema en 2D
```

Esta sintaxis de Numba CUDA es equivalente a la siguiente instrucción para obtener los valores de los índices.

```
I = threadIdx.x + blockDim.x * blockIdx.x

J = threadIdx.y + blockDim.y * blockIdx.y
```

EXPERIMENTO MULT MATRICES TESLA T4		
# EXPERIMENTO	TAM. ENTRADA	TIEMPO SEG
1	2	0.4045
2	4	0.6157
3	8	0.6978
4	16	0.4045
5	32	0.6938
6	64	0.5523
7	128	0.6429
8	256	0.6565
9	512	2.909
10	1024	40.8932
11	2048	908.9013

FIGURA 4. EXPERIMENTO MULT. MATRICES TESLA T4

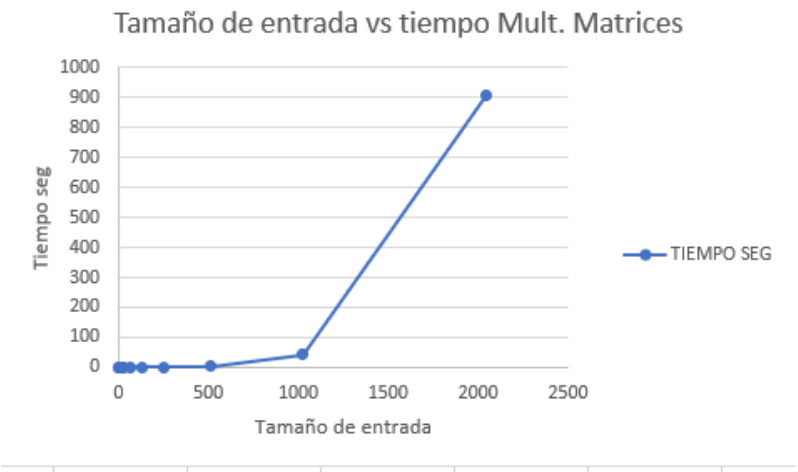

FIGURA 4 TAMAÑO DE ENTRADA VS TIEMPO MULT. MATRICES

```
================================

Ingresar la dimension de la multiplicacion : 4
kernel =
[7 8 2 0]
[7 4 4 4]
[4 9 0 2]
[1 1 2 1]
(4, 4)

filtro imagen =
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
(4, 4)

gridsize  (1, 1)
blocksize  (16, 16)
========================================
Numero total de hilos x =  16
Numero total de hilos y =  16
========================================
----INDICES---#
[[1.  1.1 1.2 1.3]
 [1.  1.1 1.6 1.7]
 [1.  1.1 1.6 1.7]
 [1.  1.1 1.6 1.7]]
(4, 4)
16
-------#
resultado
[7 8 2 0]
[7 4 4 4]
[4 9 0 2]
[1 1 2 1]
(4, 4)
int64
```

FIGURA 5 PROGRAMA DE MULT. MATRICES EN TESLA T4 PARTE 1

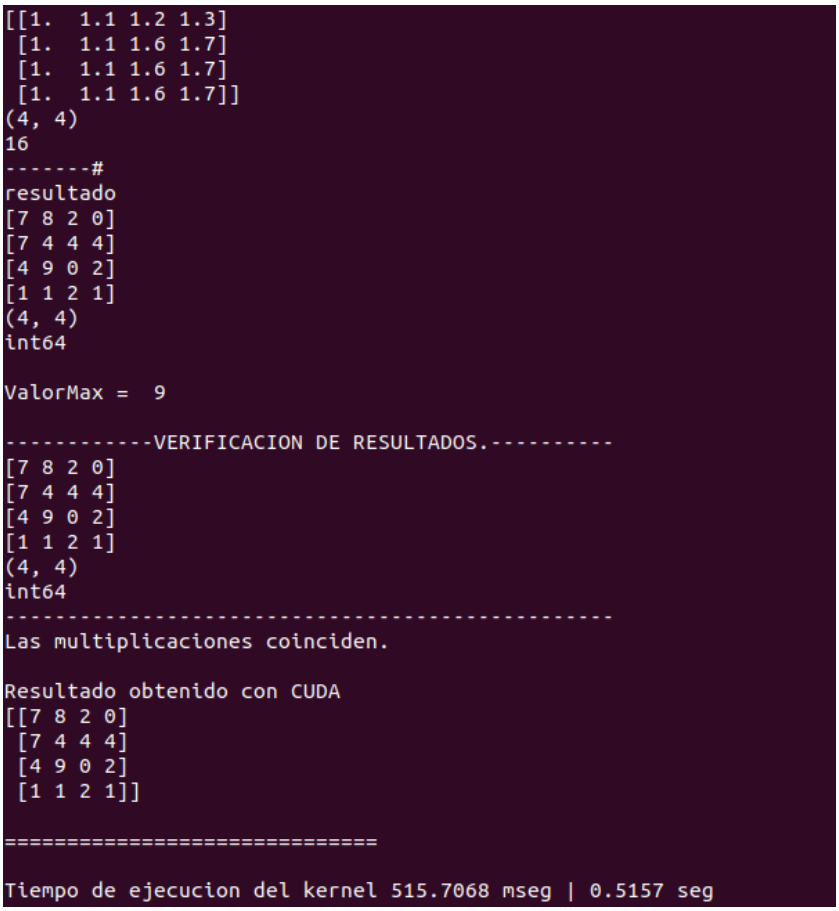

FIGURA 6. PROGRAMA DE MULT. MATRICES EN TESLA T4 PARTE 2

Cálculo del tamaño de salida del mapa de características.

Para el cálculo del tamaño de la salida del mapa de características por medio de la operación de Max Pooling se utiliza la siguiente fórmula.

$$n_{salida} = \left\lfloor \frac{n_{entrada} + 2p - k}{s} \right\rfloor + 1$$

Donde

$n_{entrada}$= Número de características de entrada

n_{salida} = Número de características de salida

K = tamaño del kernel de convolución

P = tamaño del padding de convolución

S = tamaño del desplazamiento (stride) de convolución

AJUSTE DE HIPERPARAMETROS

Con los siguientes hiper parámetros se obtiene la siguiente salida donde el mapa de características de la imagen final se reduce a la mitad. Se recomienda trabajar con matrices de orden par para facilitar la verificación durante los experimentos de la operación de Max Pooling.

- P = 0
- K = 2
- S = 2

$$n_{salida} = \left\lfloor \frac{n_{entrada} + 2(0) - 2}{2} \right\rfloor + 1$$

$$n_{salida} = \left\lfloor \frac{n_{entrada} + 0 - 2}{2} \right\rfloor + 1$$

$$n_{salida} = \left\lfloor \frac{n_{entrada}}{2} - \frac{2}{2} \right\rfloor + 1$$

$$n_{salida} = \left\lfloor \frac{n_{entrada}}{2} - 1 \right\rfloor + 1$$

$$n_{salida} = \frac{n_{entrada}}{2}$$

En el programa de Max Pooling el mapa de características se representa por medio de la variable imagen_Final la cual es una lista declarada como global. Esta variable ira acumulando los valores obtenidos a partir de la operación de Max Pooling. Al terminar el proceso de extracción de kernel y multiplicación de matrices se ajustarán las dimensiones de acuerdo a los hiperparámetros establecidos.

Características aleatorias o no supervisadas.

La parte más difícil de entrenar una red convolucional es aprender las características. La capa de salida es poco costosa debido al pequeño número de características como entrada de esta capa pasando a través de muchas capas o pooling.

De la misma forma que con los perceptrones multicapa, se usa pre entrenamiento codicioso de capas codiciosas, para entrenar la primera capa en soledad, entonces extrae todas las características desde la primera capa sólo una vez, entonces se entrena la segunda capa y así sucesivamente. Es posible usar entrenamiento no supervisado para entrenar una red convolucional sin usar la convolución durante el proceso de entrenamiento. Para más información referirse a (Agrawall, 2019)

Generalización de aprendizaje profundo

En términos prácticos el overfitting se refiere a que una red neuronal provea rendimiento de predicción sobre el conjunto de entrenamiento, pero no se desempeñara bien en datos para los cuales no fue entrenado. Las formas extremas de overfitting se refieren como memorización.

Varianza del modelo

La diferencia en las predicciones para la misma instancia de prueba recibe este nombre. Los modelos con alta varianza tienden a memorizar artefactos aleatorios del conjunto de entrenamiento provocando inconsistencias y falta de precisión en la predicción de instancias de prueba no vistas. El overfitting depende tanto de la complejidad del modelo y la cantidad de datos que se tengan disponibles. La complejidad del modelo se define por una red neuronal sobre el número de parámetros delineados

Métodos de continuación

Estos métodos consisten en entrenar modelos más simples para posteriormente hacerlos más complejo para así evitar al principio el overfitting. En algunos dominios de datos tales como los casos de texto o imágenes, se tiene una idea del espacio de parámetros. Algunos de los parámetros en diferentes partes de la red pueden establecerse a algún valor para reducir los grados de libertad del modelo. Dado un conjunto de datos etiquetados, se necesita usar este recurso para el entrenamiento, tuneo y prueba de la precisión del modelo. No se puede usar el recurso entero de los datos etiquetados para la construcción del modelo. La meta principal de la clasificación es generalizar el modelo de datos etiquetados a instancias de pruebas no vistas. Un error es usar los mismos conjuntos de datos tanto para el tuneo de parámetros y evaluación final para evaluación.

Un conjunto de datos siempre debe estar dividido en 3 partes definidas a través de la forma en la cual los datos son utilizados.

1.-Datos de entrenamiento

2.-Datos de validación.

3.-Datos de prueba.

Validación cruzada

En este método los datos etiquetados se dividen en q segmentos iguales.

Problemas con el entrenamiento a escala.

Surge un problema cuando el tamaño del conjunto de datos es grande. El tiempo de entrenamiento es una consideración importante en el modelado de redes neuronales que muchos compromisos tienen que hacer para habilitar implementación práctica.

Las redes neuronales son difíciles de entrenar y un error grande puede ser causado por el comportamiento de convergencia débil del algoritmo. El error puede ser causado por un alto bias, lo cual se refiere como underfitting. El overfitting puede llegar a causar una gran parte del error de generalización. El overfitting es manifestado por una brecha grande entre la precisión de prueba y de entrenamiento. La sobre simplificación reduce el poder expresivo de una red neuronal, y que no es posible ajustar lo suficiente las necesidades de diferentes tipos de conjuntos de datos.

Validación de modelo

Se pueden ajustar los parámetros para rendimiento optimo sobre una porción del conjunto de datos que no se usa para aprender los parámetros. Para cualquier peso dado w1 en una red neuronal, las actualizaciones son definidas por medio del gradiente descendiente.

Características de las redes convolucionales

Estas redes se diseñan para poder trabajar con entradas estructurada en rejillas. Otra forma de datos secuenciales tales como texto, series de tiempo y secuencias. La vasta mayoría de aplicaciones de las redes convolucionales se ajustan sobre imágenes de datos, algunos pueden usar estas redes de todos los tipos de datos.

Las redes convolucionales tienden a crear valores de características similares desde regiones locales con patrones similares. La operación de convolución es una operación de producto punto entre un conjunto de pesos estructurada en rejilla hechas a partir de las locaciones espaciales en el volumen de entrada.

Los 3 tipos de capas que son presentes en una red convolucional son la convolución, pooling y las funciones de activación tales como ReLU. En la red convolucional, los parámetros son organizados en conjuntos de unidades de estructuras de 3 dimensiones conocidos como filtros o kernels. La operación de convolución ajusta los filtros de cada posible posición en la imagen por lo que el filtro sobrepasa con la imagen y realiza el producto punto entre los parámetros en el filtro y la rejilla en el volumen de entrada. El producto punto se realiza tratando sus entradas en la región relevante de 3 dimensiones del volumen de entrada y el filtro como vectores, por lo que los elementos basados en ambos vectores están basados en sus posiciones correspondientes en el volumen de estructura de rejilla.

Mapa de características

Los conjuntos de características arregladas de forma espacial obtenidas a partir de la salida de un solo filtro. El número de filtros usado en cada capa controla la capacidad del modelo debido a que directamente controla el número de parámetros. Al incrementar el número de filtros en una capa particular incrementa el número del mapa de características. Por lo general las capas posteriores tienden a tener una huella espacial menor, pero mayor profundidad en términos del número del mapa de características.

Padding

La operación de convolución reduce el tamaño de la (q+1)-capa en comparación con el tamaño de la q-capa. Este tipo de reducción tiende a perder información sobre los bordes de las imágenes. La técnica de padding consiste en agregar (Fq - 1) / 2 pixeles sobre los bordes de los mapas de características para mantener la huella espacial. El valor de cada uno de estos valores de características es establecido a 0, respecto a que la entrada o las capas ocultas se les aplique el padding. Esta técnica se aplica después de que se aplica la convolución.

El padding permite la operación de convolución con una porción del filtro desde los bordes de la capa y ejecutar el producto punto solamente sobre la porción de la capa donde los valores están definidos. En el caso de Max Pooling el parámetro de padding se recomienda que este parámetro sea un múltiplo del valor de la dimensión de la imagen de entrada para la que se pueda obtener una mayor extracción de características reduciendo así la perdida de información de la imagen de entrada en las orillas del mapa de características.

Capa ReLU

Las funciones de activación de ReLU son aplicadas a crear valores de umbral y estos valores pasan a la siguiente capa. Una función ReLU por lo general sigue una operación de convolución. Aplicar ReLU no cambia las dimensiones de una capa debido a que es un mapeo de uno a uno de los valores de activación. La función ReLU es relativamente reciente sobre todo con la llegada de AlexNet mientras que otras funciones tales como sigmoid y tanh se han usado en ocasiones anteriores. Las ventajas de ReLU en velocidad y precisión ha sido la razón por la cual ha sido la elección en las funciones de

activación en arquitecturas posteriores. Esta operación trabaja sobre regiones de rejilla de cierto tamaño en cada capa.

Tipos de arquitecturas o modelos convolucionales

LeNet

La arquitectura LeNet es una red neuronal convolucional utilizada para la clasificación de imágenes. El entrenamiento de la red se hizo por medio de la base de datos MNIST la cual consiste de una gran cantidad de imágenes de dígitos. Este conjunto de imágenes consiste de 60,000 ejemplos y un conjunto de prueba de 10,000 ejemplos de imágenes en blanco y negro de números que van desde el 0 hasta el 9. Estas imágenes tienen una dimensión de 28x28.

Esta red hace en su primer parte la extracción de características y posteriormente hace la clasificación de imágenes. La extracción de características se hace por medio de una serie de capas usando ReLU como la función de activación. La parte de clasificación se hace por medio de un perceptrón multicapa.

AlexNet

La red AlexNet fue la arquitectura ganadora de la competencia ILSVRC en 2012 debido a sus avances y contribuciones en el uso de funciones de activación, arquitectura y porcentaje de error obtenido a partir del conjunto de datos de entrenamiento.

La red consiste de 8 capas y se divide en una etapa de extracción de características y otra etapa de clasificación posteriormente. Las primeras 5 capas son convolucionales y las últimas 3 capas son capas completamente conectadas cada una con Dropout y una capa de softmax para la predicción. AlexNet usa ReLU en vez del tangente hiperbólico como función de activación para poder conseguir un mejor tiempo de entrenamiento. En este modelo o arquitectura el número de canales va aumentando después de la primera convolución. Las capas completamente conectadas hacen la clasificación.

Para más información referirse a (Agrawall, 2019)

GPUs Y SU FUNCIÓN COMO ACELERADOR DE ELEMENTOS COMPUTACIONALMENTE INTENSIVOS

Las GPUs se consideran aceleradores debido a que son capaces de poder resolver operaciones de algebra lineal de forma eficiente. Estas operaciones que se presentaban en aplicaciones de gráficos y videojuegos como el uso original de las GPUs al realizar multiplicaciones de matrices en paralelo para el renderizado de gráficos. Estas operaciones son muy usadas en aplicaciones de inteligencia artificial tales como aprendizaje de máquina y aprendizaje profundo. Debido a que las multiplicaciones de matrices, así como otras operaciones consideradas computacionalmente intensivas aparecen durante la propagación hacia adelante.

En el modelo SIMT (Single Instruction Multiple Threads) los hilos ejecutan el mismo código de forma concurrente ofreciendo un grado alto de paralelismo. Esta característica permite a las GPUs realizar tareas repetitivas que se resuelvan por medio de ciclos de forma eficiente, aunque su ciclo de reloj sea más bajo que el de la CPU. Los hilos se agrupan en unidades llamadas warps los cuales son grupos de 32 donde cada warp trabaja con el mismo código. El ancho de banda es la velocidad de acceso en la cual los procesadores puede accesar a localidades de memoria.

ARQUITECTURA SIMT

El modelo SIMT se basa en una serie de Streaming Multiprocessors (SMs) en forma de arreglo, donde cada uno de estos multiprocesadores puede ejecutar y administrar los hilos de forma concurrente. Los multiprocesos ejecutan los hilos en grupos de 32 llamados warps, siendo medio warp 16 y un cuarto de warp 8 como unidad de medida de los procesos lanzados por la GPU. Cada warp trabaja con una ejecución de código a la vez. Cada hilo es independiente debido a que tiene cada uno sus propios registros y contador de direcciones.

El número de los warps en grupos de 32 hilos influye en el rendimiento de los programas de la GPU y viene preestablecido en el hardware. Se recomienda la configuración de este parámetro en múltiplos de 32 para obtener una mayor eficiencia y evitar la divergencia de warp en caso de que instrucciones condicionales estén presentes en el kernel de la GPU o función de dispositivo.

Viendo la correspondencia entre el software y el hardware de la GPU podemos asociar los componentes que conforman la jerarquía de memoria de la siguiente manera.

- Hilo.
  Unidades independientes con su propios registro y memoria, siendo estos registros los más rápidos dentro de la jerarquía de memoria de la GPU. Cada uno corresponde a un núcleo CUDA.

- Bloque.
  Agrupan los hilos en warps. Un número de bloques conforman un SM (Streaming Multiprocessor). Los núcleos CUDA se agrupan por medio de SM. Los hilos dentro de estos bloques tienen acceso a la memoria compartida.

- Grid
  Esta parte de la GPU conforma un conjunto de bloques. Cada uno de estos bloques conforman el arreglo de SM que se refiere al device. La memoria global puede ser accesada por todos los SMs del dispositivo.

Jerarquía de memoria

Cada hilo cuenta con sus propios registros de estado y contador de memoria siendo este la unidad de cómputo y memoria básica dentro de la jerarquía de memoria de la GPU.

A continuación, se muestran los índices para problemas en 2D los cuales deben estar presentes en la sintaxis de CUDA.

i = threadIdx.x + blockIdx.x * blockDim.x

J = threadIdx.y + blockIdx.y * blockDim.y

Para más información referirse a (Hwu, 2013)

Capacidad de cómputo

Se refiere a la generación o arquitectura a la que pertenece la GPU además de las características que tiene un modelo en particular. Se tiene un número entero como el valor mayor que representa la generación o arquitectura y un número menor después del punto decimal que representa las características de la GPU.

Para más información referirse a (Cheng, 2015)

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

Contexto CUDA

Para referirse a los procesos presentes en el CPU estos se refieren como contextos CUDA, en el cual las acciones a ejecutar y los recursos asignados de la GPU se encuentran dentro de este contexto, donde cada contexto tiene su propia dirección de memoria. Los contextos se acumulan en una pila de contextos controlados por un hilo de host.

Aplicaciones de las GPUs

Las GPUs al considerarse aceleradores de código que puede ser paralelizable en áreas como el manejo de ciclos puede tener aplicaciones en diferentes áreas tanto en la industria como en la investigación. A continuación, se mencionan alguna de las aplicaciones que puede utilizarse las técnicas de CAR por medio de GPUs.

- CAR, Cómputo numérico y científico
- Inteligencia artificial
- Ciencia de datos

Partición de bloques

Cada hilo toma solamente una porción de los datos a los procesos. En el caso de la partición cíclica, cada hilo toma más de una porción de los datos a los procesos.

Las GPUs tienen una arquitectura llamada SIMT (Single Instruction Multiple Thread) esta arquitectura está basada en SIMD y MIMD. Dentro de la taxonomía de Flynn los programas que se ejecutan en GPU se acercan más a SPMD, aunque esta arquitectura se refiere a programas de forma serial.

Actualmente en el momento de la elaboración del presente trabajo se busca establecer un enfoque de cómputo heterogéneo para los sistemas de cómputo para combinar las características tanto del CPU como del GPU. En el caso del CPU su diseño permite resolver gran cantidad de operaciones lógicas. Tal es el caso de la lógica que acompaña la operación de los sistemas operativos y muchos programas de uso cotidiano tales como programas de ofimática, calendarios, buscadores web entre otros programas de tareas generales.

En el caso de las GPU se componen de una gran cantidad de núcleos de procesos ligeros los cuales permiten realizar tareas que tienen paralelismo de tarea con menos lógica presente en su código. Esta característica permite la aceleración de secciones de código que puedan ser candidatos a ser paralelizables, lo cual se conoce como tareas embarazosamente paralelas las cuales por lo general se componen de múltiples ciclos anidados. Recordemos que el uso de ciclos anidados se considera una tarea computacionalmente intensiva donde aumenta el grado de complejidad por cada ciclo anidado que se va agregando.

Desde el punto de vista de instrucciones básicas de programación en el lenguaje C el CPU puede ejecutar más rápidamente instrucciones lógicas tales como las operaciones lógicas tales como if, if-else y switch bajo un enfoque serial. Actualmente en enfoque paralelo lleva a que los procesadores tengan una mayor cantidad de núcleos. Para el caso de las GPUs permiten una mayor velocidad en las operaciones de ciclos tales como for, do… while y while bajo un enfoque paralelo entre la cantidad de núcleos de la GPU disponibles. La GPU también puede realizar cálculos de operaciones lógicas, pero es posible que incluso puede llegar a ser más tardado que la CPU para este tipo de tareas.

Originalmente las GPUs se desarrollaron con el objetivo de hacer cálculos de gráficos en paralelo, sin embargo, han ido buscando desarrollar una interfaz de programación que no requiere de librerías graficas para su uso y resolver problemas de tipo más general tal como aplicaciones de ciencia e ingeniería.

La CPU y las GPU se conectan por medio de un bus PCI en un solo nodo o en el caso de la computación personal. Actualmente se busca que ambos procesadores conformen un sistema de arquitectura heterogénea donde el CPU recibe el nombre de host y la GPU recibe el nombre de device. El CPU se encarga de la ejecución del código, así como la configuración de los ambientes y las herramientas de desarrollo, mientras que las partes del código que son paralelizables o se consideran computacionalmente intensivas se envían a la GPU. Para el caso de las GPUs se clasifican en capacidad de cómputo las cual es un acrónimo que se refiere a la generación de cada tarjeta la cual va agregando características nuevas. Las partes que sean computacionalmente intensivas se pueden enviar a la GPU por medio de una función de kernel. Se debe de tomar en cuenta que estos programas pueden llegar a tener retrasos por la transferencia entre el CPU y la GPU para aplicaciones computacionalmente intensivas en un kernel de CUDA. Para el caso de que una vez que el kernel sea lanzado en la GPU y termine su tarea el control se regresa de nuevo al CPU para continuar con la operación del programa.

Para el uso de la GPU se debe de tener en cuenta la jerarquía de memoria de las GPUs. A nivel de bloque se tiene una memoria compartida que comparten los hilos en ese bloque mientras que la memoria global conocida como DRAM la comparten todos los bloques que conforman la tarjeta.

Los registros al estar dentro de los núcleos de la GPU tienen una latencia muy baja y un mayor ancho de banda y no consume memoria global de parte de la DRAM.

Estos registros de alta velocidad se encuentran en la memoria del chip. Estos registros están a nivel de hilos donde cada hilo tiene sus propios registros. La memoria compartida es a nivel de bloque, los cuales están conformados por hilos. Los registros podemos hacer una similitud con el registro de archivos del modelo de Von Neumann. El número de registros en la GPU se debe tomar en cuenta que es un recurso limitado de la GPU. En el caso de las variables que estén en la memoria compartida estás pueden tener acceso a todos los hilos en ese bloque a una velocidad muy alta y de forma paralela en su bloque correspondiente.

Por lo general los SM (Streaming Multiprocesors) tienen varios bloques que lo conforman. El tamaño de la memoria en cada SM puede variar entre cada modelo de GPU. Los registros, memoria compartida y memoria constante se pueden acceder a una mayor velocidad y de forma paralela que la memoria global.

Conceptos de cómputo de alto rendimiento

Desde 2002 cuando los procesadores adoptaron por un diseño paralelo debido a las limitaciones en la frecuencia de reloj y la dificultad de agregar una mayor cantidad de transistores en el diseño del procesador se buscó una forma de poder aprovechar la disponibilidad de múltiples núcleos los cuales pueden manejar distintos procesos.

Para poder aprovechar la disponibilidad de procesadores multinúcleo o más se debe cambiar de los programas seriales a las técnicas paralelas. Para el caso del cómputo de alto rendimiento se tiene la posibilidad de aplicar principalmente dos técnicas que están disponibles las cuales son memoria compartida y memoria distribuida.

Dentro de las principales aplicaciones del cómputo de alto rendimiento tenemos

- Modelado de clima
- Proteínas plegables
- Descubrimiento de nuevos medicamentos y tratamientos médicos.
- Investigación de energía.
- Análisis de datos

En el caso de memoria distribuida se tiene el uso por medio de paso de mensajes por medio de MPI. Para el caso de memoria compartida los núcleos comparten acceso a la memoria de la computadora en el que cada núcleo lee y escribe cada locación de memoria. En el caso de memoria compartida se utilizan los hilos y OpenMP para el uso de memoria compartida.

Arquitectura Von Neuman

La arquitectura Von Neumann está conformada de la memoria principal, la unidad central de procesamiento y un camino de interconexión entre la memoria y el CPU. La memoria principal

consiste en una colección de localidades de memoria cada una con la posibilidad de guardar tanto instrucciones como datos.

Los datos en el CPU sobre el estado del programa se almacenan en una memoria de alta velocidad de respuesta conocido como registros. La unidad de control tiene una unidad de registro llamada contador de programa la cual direcciona la siguiente instrucción que debe de ser ejecutada. En el momento en que las instrucciones y losa datos se transfieren desde la memoria al CPU se dice que las instrucciones o datos son leídos desde la memoria. La separación entre el CPU y la memoria recibe el nombre de cuello de botella Von Neumann, desde que la interconexión determina la tasa en la cual las instrucciones y los datos pueden tener acceso.

Previo a la selección de un método de paralelización se debe evaluar si el problema es paralelizable identificando la región paralela del problema, así como la revisión de buscar que no haya dependencia de procesos. Una vez verificada las evaluaciones anteriores se puede seleccionar un método de paralelización los cuales se dividen principalmente en dos grupos los cuales son conocidos como memoria compartida y memoria distribuida.

Los hilos son un método de memoria compartida que permiten dividir las tareas en tareas independientes con la propiedad de que un hilo se bloquea otro hilo lo puede correr. Los hilos están contenidos dentro de los procesos, por lo que pueden usar los mismos ejecutables y por lo general compartir memoria y los mismos dispositivos de E/S. Los sistemas SIMD (Single Instruction Multiple Data) son sistemas paralelos. Estos sistemas operan sobre múltiples flujos de datos aplicando la misma instrucción sobre múltiples instancias de datos.

Para más información referirse a (Stallings, 2005)

LEY DE AMDAHL

Esta fórmula permite obtener el Speedup que se puede obtener a partir de una región paralela cuando se agregan un cierto número de procesadores. No todo el programa puede ser paralelizable por lo que se debe de identificar una región paralela y como esta contribuye al Speedup durante la ejecución del programa. El Speedup se limita debido a la parte serial del programa. A continuación, se muestra la demostración de la Ley de Amdahl.

Demostración de la Ley de Amdahl

$$T_m = T_a\left((1 - F_m) + \frac{F_m}{A_m}\right)$$

$$T_m = T_a - T_a F_m + \frac{F_m T_a}{A_m}$$

$$\frac{F_m T_a}{A_m} = T_a - T_a F_m - T_m$$

$$1 = \frac{T_a}{T_m} - \frac{T_a F_m}{T_m} + \frac{\frac{F_m T_a}{A_m}}{\frac{T_m}{1}}$$

$$Realizando\ la\ siguiente\ sustitución\ \ A = \frac{T_a}{T_m}\ \ tenemos\ la\ siguiente\ ecuación.$$

$$1 = A - AF_m + \frac{F_m}{A_m}A$$

$$1 = A\left(1 - F_m + \frac{F_m}{A_m}\right)$$

$$A = \frac{1}{(1 - F_m) + \frac{F_m}{A_m}}$$

Cálculo de Speedup

$$Speedup = \frac{1}{0.38 + \frac{0.62}{n_{GPU}}}$$

Siendo n_{GPU} el número de núcleos por cada una de las tarjetas tenemos los siguientes datos

GTX 1050 = 640 núcleos

Tesla K20 = 2496 núcleos

Tesla T4 = 2560 núcleos

$$p = \frac{0.5267}{0.8604} = 0.61 \quad \rightarrow \quad 61.21\%\ \ del\ programa\ es\ paralelizable$$

$$Speedup = \frac{1}{0.39 + \frac{0.61}{640}} = 2.5578$$

$$Speedup = \frac{1}{0.39 + \frac{0.61}{2496}} = 2.5624$$

$$Speedup = \frac{1}{0.39 + \frac{0.61}{2560}} = 2.5625$$

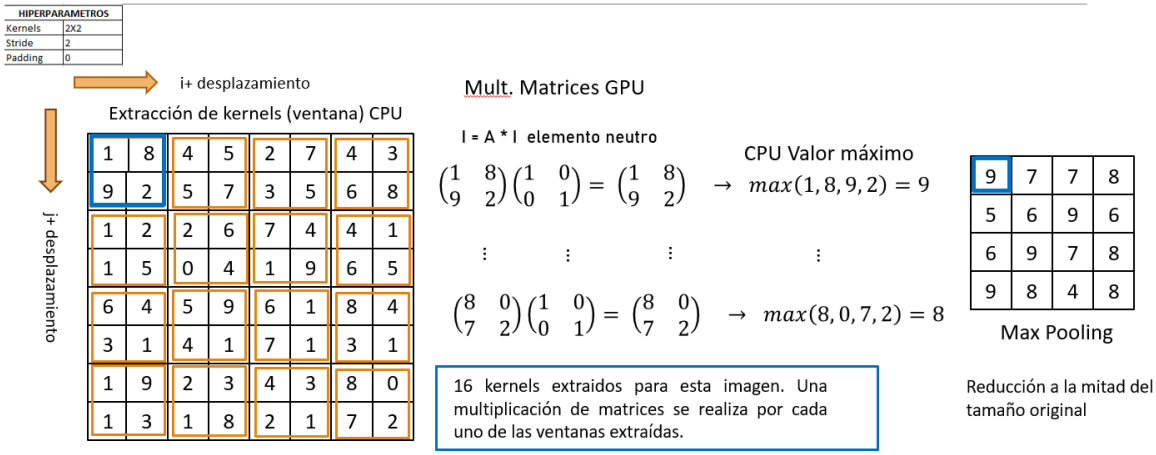

FIGURA 7. PROCESO DE MAX POOLING COMO SISTEMA HETEROGENEO CPU + GPU

La siguiente imagen muestra de forma resumida el proceso de Max Pooling como un sistema heterogeneo de la extracción de kernels por medio de CPU y la multiplicación de matrices como proceso computacionalmente intensivo obtenido por medio de la GPU y su posterior comunicación de regreso al CPU para conformar el mapa de características que generara la imagen de Max Pooling de las dimensiones de la mitad del tamaño de la imagen original por las propiedades de los hiperparametros previamente mencionadas.

Pruebas de entrenamiento en CPU y CUDA

Las pruebas de entrenamiento en CPU se realizaron en una computadora Windows 10 con procesador Core i7, mientras que las pruebas en GPU se realizaron en Windows 10 con una GPU NVIDIA GTX 1050.

Las GPU están optimizadas para el procesamiento de datos paralela con instrucciones de repetición en la cual no hay instrucciones de decisión. La GPU no es una plataforma independiente, sino que trabaja en conjunto con el CPU a través de un bus PIC-Express.

El CPU administra el ambiente, el código y los datos para el dispositivo antes de poder cargar tareas de cómputo intensivo sobre el dispositivo. Las GPUs permiten acelerar secciones de código intensivo de instrucciones que contienen ciclos y sin tener dependencia de datos. Estas características

muestran en las secciones de código un alto grado de paralelismo. Conforme el tamaño de los datos aumenta y se aumenta el grado de paralelismo tienden a ser problemas más tratables con la GPU.

Fechas de experimentación ETAPA 1

Las pruebas con CPU se realizaron el viernes 26 de agosto de 2022 en ambiente local en una computadora con Windows 10, procesador Intel Core i-7 de 4 núcleos y 8 procesos y tarjeta NVIDIA GTX 1050 de 4Gb de memoria. Las pruebas con GPU se realizaron el 10 de septiembre de 2022 en la misma computadora con CUDA habilitado en ambiente Windows 10. El programa se realizó con PyTorch 1.10.2 y Python 3.8.13 utilizando la herramienta Anaconda y Jupyter Notebook. Se recomienda el uso de Interact como herramienta para visualizar el contenido de los notebooks en formato ypnb en caso de que solamente se quiera consultar estos códigos sin abrirlos en un ambiente con su configuración correspondiente de librerías para su ejecución.

Hilos de la GPU

En la GPU los hilos son ligeros y pueden cambiar de hilos paralelizados a otros hilos sin costos en cada ciclo de reloj. Existe otra opción para mejorar el rendimiento del entrenamiento y una convergencia en menos épocas con los optimizadores. Actualmente el optimizador más utilizado es Adam. A continuación, se muestran los resultados de las pruebas realizadas en CPU y en GPU. Las pruebas se realizaron con una red AlexNet con modificaciones para el entrenamiento por medio del conjunto de imágenes de datos de imágenes cerebrales y librerías para el diagnóstico para facilitar el manejo del conjunto de imágenes con el propósito de familiarizarse con el funcionamiento de las redes convolucionales.

Las capas de pooling presentes en esta arquitectura reducen el tamaño de los mapas de características acelerando el procesamiento de la red debido a la reducción de los mapas de características del conjunto de entrenamiento de la red al mismo tiempo que se extraen las características.

Métricas de desempeño en GPU

```
print(f'Numero de ejemplos de entrenamiento: {len(datos_entrenamiento)}')
print(f'Numero de ejemplos de validacion: {len(valid_data)}')
print(f'Numero de ejemplos de prueba: {len(datos_prueba)}')
```

```
Numero de ejemplos de entrenamiento: 245
Numero de ejemplos de validacion: 28
Numero de ejemplos de prueba: 99
```

FIGURA 8. NÚMERO DE EJEMPLOS DE LA RED CONVOLUCIONAL

Se apartaron 32 imágenes del conjunto de imágenes cerebrales para cada una de las 3 clases como conjunto de prueba y el resto para el conjunto de entrenamiento. Las pruebas se realizaron el 10 de septiembre de 2022 en la misma computadora con CUDA habilitado en ambiente Windows 10.

```
plot_lr_finder(lrs, losses)
```

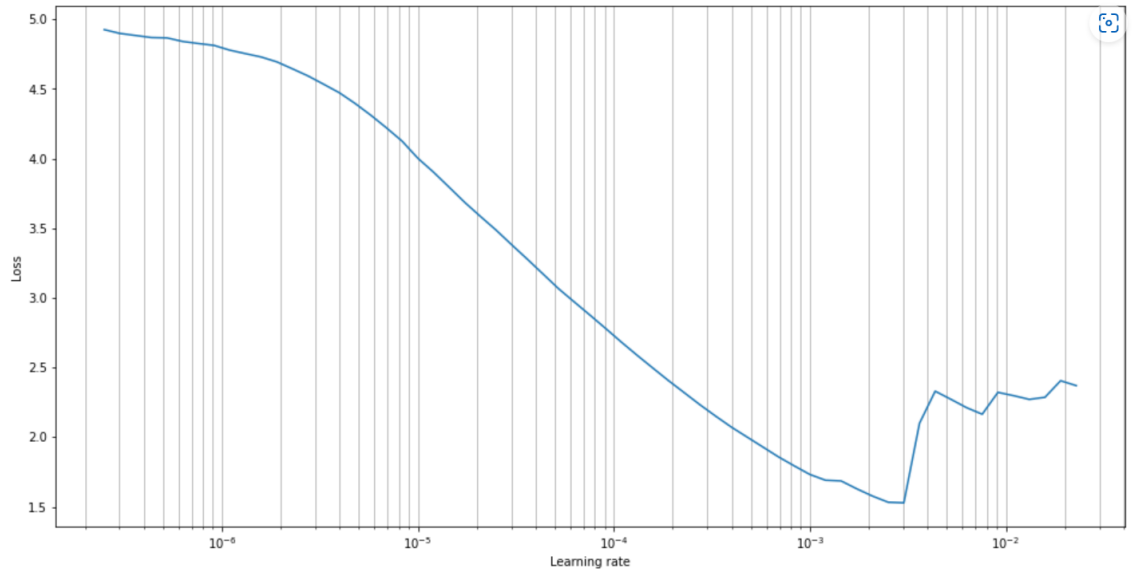

FIGURA 9. TAZA DE APRENDIZAJE VS PERDIDA

```
model.load_state_dict(torch.load('tut3-model.pt'))

test_loss, test_acc = evaluate(model, iterador_prueba, criterion, device)

print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.303 | Test Acc: 87.88%
```

FIGURA 10. RESULTADOS DE PRUEBAS DE ENTRENAMIENTO

RESULTADOS DE PRUEBAS DE ENTRENAMIENTO

Para el entrenamiento de 25 épocas se obtuvo una precisión de 87.88% en un tiempo de 24.87 segundos en una computadora Intel Core i7 en ambiente Windows 10. Para el entrenamiento de 50 épocas se obtuvo una precisión de 94.95% en un tiempo de 54.31 segundos en una computadora Intel Core i7 en ambiente Windows 10. Las pruebas se realizaron el 10 de septiembre de 2022 en la misma computadora con CUDA habilitado en ambiente Windows 10.

```
plot_confusion_matrix(labels, pred_labels, classes)
```

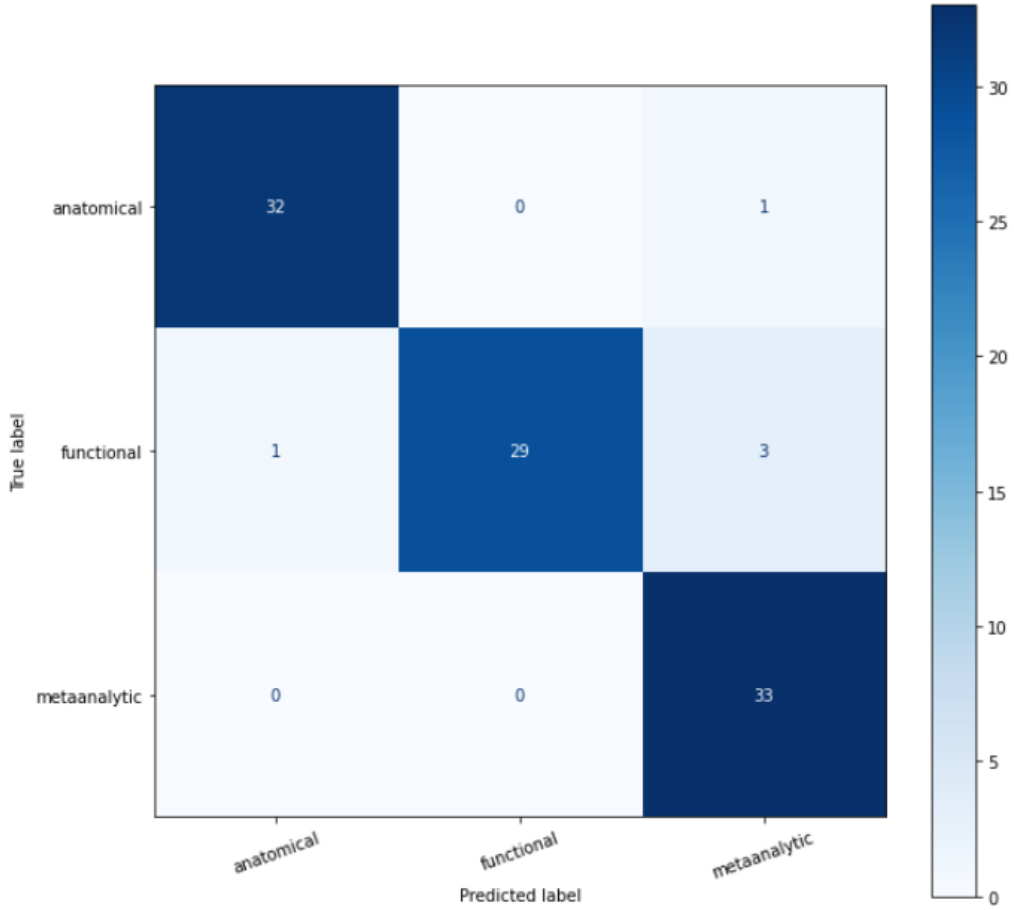

FIGURA 11. MATRIZ DE CONFUSIÓN OBTENIDA DEL ENTRENAMIENTO DE LA RED

Para este caso en el cual se tienen 3 clases la matriz de confusión. Esta matriz permite la visualización del rendimiento del algoritmo. Cada uno de los renglones representa las instancias en una clase predicha mientras que las columnas representan las instancias en la clase correspondiente. La matriz de confusión muestra los resultados de la clasificación de la arquitectura. Los resultados obtenidos muestran que la mayoría de los datos de entrenamiento de la predicción coinciden con los datos de prueba del conjunto de datos de entrenamiento. Aunque existen 5 imágenes que se clasificaron de forma errónea el resto fueron clasificadas correctamente.

Pruebas en GPU GTX 1050

Conceptos de concurrencia y paralelismo

El paralelismo se refiere a las tareas que se llevan a cabo al mismo tiempo. El paralelismo es usado cuando los problemas de cómputo son grandes tales como una gran cantidad de datos. Existe una relación con la concurrencia, pero la principal diferencia con el paralelismo es que cada tarea puede trabajar de forma independiente. La concurrencia se refiere a que las tareas se agendan de forma concurrente. En el caso de la concurrencia las tareas se agendan una después de otra. Un ejemplo

de concurrencia es el de agendar procesos en el sistema operativo. Un cluster se define como un grupo de computadoras que trabajan de forma cercana que trabajan de forma unida. Las computadoras en un cluster se conocen como nodos.

Para más información sobre conceptos de CAR referirse a (Wilkinson, 2004)

MEMORIA TÉCNICA DESCRIPTIVA MAX POOLING

DESCRIPCION GENERAL

El siguiente programa aplica la operación de Max Pooling sobre una matriz aleatoria la cual funciona como la imagen de entrada sobre la cual se va a aplicar la operación correspondiente para obtener el mapa de características. Debido a que este es un programa que utiliza una GPU para la operación de multiplicación de matrices para obtener la operación de Max Pooling se requiere de una GPU con capacidad de cómputo 3.5 o mayor para la librería Numba.-

Para este programa se utilizaron de las siguientes dependencias.

- Python versión 3.8
- Librería Numpy 1.23
- Librería Numba 0.53
- Librería Cupy 8.53
- CUDA versión 11.4 / 11.6 PARA TESLA T4

El programa ha sido probado en los sistemas operativos Windows 10 y Ubuntu Server 20.04. con CUDA en GPUs de la serie GTX1050 para laptop y GPUs para centros de datos Tesla T4 y Tesla K20. El uso de Numba CUDA en este programa es fundamental por lo que se recomienda revisar las instalaciones de los paquetes en sus versiones adecuadas para poder funcionar adecuadamente en la siguiente página.

https://numba.readthedocs.io/en/stable/cuda/overview.html#requirements

DESCRIPCION DEL PROGRAMA

Para este programa se deben identificar las variables involucradas en el proceso de la multiplicación de matrices y cuales son parte de la operación de Max Pooling. En el caso de la multiplicación de matrices las variables ‘filtro_imagen’, ‘resultado’ y ‘kernel’ son variables por medio de las cuales se realiza la operación de multiplicación por lo que estas variables deben ser globales debido a que las funciones de dispositivo Numba CUDA no regresan valores.

Las variables ‘imagen_entrada’ e ‘imagen_Final’ se declaran como variables globales las cuales estarán presentes a lo largo de la ejecución de todo el programa en su parte serial y de ‘imagen_entrada’ se extraerán los kernels correspondientes para la operación de Max Pooling.

```
#import random
import math
import numpy as np
#import threading
import time

import numba
#from numba import cuda
from numba import cuda, float32
import sys
import os
import cupy as cp

import warnings
from numba.core.errors import NumbaPerformanceWarning

import platform

### VARIABLES DE MAX POOLING

#imagen_entrada = []

#img_sinPadding = []  # copia de la imagen original para el padding

stride = 2
dim_entrada = 4

padding = 0

#imagen_Final = [] # matriz que guarda la imagen final

#### VARIABLES QUE INTERVIENEN EN LA PARALELIZACION MULT MATRICES
#filtro_imagen = []  # filtro
#resultado = []  # resultado del cual se extrae el max pooling


dim_mult = 2 # dimensiones por defecto de matrices de 2X2

numHilos = 1   #probar primero con un hilo

tiempoGPU = 0.0  # TIEMPO EN MSEG
numKernels = 0      # Numero de multiplicaciones (kernels extraidos)

# variable para generar matrices reproducibles
semillaNumAleatorio = None
```

FIGURA 12. DECLARACION DE VARIABLES GLOBALES

El programa comienza su ejecución verificando la disponibilidad de la GPU por medio del comando cuda.is_available(). Una vez que se verifica la disponibilidad de una GPU se llama a la función obtenerMaxPooling() para la inicialización de los parámetros previo al recorrido de la imagen para aplicar la operación de Max Pooling. Una vez que haya concluido el cálculo de la operación de Max Pooling se realiza el cálculo de las dimensiones de la imagen de salida por medio de la variable 'imagen_Final'.

La inicialización de variables comienza con el ingreso de las dimensiones de la imagen de entrada, así como el ingreso de hiperparámetros para la operación de Max Pooling tales como la dimensión del kernel, el desplazamiento del kernel en la imagen(stride) y el padding.

Este programa genera la imagen de entrada como una matriz aleatoria con rango que va desde 0 hasta un valor asignado, en este caso esta fijo a 9. Para este programa se trabajará con matrices

cuadradas tanto para la generación de la imagen como para los kernels de la imagen y la multiplicación de matrices para mantener las propiedades de los operadores de Schur.

El número de opciones permite hacer un ingreso manual de valores o bien hacer un ingreso aleatorio por medio de una función aleatoria. Una vez definida las dimensiones de la matriz de entrada y obtenida se realiza el ingreso de los hiperparámetros los cuales se utilizarán para la configuración de la operación de Max Pooling. El ingreso de la opción de Padding agrega ceros sobre la imagen previamente generada lo cual se consigue haciendo una copia del valor de la imagen previamente generada en una nueva imagen de una matriz de ceros. En esta función el ciclo for i-j actúa sobre el recorrido de la imagen original para agregar los valores de la imagen original en la nueva imagen con padding. Los valores se agregan en coordenadas de la forma imagen_entrada[i-padding][j-padding] y se regresa el valor con el padding correspondiente para que una vez generada la nueva imagen de entrada con el padding este se envíe de regreso a la función recorrerImagen().

```
344  # ------------------------------------------------------------------------
345
346  def agregarPadding(imagen_ent):
347      global dim_padding
348      #
349      # si es padding = 1
350      dim_padding = dim_entrada+padding*2
351
352      imagenPadding = np.zeros((dim_padding, dim_padding))
353      imagenPadding = imagenPadding.astype(int)
354
355      renglon_imagen = imagen_ent.shape[0]
356      col_imagen = imagen_ent.shape[1]
357
358      for i in cp.arange(padding, renglon_imagen+padding, step=1):
359          for j in cp.arange(padding,col_imagen+padding, step=1):
360
361              i = int(i)
362              j = int(j)
363
364              # copiar valor de la imagen anterior en la nueva imagen
365              imagenPadding[i][j] = int(imagen_entrada[i-padding][j-padding])
366
367
368      # ..........fin for i-j
369
370      return imagenPadding
371
372  # ------------------------------------------------------------------------
```

FIGURA 13. CICLOS PARA AGREGAR EL PADDING

Solamente se llama a esta función en caso de que se agrega el padding con un valor mayor o igual que 1. Una vez que todas estas variables estén inicializadas se realiza el recorrido de la imagen para sustraer los kernels. Debido a que las funciones de dispositivo también reciben el nombre de kernel

para evitar confusiones se le llamara funciones de dispositivo al kernel GPU refiriéndose a las funciones que se envían a la GPU.

```
70  def recorrerImagen(imagen_ent):
71      global imagen_Final
72      imagen_Final = []
73
74      renglon_imagen = imagen_ent.shape[0]
75      col_imagen = imagen_ent.shape[1]
76      #
77
78      for i in cp.arange(0, renglon_imagen, stride):
79          for j in cp.arange(0,col_imagen, stride):
80
81              i = int(i)
82              j = int(j)
83
84              # extrae el kernel correspondiente
85              kernel = imagen_ent[i:i+dim_mult, j:j+dim_mult]
86
87              # si el kernel es cuadrado extraerlo para enviarlo a la multiplicacion
88              # de matrices en caso contrario ignorar el kernel
89              if kernel.shape == (dim_mult, dim_mult):
90
91                 # aplicar la operacion de max pooling
92                 # -----------------------
93                 kernel = np.array(kernel)
94                 '''
95                 print("===kernel=====1")
96                 print(" " + str(kernel))
97                 print("==============2")
98                 '''
99                 ##............... PARALELIZACION .............##
100                '''
101                print("---------------6")
102                print("valor_max= " + str(valor_max))
103                print("---------------6")
104                '''
105                #para evitar un error logico en Max Pooling
106                #resultado = np.zeros((dim_mult, dim_mult))
107                #resultado = resultado.astype(int)
108
109                valorMax = multGPU2(kernel)    #### ---> enviar el kernel extraido a la mult de matrices
110
111                imagen_Final.append(valorMax)
112
113             # ............fin if
114     # ..........fin for i-j
```

FIGURA 14. CICLOS PARA RECORRER IMAGEN

La función de recorrerImagen() realiza un recorrido de la imagen representada como una matriz aleatoria bidimensional por medio de un ciclo for anidado para poder realizar la extracción de los kernels correspondientes y revisar que cumplan con las dimensiones que se establecieron previamente. Una vez que se extrae el kernel este se envía a la función multGPU2() para realizar el lanzamiento de la función de dispositivo en la GPU, esta función debe de regresar el valor máximo de cada una de las operaciones de multiplicación de matrices para agregarlo posteriormente a la lista imagen_Final. Los ciclos por medio de la función cp.arange() pueden manejarse de mejor forma por medio de esta librería para la GPU cuando los ciclos para los arreglos que debe de procesar son más grandes.

```python
# ------------------------------------------------------------------
def ingresoHiperparametros():
    # intervienen en la paralelizacion
    global dim_mult

    # variables e hiperparametros de Max Pooling

    global stride
    global padding

    print()
    while True:
        try:
            dim_mult = input("Ingresar la dimension del kernel : ")
            dim_mult = int(dim_mult)
            break
        except ValueError:
            print("Entero no valido. Intente de nuevo ...")
    print("dim kernel ingresado.")

    while True:
        try:
            stride = input("Ingresar el stride (desplazamiento) : ")
            stride = int(stride)
            break
        except ValueError:
            print("Entero no valido. Intente de nuevo ...")
    print("stride ingresado.")

    while True:
        try:
            padding = input("Ingresar el padding : ")
            padding = int(padding)
            break
        except ValueError:
            print("Entero no valido. Intente de nuevo ...")
    print("Padding ingresado.")


# ------------------------------------------------------------------------
```

FIGURA 15. INGRESO DE HIPERPARAMETROS PARA LA OPERACIÓN DE MAX POOLING

```
228
229 def ingresoAleatorio():
230     global imagen_entrada
231     global semillaNumAleatorio
232
233     while True:
234         try:
235             matricesRep = input("Genera matrices reproducibles?(0 ACEPTAR, otro valor descartar )?: ")
236             matricesRep = int(matricesRep)
237
238             if matricesRep == 0:
239                 semillaNumAleatorio = True
240                 print("Matrices reproducibles (semilla)")
241             else:
242                 print("Se generara una matriz aleatoria")
243                 semillaNumAleatorio == False
244
245             break
246         except ValueError:
247             print("Entero no valido. Intente de nuevo ...")
248     print("Opcion ingresada.")
249
250     if semillaNumAleatorio == True:
251         np.random.seed(35)
252
253     valores = 10   # valores aleatorios desde el 0 hasta el 9
254     imagen_entrada = np.random.randint(valores, size=(dim_entrada, dim_entrada))
255     imagen_entrada = imagen_entrada.astype(int)
256
```

FIGURA 16. LA OPCION DE INGRESO ALEATORIO GENERA UNA MATRIZ ALEATORIA

```
# ------------------------------------------------------------------------
def ingresoManual():
    global imagen_entrada

    imagen_entrada = np.zeros((dim_entrada, dim_entrada)).astype(int)

    renglon_imagen = imagen_entrada.shape[0]
    col_imagen = imagen_entrada.shape[1]

    numValores = imagen_entrada.size
    print("Num. valores a ingresar", numValores)
    print()

    cont = 1 # conteo de numero de valores
    print("Ingreso manual de la matriz")
    print("---------------------------")
    for i in np.arange(renglon_imagen):
        print("Renglon {}  ".format(i))
        print("-----------")
        for j in np.arange(col_imagen):
            print(f"# Valor={cont} | img[{i}][{j}]", end='')

            #imagen_entrada[i][j] = int(input(" = "))
            while True:
                try:
                    imagen_entrada[i][j] = input(" = ")
                    imagen_entrada[i][j] = int(imagen_entrada[i][j])
                    break
                except ValueError:
                    print("Entero no valido. Intente de nuevo ...")

            cont = cont + 1
        print()   # nueva linea

    # ----- fin for i-j


# ------------------------------------------------------------------------
```

FIGURA 17. LA OPCIÓN DE INGRESO MANUAL INGRESA VALORES EN LA MATRIZ

Lanzamiento de la función de dispositivo GPU

La función multGPU2() es la encargada de realizar la configuración y el lanzamiento de la función de dispositivo a la GPU. Se realiza la configuración del bloque GPU donde se realiza el envío de las variables involucradas en la multiplicación de matrices al dispositivo, en este caso las variables ‘kernel’, ‘filtro_imagen’ y ‘resultado’. Todas estas variables están previamente declaradas como arreglos con las dimensiones del kernel extraído establecido previamente en la configuración de los hiperparámetros.

Posteriormente se realiza la configuración de los bloques GPU. Debido a que es un problema bidimensional se configura por medio de la variable HPB (hilos por bloque) a 32 resultando en 1024 = (32 x 32 x 1). Los hilos en la GPU se agrupan en warps (grupos de 32 hilos) por lo que estos valores se deben configurar en múltiplos de 32 o bien de 16 como medio warp. Para el caso de problemas bidimensionales se utilizan como límite 1024 hilos para las dimensiones x e y, así como 64 para la

dimensión z. Una vez configurados los hilos por bloque se configura los bloques de la GPU que serán utilizados lo cual se envía en un formato en forma de tupla. Se establece el número de bloques de acuerdo a las dimensiones de la imagen para poder obtener el número de bloques necesarios para realizar la multiplicación de matrices.

Previo a lanzar la función de dispositivo la variable 'resultado' debe resetearse para evitar acumular valores de una multiplicación anterior. Esta operación de reseteo es importante para evitar acumulaciones de un cálculo anterior y evitar errores lógicos. Antes del lanzamiento de la función de dispositivo también se declara la creación de eventos para realizar la medición del tiempo desde la GPU. Previo se declaran variables con Cupy de las variables relacionadas con las multiplicaciones de matrices para cargarlas a la memoria de la GPU.

El lanzamiento de la función de dispositivo se realiza por medio de la declaración

matmul3[blockspergrid, threadsperblock](kernel_cupy, resultado_cupy)

donde las variables en forma de tupla entre corchetes previamente establecidas se refieren al número total de hilos en el bloque y al número total de bloques en el grid. Las variables entre paréntesis se refieren a las variables que se envían al dispositivo GPU. La variable filtroImagen la cual es una matriz identidad utilizada en la multiplicación de matrices puede ser declarada como constante dentro de la definición del kernel GPU para que haya una menor transferencia de variables en el lanzamiento del kernel GPU y reducir las comunicaciones entre el CPU y la GPU.

```
# configurar los bloques GPU para matmul3 (multiplicacion con memoria compartida)
# ================================================================================
grid_y_max = max(kernel.shape[0], filtro_imagen.shape[0])
grid_x_max = max(kernel.shape[1], filtro_imagen.shape[1])
blockspergrid_x = math.ceil(grid_x_max / threadsperblock[0])
blockspergrid_y = math.ceil(grid_y_max / threadsperblock[1])

blockspergrid = (blockspergrid_x, blockspergrid_y)

# ================================================================================



# ---------------------------------------------------
resultado = np.zeros((dim_mult,dim_mult)).astype(int)   # reseteo de la multiplicacion


# se crean dos eventos uno al inicio del computo y otro al final del computo
inicioEvento = cuda.event()
finEvento = cuda.event()

# iniciar el kernel
inicioEvento.record(stream=stream)

'''
print("blocks per grid =", blockspergrid)
print("ThreadsPerBlock =", threadsperblock)
'''
# numero total debloques en el grid
# numero total de hilos en el bloque

#matmul2[blockspergrid, threadsperblock](kernel_device, filtro_device, resultado_device, indices_device)

matmul3[blockspergrid, threadsperblock](kernel_cupy , resultado_cupy)

finEvento.record(stream=stream)
```

FIGURA 18. LA FUNCION MULTGPU CONFIGURA LOS PARAMETROS DEL KERNEL

Una vez concluido el proceso por parte de la función de dispositivo se realiza el fin de medición de ejecución de esta función por parte de los eventos y se sincronizo el evento con el CPU con la finalidad de que pueda utilizar sus valores hasta que el proceso termine. Una vez terminada las mediciones los valores de la variable 'resultado' se regresan al host. Este valor se utiliza para obtener el valor máximo a partir de la multiplicación de matrices el cual se envía de regreso a la función recorrerImagen().

Función de dispositivo matmul3()

La multiplicación de matrices puede considerarse un problema computacionalmente intensivo sobre todo cuando aumenta el tamaño de la multiplicación de matrices que debe de realizarse. A partir del análisis de la operación de Max Pooling la operación de la multiplicación de matrices se considera una región del programa que puede ser paralelizable por lo que para esta parte del programa se probaron distintas funciones de dispositivo para poder ver cuál sea el más adecuado para realizar la multiplicación de matrices.

Para la función matmul3() encargada de realizar la multiplicación C = A * B donde A es el kernel, B es el filtro y C es el resultado se puede declarar como constante la variable 'filtro_imagen' el cual es una matriz identidad que funciona como elemento neutro en la multiplicación de matrices por medio de la propiedad C = A * I.

Esta función al ser B constante solamente recibe dos arreglos reduciendo los tiempos de transferencia al enviar menos variables. La función utiliza la memoria compartida de la GPU al definir un arreglo en esta memoria donde el tamaño y tipo de los arreglos deben estar previamente definidos. Se declaran posteriormente los índices de la rejilla para problemas en 2D siguiendo las reglas de la sintaxis de Numba CUDA.

Cada uno de los hilos se encarga de realizar el cálculo de un elemento del resultado de la multiplicación donde esta operación se divide en productos punto de vectores largos y se procesan en ciclo conforme se establece la longitud de los bloques en el grid (GPU). Se calculan los productos parciales sobre la memoria compartida previo a que se realice la precarga y terminen sus cálculos por medio de la instruccion cuda.syncthreads(). Una vez concluido el proceso de la multiplicación de matrices el valor de la variable resultado se envía de regreso al host (CPU) para poder ser utilizado en el proceso de Max Pooling.

```
499  HPB = 32
500
501  @cuda.jit
502  def matmul3(A, C):
503      '''
504      Realiza la multiplicacion de matrices de C = A * B usando memoria compartida de CUDA
505
506      '''
507      B = cuda.const.array_like(filtro_imagen)
508
509      # define un arreglo en la memoria compartida
510      # el tamanio y el tipo de los arreglos deben ser conocidos en tiempo de compilacion
511      sA = cuda.shared.array(shape=(HPB, HPB), dtype=float32)
512      sB = cuda.shared.array(shape=(HPB, HPB), dtype=float32)
513
514      # indices de la rejilla para problemas en 2D
515      x, y = cuda.grid(2)
516
517      tx = cuda.threadIdx.x
518      ty = cuda.threadIdx.y
519      bpg = cuda.gridDim.x      # blocks per grid
520
521      # cada hilo calcula un elemento en la matriz resultante
522      # el producto punto es fragmentado en productos punto de vectores largos-HPB
523      tmp = float32(0.)
524      for i in range(bpg):
525          # precargar datos en la memoria compartida
526          sA[ty, tx] = 0
527          sB[ty, tx] = 0
528
529          if y < A.shape[0] and (tx + i * HPB) < A.shape[1]:
530              sA[ty, tx] = A[y, tx + i * HPB]
531          if x < B.shape[1] and (ty + i * HPB) < B.shape[0]:
532              sB[ty, tx] = B[ty + i * HPB, x]
533
534          # esperar hasta que todos los hilos terminen su precarga
535          cuda.syncthreads()
536
537          # calcular el producto parcial sobre la memoria compartida
538          for j in range(HPB):
539              tmp += sA[ty, j] * sB[j, tx]
540
541          # esperar hasta que todos los hilos terminen de calcular
542          cuda.syncthreads()
543      if y < C.shape[0] and x < C.shape[1]:
544          C[y, x] = tmp
```

FIGURA 19. LA FUNCION MATMUL3 REALIZA MULTIPLICACIÓN POR MEDIO DE SEGMENTOS Y MEMORIA COMPARTIDA

Otras funciones de dispositivo matmul1() y matmul2()

Estas funciones de dispositivo realizan la multiplicación de matrices y funcionan bajo principios similares, sin embargo, la forma en la cual recorrer el arreglo es diferente entre ambas funciones. El for -k es utilizado para la multiplicación de los elementos renglón – columna para obtener cada uno de los valores de la multiplicación de matrices.

Las funciones matmul1() funciona por medio de una instrucción de selección if la cual revisa que la matriz 'resultado' cumpla con las dimensiones de la matriz utilizando los índices de la rejilla 2D de Numba CUDA. El for-k es el encargado de realizar las multiplicaciones renglón-columna por medio de una variable auxiliar tmp la cual funciona como acumulador para esta operación por cada valor del proceso de multiplicación renglón-columna. Una vez terminado este ciclo se asigna el valor de esta variable al valor correspondiente en la variable C para el resultado de la multiplicación de matrices.

La función matmul2 funciona por medio de un for anidado i-j-k donde los índices i-j de los dos primeros ciclos corresponden a las coordenadas de los hilos en la rejilla 2D de Numba CUDA.

Los dos primeros ciclos se utilizan para hacer un recorrido de la matriz que se está multiplicando mientras que el tercer ciclo for-k se utiliza para la multiplicación de los elementos renglón-columna los cuales se guardan en la variable auxiliar suma. Esta variable se resetea por cada multiplicación renglón-columna y una vez que termine la multiplicación de los elementos renglón – columna se asigna a la variable C.

```
440     # ------------------------------------------------------
441     # realiza la multiplicacion de matrices cuadradas
442     # ------------------------------------------------------
443     # CUDA kernel HOST --> DEVICE
444     # ### ---------------------- CODIGO DE DEVICE
445     @cuda.jit
446     def matmul(A, B, C):
447         i, j = cuda.grid(2)
448         if i < C.shape[0] and j < C.shape[1]:
449             tmp = 0
450             for k in range(A.shape[1]):
451                 tmp += A[i, k] * B[k,j]
452             C[i, j] = tmp
453     # ------------------------------------------------------
454     # ------------------------------------------------------
455     # CUDA kernel HOST --> DEVICE
456     # ### ---------------------- CODIGO DE DEVICE
457     @cuda.jit
458     def matmul2(A, B, C, indices):
459         # se obtienen las coordenadas de hilos unicas x, y en el grid 2D
460         i, j = cuda.grid(2)
461 
462         # The above is equivalent to the following 2 lines of code:
463         #i = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
464         #j = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y
465 
466         for i in range(i, dim_mult):
467             for j in range(j, dim_mult):
468                 suma = 0
469                 for k in range(A.shape[1]):
470                     suma += A[i, k] * B[k,j]
471                     #indices[i][j] = i  + j/ 10
472                 C[i, j] = suma
473 
474         # fin for------------------- i.j-k
475         # Escribir el indice-x seguido por un decimal y el indice-y
476         #
477         indices[i][j] = i + j / 10
478 
```

FIGURA 20. LAS FUNCIONES MATMUL1 Y MATMUL2 PARA LA MULTIPLICACIÓN DE MATRICES

En caso de que se requiera consultar los índices dentro del proceso de multiplicación se puede agregar una variable 'indices' la cual es un arreglo al final de la función que guarda los valores de las coordenadas de los índices i-j de la rejilla 2D utilizada para problemas de dos dimensiones.

Al resultado de la multiplicación de matrices una vez que se regresa el control a la variable resultado al CPU este resultado se envía a la función valorMayor() para obtener el valor máximo del resultado de la multiplicación de matrices. Esta operación se realiza por medio de un ciclo for anidado para realizar una comparación sucesiva de valores estableciendo el primer valor como el mayor y realizando dicha comparación hasta completar el recorrido del arreglo. Los límites de los ciclos en esta función se obtienen a partir de la propiedad shape de Numpy aunque se puede utilizar la variable dim_mult debido a que esta variable tiene las mismas dimensiones que las otras dos variables para la multiplicación de matrices.

Obtención del mapa de características a partir de la imagen original.

Una vez que concluya el recorrido de la imagen y se hayan obtenido todos los valores a partir de la multiplicación de matrices se obtienen las dimensiones del mapa de características que representa la imagen sobre la cual se aplicó el Max Pooling por medio de la siguiente fórmula.

n_salida = math.floor(((n_ent + 2*p -k)/s)) + 1

Donde el método math.floor() regresa el número entero más cercano menor o igual a un número dado para representar la función de piso, $\lfloor a \rfloor$ también conocido como función de suelo.

El tamaño de salida de imagen_Final se calcula con esta fórmula usando los siguientes hiperparámetros.

- n_ent dimensión del tamaño de entrada
- p tamaño del padding
- k dimensión del kernel de Pooling
- s Stride tamaño del stride o desplazamiento del kernel de Pooling

Debido a que este programa funciona con dimensiones cuadradas el valor de n_salida es el mismo para coordenadas de x y de y para la representación del mapa de características.

```
776
777  def obtenerMaxPooling():
778      global imagen_Final
779
780      global tiempoGPU
781      global numKernels
782
783      tiempoGPU = 0.0  # reseteo del tiempo de la mult matrices GPU (en mseg)
784      numKernels = 0   # reseteo de contador
785
786
787      ingresoMatriz()
788      ingresoHiperparametros()
789
790
791      imagen = inicializarMatrizEnt()
792      inicializarMatrizMult()
793
794
795      tiempo_inicio = time.time()
796      recorrerImagen(imagen)
797
798      tiempo_fin = time.time()
799      print()
800      tiempoExperimento = tiempo_fin - tiempo_inicio
801
802
803
804      # calcular el tamanio de salida para 'imagen_Final'
805      n_ent = dim_entrada
806      p = padding                    # tamanio del padding
807      k = dim_mult                   # tamanio del kernel de convolucion
808      s = stride                     # tamanio del stride de convolucion
809      #print("--------")
810      #print(n_ent)
811      #print(p)
812      #print(k)
813      #print(s)
814
815      #print("------")
816
817      n_salida = math.floor(((n_ent + 2*p -k)/s)) + 1
818      # math.floor devuelve el numero entero mas cercano menor o igual a un numero dado
819      #print(n_salida)
```

FIGURA 21. FUNCIÓN PARA LANZAR LA FUNCIÓN RECORRERIMAGEN

La variable ‘imagen_Final’ la cual definida como lista se convierte a un arreglo de Numpy con las dimensiones establecidas de n_salida funciona como una matriz de dimensiones cuadradas.

Al final de la función obtenerMaxPooling() se muestra una serie de print() y opciones para mostrar los resultados del experimento, tales como la llamada a la función de impresión para imprimir el mapa de características generado. Esta sección incluye además una serie de opciones para desplegar información sobre los experimentos a modo de resumen tal como las dimensiones de la imagen de Max Pooling, así como los contadores de tiempo de los procesos de multiplicación.

Las funciones info_CPU() y revisarVersionGPU() son funciones que buscan mostrar las características tanto del CPU como del GPU. La primera utiliza funciones de la biblioteca estándar de Python para revisar versiones del sistema operativo, tipo de procesador CPU. Esta característica se debe de tomar en cuenta debido a que este programa funciona como un sistema heterogéneo donde muchas de las funciones del programa funcionan desde el host CPU.

La función de revisarVersionGPU() detecta las versiones de Python y las dependencias utilizadas en este programa, así como las características de la GPU por medio de cuda.detect().

Interoperabilidad y ajustes

El uso de la librería Cupy como equivalente de Numpy en la GPU. Esta librería implementa muchas de las funcionalidades de Numpy como definición de arreglos y métodos para arreglos que pueden usarse en la memoria de la GPU, sin embargo, se debe de tomar en cuenta las transferencias entre el CPU y la GPU para que no afecte el rendimiento en el programa.

así como otros ajustes permitieron reducir los tiempos de operación del programa y tener un mejor manejo de los arreglos conforme estos aumentan el tamaño de entrada durante los experimentos. Para el caso de arreglos de dimensiones 2, 5, o 9 se consideran arreglos pequeños mientras que para arreglos de dimensiones 500, 1000 o mayores se pueden considerar arreglos grandes. La librería de Cupy es más adecuada para arreglos de un mayor tamaño además de ejecutar los ciclos más rápidamente para el caso de arreglos más grandes. Esta librería puede utilizarse para procesar los ciclos o puede guardar variables en la memoria de la GPU. En caso de que los arreglos se declaren como arreglo Cupy están cargados en la memoria de la GPU y nuevamente deben de convertirse a su equivalente Numpy en caso de ser utilizados en el CPU.

Para la definición de los kernels se debe de tomar en cuenta que estas funciones definidas en Numba CUDA no pueden tener funciones de librerías tales como Numpy, tal como los métodos de la librería o definición de nuevas variables por medio de la librería, entre otras características. Solamente instrucciones de ciclos y condicionales básicas del lenguaje de Python, así como operaciones definidas por el lenguaje son permitidas dentro de la definición del kernel de Numba CUDA.

EXPERIMENTOS ETAPA 1 Y ETAPA 2 MAX POOLING

En las gráficas obtenidas se puede ver un comportamiento de tipo exponencial obtenido en los experimentos lo cual muestra que conforme aumenta el tamaño de entrada se incrementa el número de operaciones a realizar por parte de la operación de Max Pooling.

Cuando se ejecuta un programa en CUDA la primera ejecución siempre es más tardada debido a que se debe de inicializar la GPU. En las siguientes ejecuciones del programa con la GPU previamente inicializada se debe de obtener un procesamiento más rápido.

ETAPA 1 PRUEBAS INICIALES

Para los experimentos de max pooling se busca hacer las pruebas en un ambiente local en una computadora en sistema operativo Linux Ubuntu 20.04 desde máquina virtual con VirtualBox. El sistema operativo base es Windows 10 en un procesador Core i-7 con 4 núcleos y 8 procesos. Las siguientes pruebas fueron realizadas el 25 de enero de 2023 variando los hiperparámetros de stride, padding y kernel. Se realizaron pruebas con diferente número de hilos para probar la paralelización de la multiplicación de matrices.

- Procesador Intel Corei5 de 4 procesadores y 8 procesos
- Disco duro 500 Gb
- Memoria 32 Gb
- GPU GTX 1050

Para esta etapa se busca comprobar el funcionamiento del programa y se realizan pruebas donde se prueba el ajuste con distintos tamaños de entrada e hiperparametros. Se busca que esta etapa tenga se familiarice con los aspectos referentes a la parte de inteligencia artificial de este trabajo incluyendo las propiedades referentes a la operación de Max Pooling.

Para facilitar la revisión de las pruebas se realiza una prueba con tamaños de entrada de 4 y 8 y los siguientes hiperparámetros.

- Kernel 2
- Stride de 2
- Padding de 0

```
Ingresar el padding : 0
Padding ingresado.

--Imagen de entrada--
[ 7  6  6  2 ]
[ 5  9  5  9 ]
[ 0  6  9  9 ]
[ 2  0  3  3 ]
Imagen Ent: (4, 4)



----MAX POOLING CON GPU----



   RESULTADO MAX POOLING

[ 9  9 ]
[ 6  9 ]
Imagen Final: (2, 2)

RESULTADOS DEL EXPERIMENTO
------------------------------------------------------------
Img. Entrada     : 4X4 | # Elementos : 16  | Memoria : 64 bytes.

Img. Max Pooling : 2X2 | # Elementos : 4   | Memoria : 16 bytes.

Tiempo medido desde CPU = 3.907677 seg | #
Tiempo mult con GPU = 3.251328 mseg |  0.003251 seg
```

FIGURA 22. RESULTADO DE LA OPERACIÓN DE MAX POOLING IMG 4X4

```
Ingresar el padding : 0
Padding ingresado.

--Imagen de entrada--
[ 7  5  7  5  9  6  8  0 ]
[ 9  7  3  1  2  7  7  9 ]
[ 7  1  7  8  3  4  9  6 ]
[ 6  9  5  9  8  3  8  5 ]
[ 9  8  0  0  3  3  2  3 ]
[ 0  1  9  1  2  0  4  9 ]
[ 8  9  9  3  5  6  0  2 ]
[ 7  9  5  2  8  0  4  8 ]
Imagen Ent: (8, 8)



----MAX POOLING CON GPU----



   RESULTADO MAX POOLING

[ 9  7  9  9 ]
[ 9  9  8  9 ]
[ 9  9  3  9 ]
[ 9  9  8  8 ]
Imagen Final: (4, 4)

RESULTADOS DEL EXPERIMENTO
------------------------------------------------------------
Img. Entrada     : 8X8 | # Elementos : 64  | Memoria : 256 bytes.

Img. Max Pooling : 4X4 | # Elementos : 16  | Memoria : 64 bytes.
```

FIGURA 23. EXPERIMENTO DE LA OPERACIÓN DE MAX POOLING IMG 8X8

Nuevamente se hacen pruebas con entrada de 4 y con los mismos hiperparametros, pero con stride de 1 para revisar el solapamiento de los kernels y que esto no afecte la extracción de los procesos. Se realiza otra prueba con los siguientes hiperparametros para probar que el solapamiento no afecte los procesos, así como ver los efectos del padding en la operación de Max Pooling

- Kernel 2
- Stride de 1
- Padding de 1

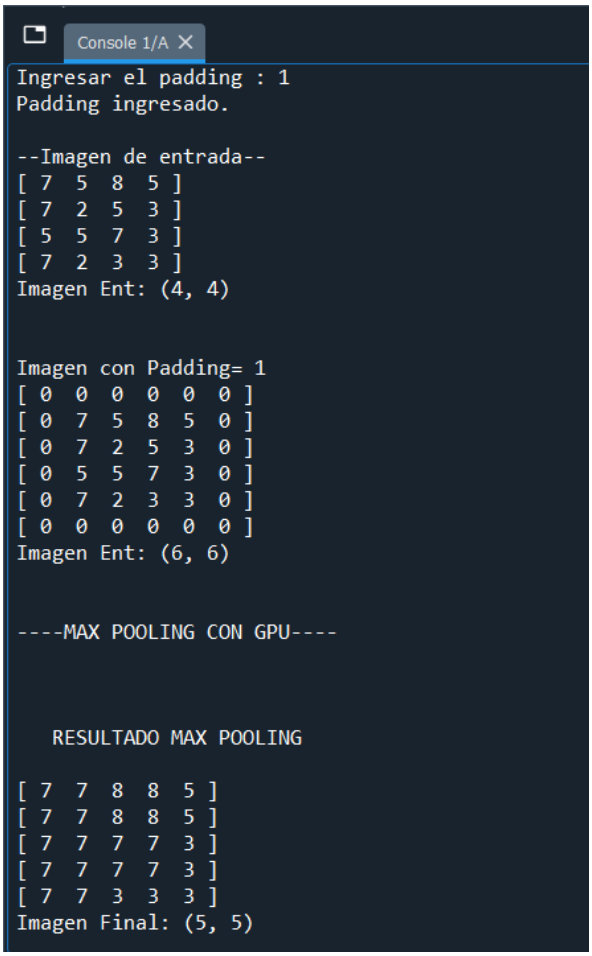

FIGURA 24. EXPERIMENTO DE MAX POOLING CON PADDING

Agregar padding a la imagen permite que se aumente el tamaño de la matriz. Este hiperparámetro evita que se pierdan características en las orillas de la imagen y la reducción de la imagen después de aplicar la operación de Max Pooling. Al aplicar un padding de 2 se puede mantener un tamaño similar de la imagen.

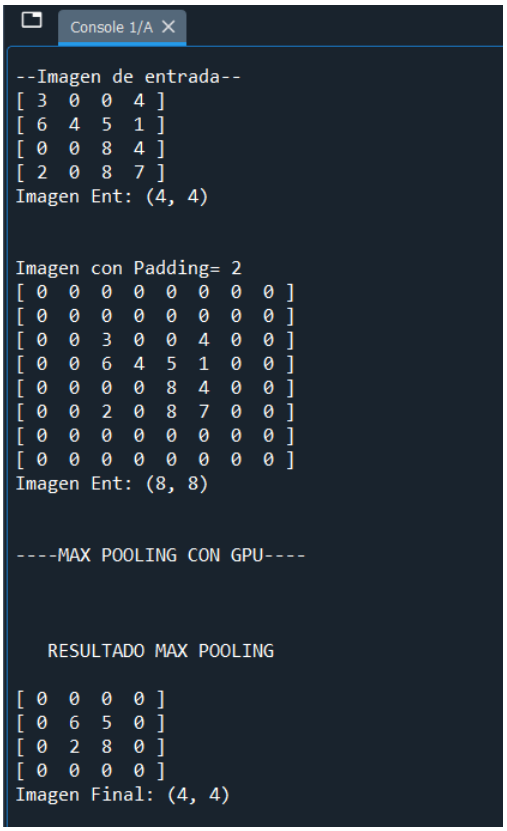

FIGURA 25. EXPERIMENTO CON PADDING = 2 PARA PRESERVAR LAS DIMENSIONES ORIGINALES DE LA IMAGEN DE ENTRADA

Se pueden realizar las pruebas de la etapa 1 también en una computadora con Linux Ubuntu 20.04 como sistema operativo en ambiente local o por medio de cluster. En el apéndice A y B al final de este documento se encuentran detalles de la instalación de las librerías necesarias para hacer las pruebas locales en ambiente Linux.

Las imágenes siendo una representación matricial en diferentes canales podemos interpretarla para una imagen de 1 solo canal como una matriz bidimensional para la representación de una imagen en escala de grises. En estas pruebas iniciales se busca como objetivo de estos experimentos comprobar el correcto funcionamiento del programa para posteriormente pasar a las pruebas posteriores. Otro de los objetivos es que se defina el conjunto de prueba sobre el cual se realizaran las pruebas en el programa de Max Pooling.

Para facilitar los experimentos se mantienen los hiperparametros de stride, kernel a 2 y un padding de cero para evitar el solapamiento de los kernels y los mapas de características se reducen a la mitad.

ETAPA 2 PRUEBAS EN CLUSTER

una vez concluida las pruebas iniciales se busca que en las pruebas posteriores sea realicen pruebas sobre diferentes conjuntos de prueba con un mayor número de procesos en cluster.

El principal objetivo de las técnicas de cómputo de alto rendimiento es obtener una mayor velocidad de procesamiento en los problemas que se buscan resolver por lo que en estas pruebas se realizaran pruebas con distinto número de procesadores para poder observar los cambios en la velocidad de procesamiento del conjunto de datos después de aplicar Max Pooling para posteriormente observar los beneficios de la paralelización con distinto número de procesadores.

Una vez que el programa funcione correctamente sobre las imágenes de prueba se busca realizar las pruebas en un mayor número de procesadores en clúster. El programa probado inicialmente en ambiente local se busca realizar pruebas en un ambiente distribuido donde se realizarán experimentos con un número distinto de procesos. Para estos experimentos se cuentan con dos nodos de un clúster con procesadores Intel Xeon con 20 procesos cada uno en ambiente Linux y una velocidad de reloj mayor que la de un CPU de escritorio. Se puede consultar las especificaciones técnicas del procesador por medio del comando $lscpu.

Las pruebas se realizarán con 2, 4, 8, 16 y 32 procesos para ver los tiempos de respuesta de ejecución del programa con distinto número de procesos sobre un conjunto de prueba definido anteriormente en las pruebas iniciales.

Proyecciones del comportamiento de Max Pooling.

A continuación, se muestra una proyección con los hiperparametros fijos en padding = 0, stride = 2 y dimensiones del kernel como 2 conforme el tamaño de la matriz sigue aumentado el número de kernels extraidos (multiplicaciones realizadas) aumentan siguiendo este patrón.

$$n_{actual} = 4 * n_{anterior}$$

Observando el código en versión serial y utilizando como método de multiplicación se tiene una complejidad O(n´5) la cual hace a la operación de Max Pooling un problema computacionalmente intensivo. Los primeros dos ciclos se utilizan para el recorrido de la imagen la cual se representa como una matriz bidimensional. Los otros ciclos son por parte de los ciclos para calcular la multiplicación de matrices.

PROYECCION DE NUM- DE KERNELS p=0, s=2, k=2		
NUM DE KERNELS GENRADOS VS TAM ENTRADA		
	Dim. Matriz cuadrada	PROYECCIONES
# Experimento	Tam Matriz	# Kernels extraidos
1	4	4
2	8	16
3	16	64
4	32	256
5	64	1024
6	128	4096
7	256	16384
8	512	65536
9	1024	262144
10	2048	1048576
11	4096	4194304
12	8192	16777216
13	16384	67108864
14	32768	268435456
15	65536	1073741824
16	131072	4294967296

FIGURA 26. PROYECCIONES DEL AUMENTO DE KERNELS CONFORME SE INCREMENTA EL TAMAÑO DE ENTRADA

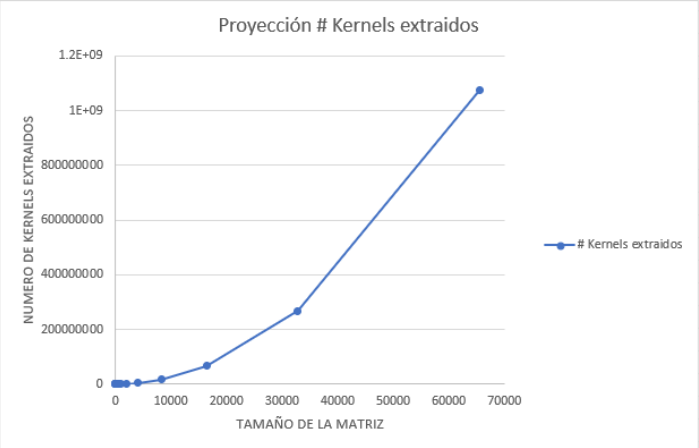

FIGURA 27. TAMAÑO DE ENTRADA VS NÚMERO DE KERNELS EXTRAIDOS

Los experimentos muestran que el aumento de los kernels extraídos (multiplicaciones) aumentan siguiendo este patrón. Los experimentos de la etapa 2 se realizaron con diferentes hardware en cluster con sistemas Linux Ubuntu 20.04 para poder realizar pruebas que requieran un mayor tiempo de ejecución y con recursos con una mayor cantidad de núcleos CUDA por parte de las GPU disponibles.

Características de cluster

Para la realización de las pruebas se cuenta con dos nodos de cluster con procesadores Intel Xeon de 12 procesadores con 24 procesos cada uno. Las tarjetas NVIDIA tienen una capacidad de cómputo lo cual es una métrica de referencia para medir su capacidad de procesamiento.

Capacidad de cómputo de las GPUs

Este número se refiere a la generación que cada una de las GPUs pertenecen donde se resume una serie de características de relevancia de la GPU tales como la arquitectura de la tarjeta a la que pertenece, entre otras características. El primer número se refiere a la generación y arquitectura de la tarjeta. El segundo valor se refiere a las características o mejoras referentes a un modelo particular de la GPU perteneciente a esa familia o arquitectura.

Para los experimentos de etapa 2 se utilizaron las siguientes GPUs en cluster con el sistema operativo Linux Ubuntu Server 20.04 en ambos nodos. A continuación, se muestran las características de los dos nodos del cluster en el cual se desarrollan los experimentos.

Tesla K20 en el nodo n48

- Arquitectura Kepler
- 2496 núcleos
- Memoria de 5 Gb
- Capacidad de cómputo 3.5
- CUDA versión 11.4

Tesla T4 en el nodo n49

- Arquitectura Turing
- 2560 núcleos CUDA
- 320 núcleos tensores
- Memoria de 16 Gb
- Capacidad de cómputo 7.5
- CUDA versión 11.6

La lista completa de capacidad de cómputo de las GPUs de NVIDIA se puede consultar en la siguiente página.

https://developer.nvidia.com/cuda-gpus

Cada núcleo CUDA puede realizar una operación por ciclo de reloj. Estos núcleos pueden realizar tareas en paralelo tomando en cuenta el número de núcleos disponibles y la velocidad de reloj manejada. La GPU Tesla T4 además de sus núcleos CUDA tiene además 320 núcleos tensores. Estos núcleos aparecieron por primera vez en las GPUs de arquitectura Volta lo cual permite un procesamiento mucho más veloz para aplicaciones de inteligencia artificial. Estos núcleos tienen entrenamiento de precisión mixta. Las GPUs para centros de datos permiten realizar experimentos en tiempos más prolongados con un mejor manejo de la temperatura, además de que por lo general

tienen una mayor cantidad de memoria dependiendo de la GPU. Esta es una característica importante para el entrenamiento de modelos de redes neuronales. Junto con el CPU para servidor que recordemos que el CPU junto con la GPU forman un sistema heterogéneo capaz de ser una herramienta que puede resolver gran cantidad de problemas científicos o de ingeniería.

Durante la realización de los experimentos se hicieron pruebas con 3 diferentes diseños de funciones de dispositivo para la implementación de la multiplicación de matrices. En el caso del primer caso la función matmul1 se puede observar que este es una de las implementaciones más clásicas para la multiplicación de matrices a través de 3 ciclos for, donde los dos primeros tienen los índices de CUDA para realizar el recorrido del arreglo mientras el ciclo for k más anidado realiza las operaciones renglón columna de la multiplicación de matrices.

Este primer kernel GPU genero resultados muy similares sobre un conjunto de entrada de números aleatorios que van desde 0 hasta 9 como representación de la imagen de entrada de un arreglo de número enteros generado a partir de una función aleatoria. Para el caso del segundo kernel se utilizó una serie de instrucciones de if la cual permite elegir valores basados en instrucciones condicionales en vez de recorrer el arreglo por completo por medio de ciclos. Para el caso de los experimentos se utilizaron experimentos en dos GPU en cada nodo con un procesador Intel Xeon el cual se encarga de procesar la parte serial del programa debido a que los programas GPU funcionan por medio de un sistema heterogéneo entre la CPU que procesa la parte serial y la GPU que procesa la parte paralela.

Para el caso del tercer kernel se utilizó en kernel basado en memoria compartida y dividir la multiplicación en segmentos para poder obtener un mayor grado de paralelismo. Este kernel permite una multiplicación rápida en el caso de multiplicaciones de un mayor tamaño al realizar la división en segmentos además de utilizar los registros de memoria compartida de la GPU lo cual es una memoria de mayor velocidad que la memoria global DRAM de la GPU.

Los experimentos han mostrado una mejoría en el tiempo de la ejecución del programa, sin embargo, podemos observar la tendencia de tipo exponencial en las gráficas del tamaño de entrada vs Max Pooling en todos los experimentos debido a que el número de operaciones crece rápidamente conforme se aumenta el tamaño de la entrada.

El comportamiento de tipo exponencial aumenta drásticamente a partir del experimento # 10 en el cual el número de operaciones aumenta llevando además a un incremento de las comunicaciones entre el host (CPU) y el device (GPU) por cada operación realizada al tener 2 operaciones de transferencia de datos del HOST al DEVICE y el resultado del DEVICE al HOST por cada kernel extraído en una de las multiplicaciones de matrices. Estas operaciones de transmisión aumentan el tiempo de ejecución del programa conforme aumenta el número de operaciones necesarias para desarrollar el Max Pooling.

En el caso del programa serial este programa no tiene transmisiones entre el host y el device además de que el programa tanto en la generación de los arreglos como muchos de los métodos utilizados se implementaron por medio de la librería de Numpy por lo que estas operaciones permiten calcularse con un buen rendimiento en el CPU proporcionado por el nodo del cluster.

EXPERIMENTOS VERSION SERIAL.

EXPERIMENTOS MAX POOLING SERIAL EN INTEL XEON nodo n49 | p=0, s=2, k=2			
EXPERIMENTOS REALIZADOS CON PROGRAMA SERIAL			
Dim. Matriz cuadrada	max Pooling CPU+GPU	Mult matrices GPU	
Tam Matriz	Tiempo seg	Tiempo (seg)	# Kernels extraidos
4	0.0004	0.0003	4
8	0.0016	0.0013	16
16	0.0057	0.0047	64
32	0.0111	0.0095	256
64	0.0371	0.0316	1024
128	0.1491	0.1277	4096
256	0.5902	0.5024	16384
512	2.3265	1.9867	65536
1024	9.1822	7.8154	262144

FIGURA 28. TABLA DE MAX POOLING SERIAL N49

FIGURA 29. TAMAÑO DE ENTRADA VS MULT. MATRICES SERIAL

EXPERIMENTOS MAX POOLING EN INTEL XEON nodo n48 | p=0, s=2, k=2			
EXPERIMENTOS REALIZADOS CON PROGRAMA SERIAL			
Dim. Matriz cuadrada	max Pooling CPU+GPU	Mult matrices GPU	
Tam Matriz	Tiempo seg	Tiempo (seg)	# Kernels extraidos
4	0.0004	0.0003	4
8	0.0015	0.0012	16
16	0.0056	0.0046	64
32	0.0105	0.0088	256
64	0.0369	0.0314	1024
128	0.1385	0.1175	4096
256	0.6131	0.5213	16384
512	2.2538	1.9178	65536
1024	9.3309	7.9426	262144

FIGURA 30. TABLA DE MAX POOLING SERIAL N48

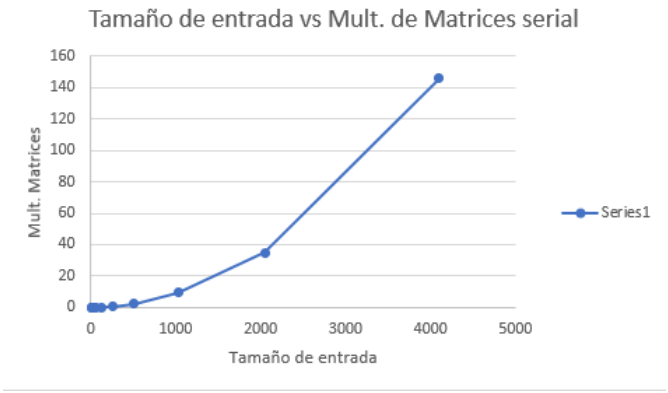

FIGURA 31. TAMAÑO DE ENTRADA VS MULT. MATRICES SERIAL N48

EXPERIMENTOS 1 CON FUNCIÓN DE DISPOSITIVO matmul1

	EXPERIMENTOS INCIALES MAX POOLING EN TESLA T4 | p=0, s=2, k=2			
	EXPERIMENTOS REALIZADOS CON KERNEL matmul1			
	Dim. Matriz cuadrada	max Pooling CPU+GPU	Mult matrices GPU	
# Experimento	Tam Matriz	Tiempo seg	Tiempo (seg)	# Kernels extraidos
1	4	0.8604	0.5267	4
2	8	0.9493	0.6142	16
3	16	0.7241	0.3216	64
4	32	1.3804	0.5721	256
5	64	2.4349	0.6324	1024
6	128	6.8259	0.8343	4096
7	256	25.7239	2.6504	16384
8	512	100.0038	8.511	65536
9	1024	396.6885	33.1336	262144

FIGURA 32. TABLA DE MAX POOLING MATMUL1

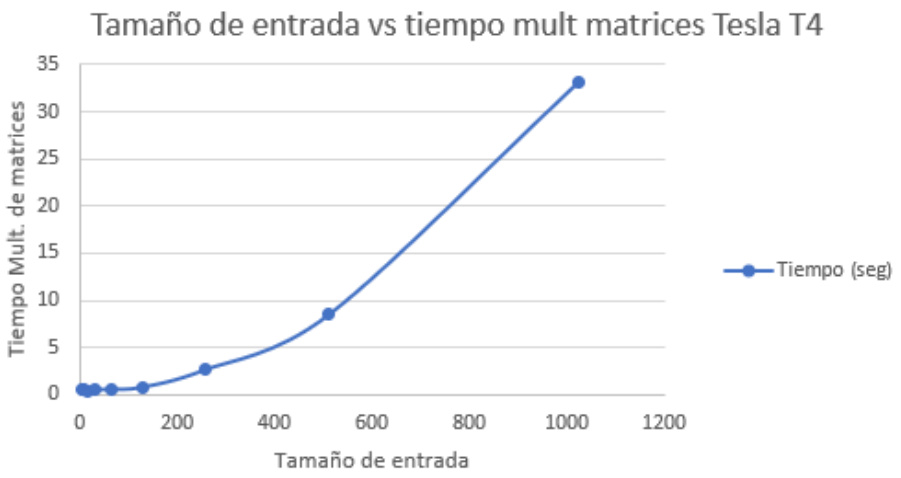

FIGURA 33. TAMAÑO DE ENTRADA VS MULT. MATRICES TESLA T4

	EXPERIMENTOS INCIALES MAX POOLING EN TESLA K20 | p=0, s=2, k=2			
	EXPERIMENTOS REALIZADOS CON KERNEL matmul1			
	Dim. Matriz cuadrada	max Pooling CPU+GPU	Mult matrices GPU	m
# Experimento	Tam Matriz	Tiempo seg	Tiempo (seg)	# Kernels extraidos
1	4	0.7887	0.6692	4
2	8	0.5172	0.379	16
3	16	0.8841	0.6792	64
4	32	0.9211	0.4255	256
5	64	0.523	0.8412	1024
6	128	7.7552	1.4392	4096
7	256	28.5782	3.6102	16384
8	512	108.7938	12.2783	65536
9	1024	425.625	47.0123	262144

FIGURA 34. TABLA DE MAX POOLING MATMUL1 TESLA K20

FIGURA 35. TAMAÑO DE ENTRADA VS TIEMPO MULT. MATRICES K20

EXPERIMENTOS 2 CON FUNCIÓN DE DISPOSITIVO matmul2

	EXPERIMENTOS INCIALES MAX POOLING EN TESLA T4 | p=0, s=2, k=2				
	EXPERIMENTOS REALIZADOS CON KERNEL matmul2				
	Dim. Matriz cuadrada	max Pooling CPU+GPU	Mult matrices GPU	Mult matrices GPU	
# Experimento	Tam Matriz	Tiempo seg	Tiempo (seg)	Tiempo (mseg)	# Kernels extraidos
1	4	1.0051	0.6899	689.9961	4
2	8	0.7328	0.387	387.0563	16
3	16	0.9742	0.5362	536.2935	64
4	32	1.3303	0.6481	648.1357	256
5	64	2.2994	0.5488	548.8156	1024
6	128	7.2513	1.2683	1268.3738	4096
7	256	25.8221	3.1433	3143.3927	16384
8	512	101.9518	11.7131	11713.1211	65536
9	1024	401.6891	44.4151	44415.1466	262144

FIGURA 36. TABLA DE MAX POOLING MATMUL2

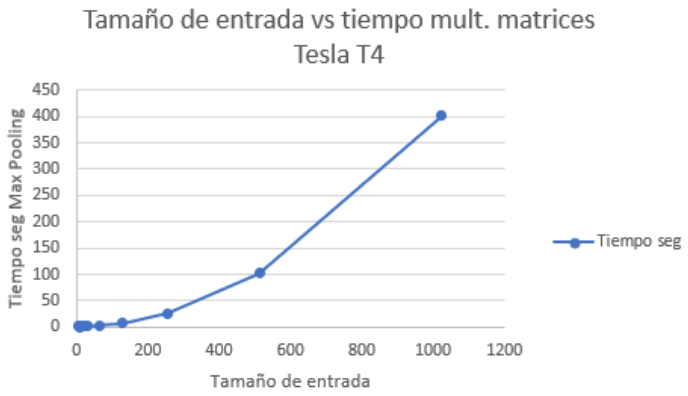

FIGURA 37. TAMAÑO DE ENTRADA VS TIEMPO MULT. MATRICES TESLA T4

	EXPERIMENTOS INCIALES MAX POOLING EN TESLA K20 | p=0, s=2, k=2				
	EXPERIMENTOS REALIZADOS CON KERNEL matmul2				
	Dim. Matriz cuadrada	max Pooling CPU+GPU	Mult matrices GPU	Mult matrices GPU	
# Experimento	Tam Matriz	Tiempo seg	Tiempo (seg)	Tiempo (mseg)	# Kernels extraidos
1	4	0.5324	0.4212	421.2608	4
2	8	0.5056	0.3662	366.2076	16
3	16	0.8829	0.6698	669.8245	64
4	32	0.9529	0.4161	416.1198	256
5	64	2.4773	0.8384	838.4717	1024
6	128	7.6884	1.3608	1360.8802	4096
7	256	28.8468	3.6669	3666.9105	16384
8	512	110.3249	12.2238	12223.889	65536
9	1024	439.9392	48.2	48200.0725	262144

FIGURA 38. TABLA DE MAX POOLING MATMUL22 TESLA K20

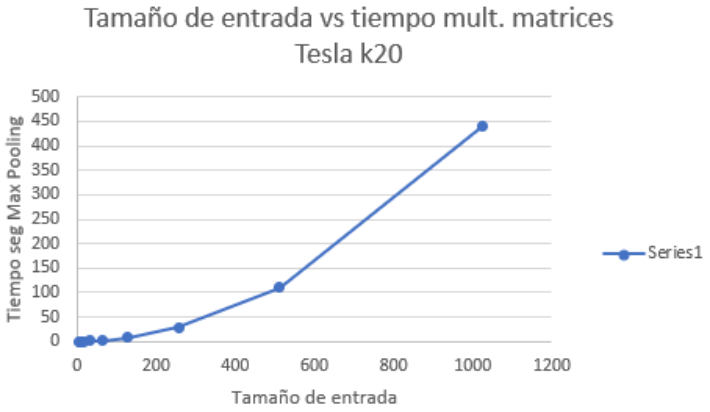

FIGURA 39. TAMAÑO DE ENTRADA VS MULT. MATRICES TESLA K20

EXPERIMENTOS 3 CON FUNCIÓN DE DISPOSITIVO matmul3

	EXPERIMENTOS INCIALES MAX POOLING EN TESLA T4 | p=0, s=2, k=2				
	EXPERIMENTOS REALIZADOS CON KERNEL matmul3				
	Dim. Matriz cua	max Pooling CPU+GPU	Mult matrices GPU	Mult matrices GPU	
# Experimento	Tam Matriz	Tiempo seg	Tiempo (seg)	Tiempo (mseg)	# Kernels extraidos
1	4	0.7539	0.4696	469.6474	4
2	8	1.0906	0.7488	748.8301	16
3	16	1.2864	0.8337	833.7523	64
4	32	1.7779	1.2097	1209.7699	256
5	64	4.5376	3.0816	3086.162	1024
6	128	14.5453	9.8119	9811.9737	4096
7	256	55.9159	37.785	37785.0036	16384
8	512	219.8545	148.9418	148941.8285	65536
9	1024	368.3011	95.3995	95399.5519	262144
10	2048	3498.9617	2376.6623		

FIGURA 40. TABLA DE MAX POOLING MATMUL3 TESLA K20

FIGURA 41. TAMAÑO DE ENTRADA VS TIEMPO MULT. MATRICES TESLA T4

	EXPERIMENTOS INCIALES MAX POOLING EN TESLA K20 | p=0, s=2, k=2				
	EXPERIMENTOS REALIZADOS CON KERNEL matmul3				
	Dim. Matriz cua	max Pooling CPU+GPU	Mult matrices GPU	Mult matrices GPU	
# Experimento	Tam Matriz	Tiempo seg	Tiempo (seg)	Tiempo (mseg)	# Kernels extraidos
1	4	0.8841	0.7787	778.734	4
2	8	0.908	0.788	788.0139	16
3	16	0.9844	0.8005	800.5051	64
4	32	0.9869	0.5973	597.3445	256
5	64	2.2262	0.9289	928.996	1024
6	128	7.2862	2.6096	2609.6061	4096
7	256	26.7946	8.2101	8210.199	16384
8	512	103.3762	30.5071	30507.1751	65536
9	1024	409.4337	120.3164	120316.4607	262144
10	2048				

FIGURA 42. TABLA DE MAX POOLING MATMUL3 TESLA K20

FIGURA 43. TAMAÑO DE ENTRADA VS TIEMPO MULT MATRICES TESLA K20

A continuación, se muestra el análisis de eficiencia de Speedup tomando en cuenta los tiempos obtenidos de Max Pooling a partir del kernel GPU matmul3 y el resultado de Max Pooling serial ambos en segundos.

ANALISIS DE SPEEDUP SERIAL VS PARALELO		
GPU MULT3	SERIAL	SPEEDUP
0.7787	0.0003	0.000513677
0.788	0.0013	0.002030457
0.8005	0.0047	0.00712055
0.5973	0.0095	0.018583626
0.9289	0.0316	0.039939714
2.6096	0.1277	0.057135193
8.2101	0.5024	0.071887066
30.5071	1.9867	0.076260936
120.3164	7.8154	0.076317111

FIGURA 44. ANALISIS DE SPEEDUP

EXPERIMENTOS INCIALES MAX POOLING EN TESLA T4 | p=0, s=2, k=2					
EXPERIMENTOS REALIZADOS CON KERNEL					
		Dim. Matriz cuadrad	max Pooling mult1	max Pooling mult2	max Pooling mult3
# Kernels ex	# Experimen	Tam Matriz	Tiempo (seg) matmul1	Tiempo (seg) matmul2	Tiempo (seg) matmul3
4	1	4	0.8604	1.0051	0.7539
16	2	8	0.9493	0.7328	1.0906
64	3	16	0.7241	0.9742	1.2864
256	4	32	1.3804	1.3303	1.7779
1024	5	64	2.4349	2.2994	4.5376
4096	6	128	6.8259	7.2513	14.5453
16384	7	256	25.7239	25.8221	55.9159
65536	8	512	100.0038	101.9518	219.8545
262144	9	1024	396.6885	401.6891	368.3011

FIGURA 45. COMPARATIVA DE KERNELS GPU UTILIZADOS.

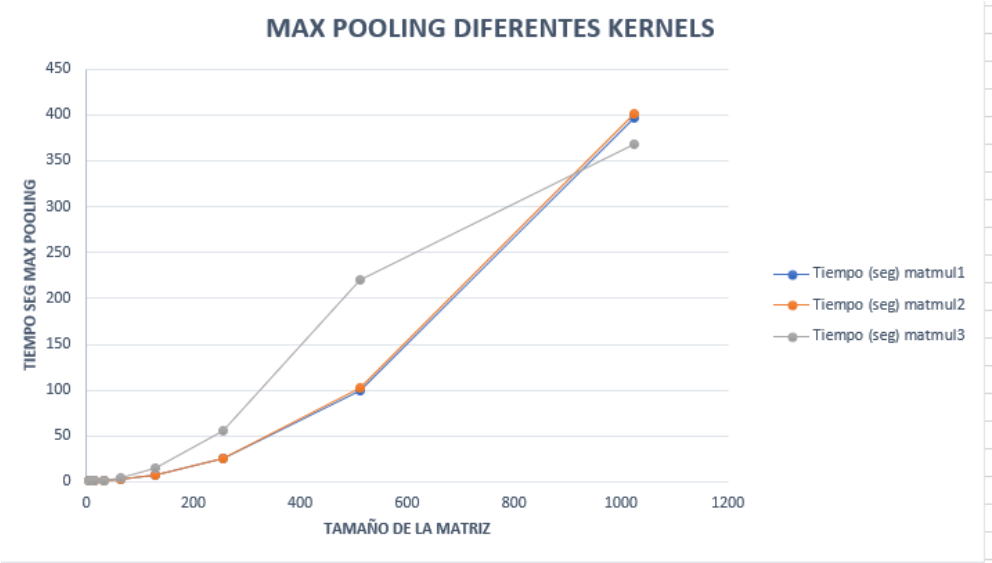

FIGURA 46. MAX POOLING CON DIFERENTES KERNELS GPU.

En los experimentos se utilizó la misma imagen de entrada como una matriz bidimensional de forma aleatoria que van desde el 0 hasta el 9 el cual se probó en 3 distintos kernels GPU. Los primeros dos kernels GPU matmul1 y matmul2 son muy similares los cuales en su diseño varían en como recorren el arreglo y tienen un ciclo for para realizar cada una de las multiplicaciones renglón-columna de la multiplicación de matrices.

El kernel GPU matmul3 muestra un mejor desempeño conforme aumenta el número de las multiplicaciones debido al tamaño de la matriz de entrada. El uso de memoria compartida en esta opción permite el acceso a la memoria más rápida solo después de los registros dentro de la jerarquía de memoria de la GPU lo cual realiza el proceso de multiplicación de matrices más rápidamente, sin embargo esta opción se recomienda utilizar más conforme el tamaño de la matriz aumenta, en este caso las comunicaciones entre el CPU y la GPU son mayores conforme aumenta el tamaño de la matriz de entrada debido a que se requiere de una mayor extracción de kernels para poder aplicar la operación de Max Pooling sobre la imagen de entrada.

EXPERIMENTOS INCIALES MAX POOLING EN GTX 1050

kernel matmul3 Mult Matrices Mem. Compartida			
Dim. Matriz cuadrada	max Pooling CPU+GPU	Mult matrices GPU	
Tam Matriz	Tiempo seg	Tiempo (mseg)	# Kernels extraidos
4	0.7029	0.2028	4
8	0.4015	0.0012	16
16	0.7712	0.003	64
32	1.796	0.0104	256
64	3.8118	0.0389	1024
128	13.7824	0.1525	4096
256	61.8872	0.6106	16384

FIGURA 47. EXPERIMENTOS INICIALES CON GTX 1050

FIGURA 48. TAMAÑO DE ENTRADA VS MAX POOLING

Podemos ver en la gráfica el comportamiento exponencial en la GPU GTX 1050 conforme aumenta el tamaño de entrada. Estas pruebas son más limitadas en lo que se refiere al tamaño de entrada sin embargo se puede ver el comportamiento exponencial aumenta conforme se extraen un mayor número de kernels a partir de aplicar la operación de Max Pooling sobre la imagen de entrada. Recordemos que cada kernel extraída es una multiplicación de matrices que debe realizarse además de la previa comunicación entre el CPU y la GPU y el regreso de la variable resultado una vez hecha la multiplicación de regreso de la GPU al CPU para obtener el valor mayor y posteriormente agregarlo a la variable imagenFinal la cual ira conformando la imagen con el Max Pooling. Este proceso se hace en el CPU el cual esta parte del programa extrae por medio de una ventana previamente establecida una parte de la imagen original para formar el mapa de características posteriormente. Toda esta parte del programa se ejecuta desde el CPU por lo que la programación con Numba CUDA se dice que funciona como un sistema heterogéneo entre el CPU y la GPU para resolver la parte del programa que sea paralelizable, así como procesos que se consideren computacionalmente intensivos cuya complejidad sea alta y sea más adecuado resolver por medio de técnicas de CAR.

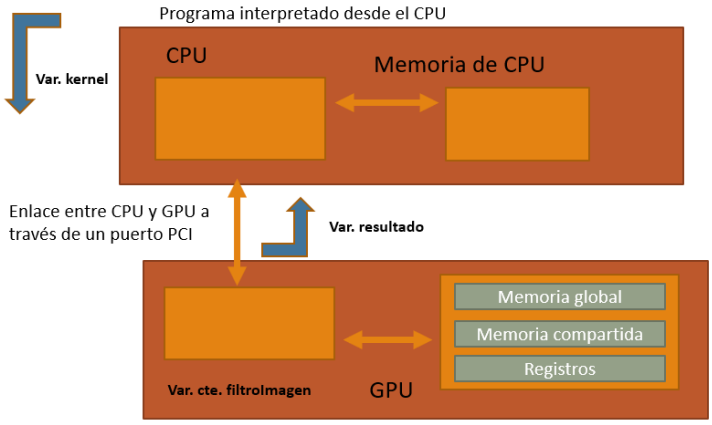

FIGURA 49. TRANSFERENCIA DE MEMORIA ENTRE HOST Y DEVICE

Las distintas GPUs aunque tienen diferentes características podemos observar el mismo comportamiento en los experimentos. Para el caso de las GPUs para cluster permiten un mejor manejo de la temperatura y un rendimiento de energía lo cual permite realizar experimentos con un tiempo más prolongado para probar imágenes de entrada más grandes en sus dimensiones.

En los siguientes experimentos se muestra la comparación de kernels en la GPU Tesla T4 para los 3 kernels GPU pero con la declaración de la variable filtroImagen en el kernel GPU como una constante para obtener una menor transferencia de datos entre el CPU y la GPU.

EXPERIMENTOS MAX POOLING EN TESLA T4 | p=0, s=2, k=2 filtro_imagen transferido en kernel GPU.					
EXPERIMENTOS REALIZADOS CON KERNEL					
		Dim. Matriz cuadrac	max Pooling mult1	max Pooling mult2	max Pooling mult3
# Kernels ex	# Experimen	Tam Matriz	Tiempo (seg) matmul1	Tiempo (seg) matmul2	Tiempo (seg) matmul3
4	1	4	0.8604	1.0051	0.9028
16	2	8	0.9493	0.7328	1.0999
64	3	16	0.7241	0.9742	1.092
256	4	32	1.3804	1.3303	2.1366
1024	5	64	2.4349	2.2994	4.6572
4096	6	128	6.8259	7.2513	16.5
16384	7	256	25.7239	25.8221	62.0277
65536	8	512	100.0038	101.9518	245.8479
262144	9	1024	396.6885	401.6891	982.2111
1048576	10	2048	1633.4812	1611.7116	3916.0432

FIGURA 50. EXPERIMENTOS MAX POOLING TESLA T4 CON VARIABLE FILTROIMAGEN TRANSFERIDA

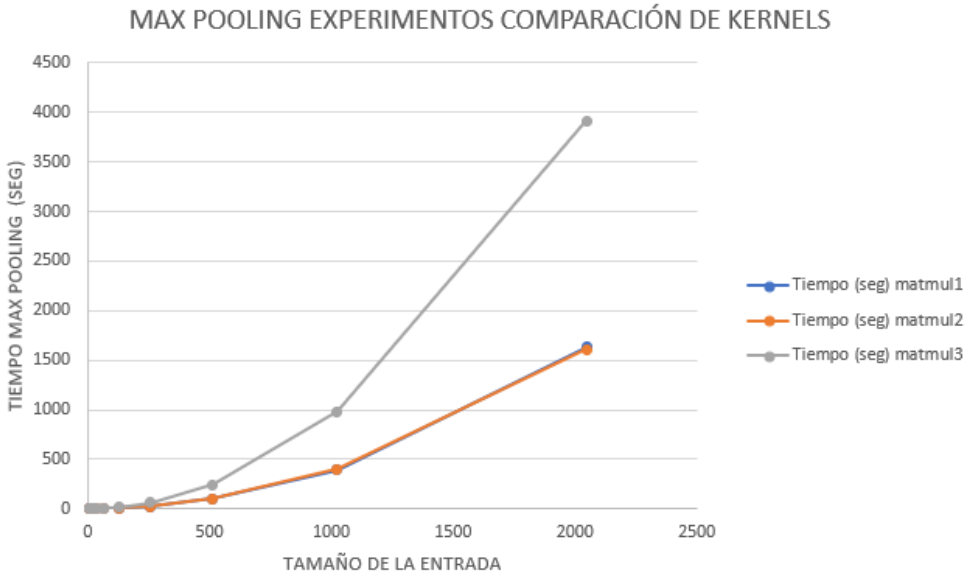

FIGURA 51. GRAFICA DE COMPARACIÓN DE KERNELS GPU DE MAX POOLING CON VARIABLE FILTROIMAGEN TRANSFERIDA

EXPERIMENTOS MAX POOLING EN TESLA T4 | p=0, s=2, k=2 filtro_imagen cte en kernel GPU.					
EXPERIMENTOS REALIZADOS CON KERNEL					
		Dim. Matriz cuadrad	max Pooling mult1	max Pooling mult2	max Pooling mult3
# Kernels ex	# Experimen	Tam Matriz	Tiempo (seg) matmul1	Tiempo (seg) matmul2	Tiempo (seg) matmul3
4	1	4	0.8183	0.5635	0.7539
16	2	8	0.9084	0.6346	1.0906
64	3	16	0.8573	0.9387	1.2864
256	4	32	1.1821	1.1628	1.7779
1024	5	64	1.7729	2.0452	4.5376
4096	6	128	5.2885	5.75	14.5453
16384	7	256	19.2966	19.5509	55.9159
65536	8	512	74.5182	74.3223	219.8545
262144	9	1024	297.8617	296.9865	368.3011
1048576	10	2048	1191.0862	1166.7998	3548.4262

FIGURA 52. EXPERIMENTOS MAX POOLING TESLA T4 CON VARIABLE FILTROIMAGEN CONSTANTE.

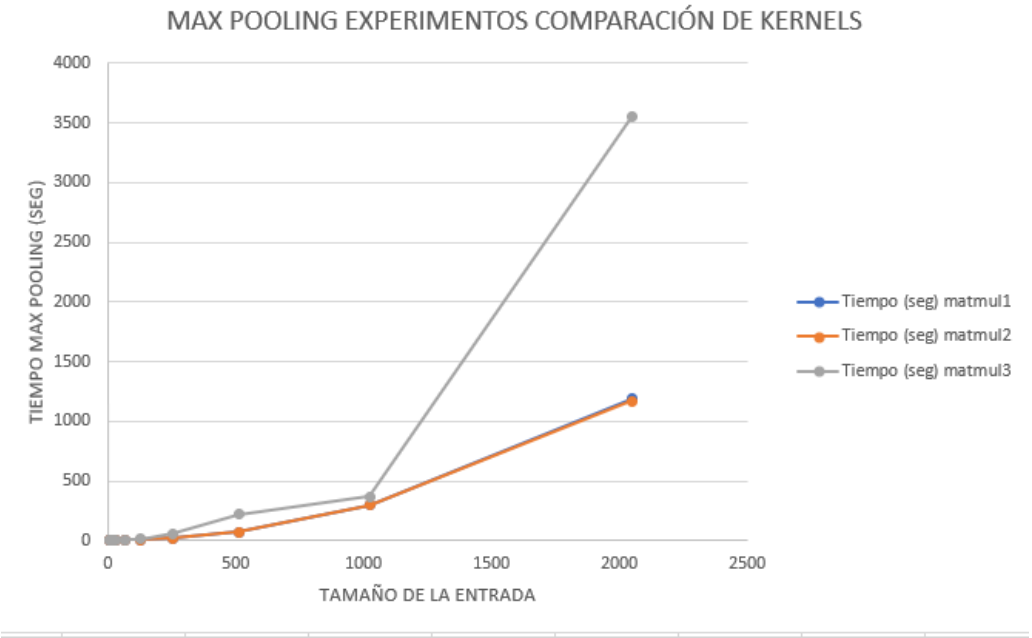

FIGURA 53. GRAFICA DE COMPARACIÓN DE KERNELS GPU DE MAX POOLING CON VARIABLE FILTROIMAGEN CONSTANTE

Se observa una pendiente con tendencia exponencial pero menos pronunciada en el caso donde se declaran la variable filtroImagen como constante. Los experimentos muestran que esta mejora nominal que hace que haya una menor transferencia de datos entre las unidades de procesamiento hacen que sea una mejora considerable en el rendimiento del tiempo de procesamiento de la operación de Max Pooling.

CONCLUSIONES

El entrenamiento de redes neuronales debido a las capas de convolución y pooling tiene diferentes operaciones de algebra lineal las cuales al contener ciclos y sin dependencia de datos tienen características estos procesos que les permiten ser paralelizables debido a la independencia de procesos. Las técnicas basadas en memoria compartida tales como hilos permite que se haga una paralelización que puede probarse en varios procesadores.

La operación de Max Pooling actualmente es uno de los componentes fundamentales sobre los cuales funcionan las redes convolucionales. En el caso de imágenes de gran tamaño o en el caso de que se procesan sobre conjuntos de datos de gran información para poder acelerar los tiempos de entrenamiento de las redes convolucionales siendo el Max Pooling una operación que se encuentra en varias arquitecturas de redes neuronales convolucionales además de producir invarianza a las traslaciones, además de realizar la extracción de características con una resolución baja siendo esta una operación de muestreo sobre la imagen original.

El entrenamiento de redes neuronales debido a las capas de convolución y Pooling tienen una gran cantidad de ciclos en ocasiones anidados los cuales son costosos desde un punto de vista algorítmico. Las técnicas de cómputo de alto rendimiento permiten distribuir las tareas de forma equitativa entre los procesadores que se tienen disponibles mejorando los tiempos de entrenamiento.

El uso de hardware especializado tal como el uso de las GPU las cuales al manejar un alto grado de paralelismo debido a la gran cantidad de núcleos con los que cuentan para el caso de operaciones que contienen ciclos permiten un tiempo de entrenamiento menor y reducir la carga por núcleo dividiendo la tarea en operaciones más simples para las redes convolucionales debido a la gran cantidad de operaciones presentes, en los ciclos utilizados en las operaciones de Pooling y convolución es una opción que permite la paralelización a través de esta técnica.

El kernel matmul1 con ciclos for anidados ha probado ser un problema con una complejidad $O(n^3)$ que se considera como un problema en 2D por medio de la sintaxis de CUDA para manejarse a través de una rejilla bidimensional, mientras que el otro programa funciona por medio de un if y se basa en estructuras de decisión para identificar el kernel que debe ser multiplicado en base a condiciones que cumplan con sus dimensiones previamente definidas.

Los experimentos han mostrado que el kernel matmul3 de memoria compartida y descomposición de matrices en bloques es efectivo en el caso de multiplicación de matrices de dimensiones grandes. Debido a que las multiplicaciones que se realizan son de pequeñas dimensiones en estos experimentos el kernel matmul3 ha sido menos efectivo que matmul1 y matmul2 debido a que conforme aumenta el tamaño de la matriz se incrementa el número de multiplicaciones de dimensión constante y comunicaciones entre el CPU y la GPU para generar la imagen obtenida con la operación de Max Pooling.

En el caso de la aceleración por medio de GPU la identificación de la región paralela como proceso computacionalmente intensivo es una parte fundamental en el diseño del programa siendo en este caso particular la multiplicación de matrices un problema que puede ser paralelizable para el cual es posible aplicar técnicas de CAR para esta operación. Este problema al requerir de una gran cantidad de operaciones necesita que la multiplicación de matrices la cual se obtiene a partir de

cada uno de los kernels extraídos a partir de la imagen de entrada sea llamada numerosas veces por lo que al ser esta una función de dispositivo (GPU) y enviar los valores obtenidos en la variable resultado de regreso al host las comunicaciones entre el CPU y la GPU pueden llegar a ser un factor que incremente los tiempos de ejecución de este programa.

Las constantes comunicaciones entre el host y el device son un factor que debe considerarse para evitar una disminución del rendimiento del programa, así como la definición de funciones definidas por el usuario para tener un diseño de kernel GPU, la declaración de constantes para evitar una mayor transferencia de variables, en la medida de lo posible el uso de instrucciones lógicas tales como if debido a que los núcleos CUDA al no tener predicciones en la ramificación de instrucciones lógicas pueden llegar a presentar tiempos más prolongados en su procesamiento.

El ajuste de hiperparámetros para probar la operación de Max Pooling en distintas condiciones para poder observar su funcionamiento con diferentes valores de hiperparámetros fue de utilidad para los experimentos y tener una forma de reproducir estos en distintos GPUs. Estas condiciones permiten verificar los experimentos no solo por sus tiempos sino por como poder obtener un resultado esperado tal como en la reducción de las dimensiones del mapa de características a partir de la imagen original. Los hiperparámetros también permiten que se establezcan condiciones en la operación de Max Pooling que determinan el comportamiento de la operación para que de esta forma se puedan lograr que los experimentos sean reproducibles en distintos GPUs, los cuales pueden mostrar diferencias en su velocidad de procesamiento y aportar diferentes recursos y características dependiendo del modelo y arquitectura de la GPU correspondiente pero podemos ver un comportamiento de crecimiento exponencial al momento de realizar los experimentos en las distintas GPUs.

La operación de Max Pooling tiene una gran cantidad de ciclos en ocasiones anidados, los cuales son operaciones consideradas computacionalmente intensivos, por lo que las técnicas de cómputo de alto rendimiento permiten distribuir las tareas de la región paralela entre los procesadores que se tienen disponibles. En el caso de este trabajo se buscó que la multiplicación de matrices aplique técnicas de CAR por medio de la GPU. La aceleración de este proceso de multiplicación ha permitido que se realice la multiplicación de matrices en un tiempo satisfactorio, sin embargo las constantes comunicaciones entre el CPU y la GPU han llevado a que conforme aumenta el tamaño de la imagen y las dimensiones de la multiplicación de matrices se mantengan constantes al obtener la operación de Max Pooling existen un tiempo de procesamiento mayor debido a cada vez un mayor número de multiplicaciones y de comunicaciones entre el CPU y el GPU debido a que este programa funciona como un sistema heterogéneo. Este proceso se realiza por cada kernel extraído del mapa de características original para formar la imagen final en el cual la dimensión de la multiplicación de matrices se mantiene constante por cada kernel extraído a lo largo del proceso de Max Pooling.

El puerto PCI se encuentra en la placa base como medio de comunicación en este caso de la CPU y la GPU. Es un bus en paralelo para la transferencia de datos por medio de un bus paralelo ofreciendo un ancho de banda mayor. Cada puerto tiene una tasa de transferencia máxima de archivos dependiendo de las características del puerto en la tarjeta madre.

El lanzamiento del kernel GPU debe de tomarse el ámbito de variables en la GPU que se utilizan debido a que la transferencia de datos entre las unidades de procesamiento es una operación costosa por lo que se debe de identificar las variables que intervienen en la operación que es paralelizable en la GPU y el resto de la operación de Max Pooling que se ejecuta en el CPU.

El ajuste de parámetros permite que se establezcan condiciones en la operación de la capa de Max Pooling para que influyen en el comportamiento de la operación, así como lograr que los experimentos sean reproducibles en distintos GPUs aunque tengan distintas características cada una de las GPUs podemos observar el mismo comportamiento de tipo exponencial en los experimentos conforme aumenta el tamaño de entrada como matriz cuadrada con valores aleatorios que van desde el 0 hasta el 9.

Bibliografía y referencias

Bibliografía

Aggarwal, C. C. (2018). *Neural Networks and Deep Learning A Textbook.* Yorktown Heights, NY, USA: Springer .

Goodfellow et al., G. e. (2016). *Deep Learning.* Cambridge, MA: MIT Press.

Calin, O. (2020). *Deep Learning Architectures A mathematical Approach.* Ypsilanti, MI, USA: Springer.

Edward, R. (2022). *Inside Deep Learning Math, Algoritthms, Models.* Shelter Island, NY: Manning Publications.

Pacheco, P. S. (1997) Parallel programming with MPI. San Francisco USA: Elsevier

Pacheco, P. S. (2011). *An Introduction to Parallel Programming.* Burlington, USA: Elsevier.

Pajankar, A. (2017). *Raspberry Pi Supercomputing and Scientific Programming.* Nashik, Maharasshtra, India: Apress.

Passi, S. T. (2019). *PyTorch Deep Learning Hands-On.* Birmingham, UK: Packt Publishing.

Stevens et al. Deep Learning with PyTorch (2020). Shelter Island, NY : Manning Publications. pp. 202-207, pp. 102-109.

Sheldon, R. (2018). *A First Course in Probability 10ed.* Boston, USA: Pearson.

Subramanian, V. (2018). *Deep Learning with PyTorch.* Birmingham, UK: Pack Publishing Ltd.

Schimdt et al., (2018). *Parallel Programming. Concepts and Practice.* Cambridge, MA, USA: Elsevier.

Barlas, G. (2015). *Multicore and GPU programming.* Waltham, USA: Elsevier.

Papa, J. (2021). *PyTorch Pocket Reference.* Sebastopol, CA : O' Reilly Media.

Runger, T. R. (2007). *Parallel Programming. For multicore and cluster systems.* Berlin Heidelberg: Springer.

Zamora, R. R. (2021). *Parallel and High Performance Computing.* Shelter Island, NY: Manning Publications.

Cheng et al., C. e. (2014). *Professional CUDA C Programming.* Indianapolis, Indiana. USA: Wrox, John Wiley & Sons. pp. 23-36, pp. 43-48, pp. 203-214.

Cook, S. (2013). *CUDA programming. A developer's guide to parallel computing with GPU's.* Wyman Street, Waltham. USA: Morgan Kauffman. Elsevier Inc.

Kirk B. David, H. W.-m. (2013). *Programming Massively Parallel Processors 2ed.* Wyman Street, Wathman, USA: Morgan Kauffman.

Wilt, N. (2013). *The CUDA handbook.* Crawfordsville, Indiana, USA: Addison Wesley, Pearson.

Tuomanen Brian. (2018). Hands-On GPU Programming with Python and CUDA. UK. Birmigham: Pack Publishing. pp. 10-14.

Deitel Paul & Deitel Harvey. (2020). Intro to Python for Computer Science and Data Science. USA Pearson.

Lipschutz Seymor. (1989). 3000 solved problems in Linear Algebra. Schaum's Solved problems Series. USA. Mc Graw Hill. pp. 28-40.

Larson E. Roland & Edwards H. Bruce. (2015) Introducción al Algebra Lineal. México. Limusa.

Calin Ovidiu. (2020). Deep Learning Architectures. A Mathematical Approach. USA. Michigan. Springer. pp. 507-515, pp. 517-528.

Wilkinson Barry & Allen Michael. (2005) Parallel Programming. Techniques and Applications using Networked workstations and parallel computers 2 ed. USA. Pearson.

Géron Aurélien . (2017) Hands-On Machine Learning with Scikit-Learn & Tensorflow. USA. O'Reilly

Ashish Ranjan Jha. (2021) Mastering PyTorch. UK. Birmingham. Packt Publishing.

Kneusel T. Ronald. (2022) Math for Deep Learning. What you need to know to understand neural networks. San Francisco, CA. USA. No starch press. pp. 232-238.

Kneusel Ronald T. (2021) Practical Deep Learning. A Python – Based Introduction. San Francisco, CA. USA. No starch press. pp. 289-292, pp. 299-301.

Ekman Magnus. (2022) Learning Deep Learning. Theory and Practice of Neural Networks, Computer Vision, Natural Language Processing, and Transformers using Tensorflow. USA. Addison-Wesley. pp. 26-29, pp. 171-179.

Raschka Sebastian & mirijalili Vahid. (2019) Python Machine Learning. Machine Learning and Deep Learning with Python, scikit-learn and Tensorflow 2. 3 ed. Birmingham, UK. Packt Publishing. pp 393-400, pp. 530-532.

Nikhil Ketkar & Jojo Moonayil. (2021) Deep Learning with Python. Learn Best Practices of Deep Learning Models with PyTorch, 2 ed. Apress.

Chollet Francois. (2021) Deep Learning with Python, 2 ed. Shelter Island, NY. Manning Publications. pp. 122-133, pp. 201-212, pp. 26-37.

Bishop M. Christopher. (2006) Pattern recognition and machine learning. Cambridge, U.K. Springer. pp. 33-38.

Friedman et al. (2009) The elements of statistical learning. Data Mining, Inference, and Prediction, 2 ed. Stanford CA, USA. Springer. pp. 411-420.

Marsland Stephen. (2015) Machine learning An Algorithmic Perpective, 2 ed. Ashhurst New Zealand. CRC Press.

Stallings William. (2005) Organización y arquitectura de computadoras, 7 ed. Madrid. España. Pearson. pp. 440-444.

Deisenroth et al. (2020) Mathematics for Machine Learning. UK. Cambridge University Press. pp. 8-16.

Pointer Ian. (2019) Programming PyTorch for Deep Learning, Creating and Deploying Deep Learning Applications. Sebastopol, CA. USA. O'Reilly.

Saleh Hyatt. (2020) The Deep learning with PyTorch workshop. Build Deep neural network and artifical intelligence applications with PyTorch. Birmingham, UK. Packt Publishing.

Murach Joel & Urban Michael. (2016) murach's Python programming. USA. Mike Murach & Associates.

Mano Morris & Kime Charles R. (2005) Fundamentos de diseño lógico y de computadoras, 3 ed. Madrid. España. Pearson. pp. 474-476.

Theodoridis Sergios. (2015) Machine Learning A Bayesian and Optimization Perspective. London, UK. Elsevier.

Cohen X Mike. (2022) Practical linear algebra for Data Science: From Core Concepts to Applications using Python. USA. O'Reilly. *Stevens et al.*

Moolayil Jojo & Ketkar Nikhil. Deep learning with Python Learn best practices of Deep Learning Model with PyTorch. 2 ed. (2021)Bangalore, India. Apress. pp. 77-90, pp. 206-214.

Glassner Andrew. Deep learning A visual approach. (2021) San Francisco , CA. USA No starch press. pp. 449-456

Raff Edward. Inside Deep learning Math, algorithms, models. (2022). Shelter Island, NY. Manning Publications. pp. 102-105.

Cormen et al. Introduction to algorithms, 3 ed. (2009) Cambridge Massachusetts, USA. MIT Press.

Johansson Robert. Numerical Python Scientific computing and data science applications with Numpy, Scipy and Matplotlib 2 ed. (2019) Chiba, Japan. Apress. pp. 57-70.

REFERENCIAS WEB

REPOSITORIOS

GitHub - bentrevett/pytorch-image-classification: Tutorials on how to implement a few key architectures for image classification using PyTorch and TorchVision.

MLDawn-Projects/MRI-Brain-Tumor-Detecor.ipynb at main · MLDawn/MLDawn-Projects · GitHub

GitHub - mrdbourke/pytorch-deep-learning: Materials for the Learn PyTorch for Deep Learning: Zero to Mastery course.

GitHub - patrickloeber/pytorchTutorial: PyTorch Tutorials from my YouTube channel

CONVERTIDOR NIFTI

https://github.com/alexlaurence/NIfTI-Image-Converter

INSTALACION DE ANACONDA EN UBUNTU SERVER

https://medium.com/@alexgo1/seamless-remote-ipython-kernel-debugging-with-spyder-ide-d34a0a44baaf

https://medium.com/@halmubarak/connecting-spyder-ide-to-a-remote-ipython-kernel-25a322f2b2be

https://www.hostinger.com/tutorials/how-to-install-anaconda-on-ubuntu/

https://www.digitalocean.com/community/tutorials/how-to-install-the-anaconda-python-distribution-on-ubuntu-20-04

https://www.digitalocean.com/community/tutorials/initial-server-setup-with-ubuntu-20-04

https://www.youtube.com/watch?v=tu_H_SHZL5U

LEY DE AMDAHL

https://www.geeksforgeeks.org/computer-organization-amdahls-law-and-its-proof/

https://studylib.net/doc/25365457/amdahl%E2%80%99s-law-explained

https://cvw.cac.cornell.edu/optimization/doitfaster

https://es.wikipedia.org/wiki/Ley_de_Amdahl

PAGINAS Y TUTORIALES PARA MPI4PY

https://www.bu.edu/pasi/files/2011/01/Lisandro-Dalcin-mpi4py.pdf

https://materials.jeremybejarano.com/MPIwithPython/collectiveCom.html

https://www.learnpdc.org/RaspberryPi-mpi4py/index.html

https://mpi4py.readthedocs.io/en/stable/tutorial.html

https://rabernat.github.io/research_computing/parallel-programming-with-mpi-for-python.html

http://sporadic.stanford.edu/thematic_tutorials/numerical_sage/mpi4py.html

https://github.com/erdc/mpi4py/blob/master/docs/source/usrman/tutorial.rst+

https://research.computing.yale.edu/sites/default/files/files/mpi4py.pdf

https://www.howtoforge.com/tutorial/distributed-parallel-programming-python-mpi4py/

https://www.learnpdc.org/RaspberryPi-mpi4py/04PointToPoint/05messagePassing.html

https://www.nesi.org.nz/sites/default/files/mpi-in-python.pdf

https://www.open-mpi.org/doc/v1.6/man3/MPI_Scatter.3.php

https://nyu-cds.github.io/python-mpi/05-collectives/

https://materials.jeremybejarano.com/MPIwithPython/index.html#

REFERENCIAS DE MAX POOLING Y MUTLIPLICACIÓN DE MATRICES

https://blog.paperspace.com/pooling-in-convolutional-neural-networks/

https://www.kaggle.com/learn/computer-vision

https://datascience.stackexchange.com/questions/19455/how-can-you-decide-the-window-size-on-a-pooling-layer

https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/

https://www.geeksforgeeks.org/cnn-introduction-to-padding/

https://programmathically.com/what-is-pooling-in-a-convolutional-neural-network-cnn-pooling-layers-explained/

https://datascience.stackexchange.com/questions/67334/whats-the-purpose-of-padding-with-maxpooling

https://www.bouvet.no/bouvet-deler/understanding-convolutional-neural-networks-part-1

https://d2l.ai/chapter_convolutional-neural-networks/pooling.html

https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/

https://www.kaggle.com/questions-and-answers/59502

http://deeplearning.stanford.edu/tutorial/supervised/Pooling/

https://numpy.org/doc/stable/reference/generated/numpy.ndarray.max.html

https://stackoverflow.com/questions/33569668/numpy-max-vs-amax-vs-maximum

https://www.sharpsightlabs.com/blog/numpy-maximum/

https://sparkbyexamples.com/numpy/calculate-maximum-numpy-array/

https://www.tutorialsandyou.com/python/numpy-max-in-python-39.html

https://medium.com/aiguys/pooling-layers-in-neural-nets-and-their-variants-f6129fc4628b

https://blog.paperspace.com/pooling-in-convolutional-neural-networks/

https://medium.com/analytics-vidhya/matrix-multiplication-in-cuda-a-simple-guide-bab44bc1f8ab

https://www.youtube.com/watch?v=uihBwtPIBxM

https://medium.com/towards-data-science/tensorflow-for-computer-vision-how-to-implement-pooling-from-scratch-in-python-379cd7717de9

https://medium.com/analytics-vidhya/simple-cnn-using-numpy-part-iii-relu-max-pooling-softmax-c03a3377eaf2

https://medium.com/rapids-ai/the-life-of-a-numba-kernel-a-compilation-pipeline-taking-user-defined-functions-in-python-to-cuda-71cc39b77625

https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/

https://www.youtube.com/watch?v=sSWCeZVIN8w

https://hardzone.es/tutoriales/componentes/puertos-pcie-tipos-placas-base/

http://vision.stanford.edu/teaching/cs131_fall1920/slides/20_deep_cnns.pdf

https://dingyan89.medium.com/calculating-parameters-of-convolutional-and-fully-connected-layers-with-keras-186590df36c6

http://www.datasciencecourse.org/notes/matrices/

https://www.kdnuggets.com/2021/05/essential-linear-algebra-data-science-machine-learning.html

https://towardsdatascience.com/convolutional-neural-network-17fb77e76c05

https://cs.colby.edu/courses/F19/cs343/lectures/lecture11/Lecture11Slides.pdf

https://courses.cs.washington.edu/courses/cse416/22su/lectures/10/lecture_10.pdf

https://blog.paperspace.com/pooling-in-convolutional-neural-networks/

https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

https://www.analyticsvidhya.com/blog/2017/05/comprehensive-guide-to-linear-algebra/

https://towardsdatascience.com/mathematics-for-data-science-e53939ee8306

https://www.javatpoint.com/linear-algebra-for-machine-learning

https://machinelearningmastery.com/gentle-introduction-linear-algebra/

REFERENCIAS DE CONVOLUCIÓN

https://siboehm.com/articles/22/Fast-MMM-on-CPU

https://www.geeksforgeeks.org/how-to-perform-faster-convolutions-using-fast-fourier-transformfft-in-python/

https://www.geeksforgeeks.org/introduction-to-convolutions-using-python/?ref=rp

https://medium.com/analytics-vidhya/fast-cnn-substitution-of-convolution-layers-with-fft-layers-a9ed3bfdc99a

https://medium.com/mlearning-ai/fast-fourier-convolution-a-detailed-view-a5149aae36c4

https://hackaday.com/2021/07/21/what-exactly-is-a-gaussian-blur/

https://www.kaggle.com/general/225375

https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215

REFERENCIAS INTEROPERABILIDAD CUDA, CUPY Y NUMBA

https://leimao.github.io/blog/CUDA-Occupancy-Calculation/

https://erangad.medium.com/1d-2d-and-3d-thread-allocation-for-loops-in-cuda-e0f908537a52

https://shiyan.medium.com/some-cuda-concepts-explained-12ecc390d10f

https://stackoverflow.com/questions/60027446/how-to-run-python-on-gpu-with-cupy

https://carpentries-incubator.github.io/lesson-gpu-programming/02-cupy/index.html

https://www.geeksforgeeks.org/python-cupy/

https://docs.cupy.dev/en/stable/user_guide/memory.html

https://cvw.cac.cornell.edu/GPUarch/computecap

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

https://medium.com/data-analysis-center/a-practical-approach-to-speed-up-python-code-numba-numpy-cupy-65ab52526ad4

https://nyu-cds.github.io/python-gpu/01-introduction/

https://stackoverflow.com/questions/7542957/is-python-capable-of-running-on-multiple-cores

https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

https://stackoverflow.com/questions/26998223/what-is-the-difference-between-contiguous-and-non-contiguous-arrays

https://stackoverflow.com/questions/8914625/how-to-set-number-of-processes-in-mpi4py

https://stackoverflow.com/questions/47262190/how-do-i-find-the-number-of-cores-available-to-mpi4py

https://waterprogramming.wordpress.com/2021/11/10/easy-batch-parallelization-of-code-in-any-language-using-mpi4py/

https://docs.ycrc.yale.edu/clusters-at-yale/guides/mpi4py/

https://carpentries-incubator.github.io/gpu-speedups/aio/index.html

https://numba.discourse.group/t/how-do-i-use-global-memory-with-numba-cuda/828

https://stackoverflow.com/questions/63311574/in-numba-how-to-copy-an-array-into-constant-memory-when-targeting-cuda

https://tbetcke.github.io/hpc_lecture_notes/numba_cuda.html

https://docs.cupy.dev/en/stable/user_guide/interoperability.html

https://leimao.github.io/blog/CUDA-Matrix-Multiplication/

https://leimao.github.io/blog/Math-Bound-VS-Memory-Bound-Operations/

https://www.kaggle.com/general/274291

CONFIGURACION DE GPU

https://stackoverflow.com/questions/4391162/cuda-determining-threads-per-block-blocks-per-grid

https://stackoverflow.com/questions/66459855/cuda-tiled-matrix-multiplication-explanation

https://shiyan.medium.com/some-cuda-concepts-explained-12ecc390d10f

https://thedatafrog.com/en/articles/cuda-kernel-python/

https://erangad.medium.com/1d-2d-and-3d-thread-allocation-for-loops-in-cuda-e0f908537a52

https://streamhpc.com/blog/2017-01-24/many-threads-can-run-gpu/

https://www.tutorialspoint.com/cuda/cuda_matrix_multiplication.htm

https://kdm.icm.edu.pl/Tutorials/GPU-intro/introduction.en/

https://stackoverflow.com/questions/32621364/pycuda-best-way-of-calling-kernel-multiple-times

https://medium.com/rapids-ai/run-your-python-user-defined-functions-in-native-cuda-kernels-with-rapids-cudf-57477dd94fb3

https://medium.com/rapids-ai/the-life-of-a-numba-kernel-a-compilation-pipeline-taking-user-defined-functions-in-python-to-cuda-71cc39b77625

https://medium.com/@gbadahamza18/exploit-your-gpu-by-parallelizing-your-codes-using-python-2dd8e2215aa8

https://www.geeksforgeeks.org/python-cupy/

https://stackoverflow.com/questions/63064638/numba-cuda-slower-than-parallel-cpu-even-for-giant-matrices

https://stackoverflow.com/questions/69509859/nuhttps://towardsdatascience.com/speed-up-your-algorithms-part-2-numba-293e554c5cc1mba-cuda-computation-seems-to-be-slower-than-sequential-run-did-i-do-obvious

https://immune-technology-institute.medium.com/optimize-your-code-with-numba-cpu-cc4029ebbb00

https://www.pugetsystems.com/labs/articles/pci-express-4-0-vs-3-0-video-card-performance-1982/

https://www.pugetsystems.com/parts/Motherboard/Gigabyte-X570-AORUS-Ultra-13233/

https://www.gigabyte.com/us/Motherboard/X570-AORUS-ULTRA-rev-11-12#kf

https://www.muycomputer.com/2017/11/09/pci-express-guia/

https://www.muycomputerpro.com/2017/11/01/pci-express-gen-4-0

https://linustechtips.com/topic/1321587-how-can-i-test-the-speed-of-my-gpus-pcie-slot-connection/

https://www.geeknetic.es/Guia/2060/Los-10-Mejores-Benchmarks-para-GPU.html

EJEMPLOS DE CLASIFICACIÓN

https://www.youtube.com/watch?v=5lgrlddp-98

https://github.com/akd6203/brain-tumor-detection

https://d2l.ai/chapter_convolutional-neural-networks/index.html

https://www.youtube.com/watch?v=RQIAmvElu1g&list=PLTKMiZHVd_2KJtIXOW0zFhFfBaJJilH51&index=69

https://github.com/rasbt/stat453-deep-learning-ss21/tree/main/L09/code/custom-dataloader

https://medium.com/analytics-vidhya/creating-a-custom-dataset-and-dataloader-in-pytorch-76f210a1df5d

https://towardsdatascience.com/understanding-pytorch-with-an-example-a-step-by-step-tutorial-81fc5f8c4e8e

https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

COMANDOS DE NVIDIA-SMI

https://medium.com/analytics-vidhya/explained-output-of-nvidia-smi-utility-fc4fbee3b124

https://medium.com/analytics-vidhya/crux-of-gpu-28fe7d37dd28

https://manpages.ubuntu.com/manpages/xenial/man1/alt-nvidia-340-smi.1.html

https://developer.nvidia.com/nvidia-system-management-interface

https://videocardz.com/35463/final-specification-of-tesla-k20-with-kepler-gk110-gpu

https://www.nvidia.com/content/tesla/pdf/nv-ds-teslak-family-jul2012-lr.pdf

https://blogs.nvidia.com/blog/2019/01/23/tesla-t4-powers-virtual-workstations/

microway.com/hpc-tech-tips/nvidia-smi_control-your-gpus/

https://enterprise-support.nvidia.com/s/article/Useful-nvidia-smi-Queries-2

PAGINA DE MANEJO DE HILOS Y MANEJO DE PYTHON NUMBA

https://www.geeksforgeeks.org/find-the-memory-size-of-a-numpy-array/

https://numba.pydata.org/numba-doc/dev/cuda/examples.html

https://python-course.eu/python-tutorial/formatted-output.php

https://www.tutorialspoint.com/python/python_multithreading.htm

https://stackoverflow.com/questions/10613131/how-to-access-list-elements

http://www.adeveloperdiary.com/data-science/computer-vision/how-to-implement-sobel-edge-detection-using-python-from-scratch/

https://stackoverflow.com/questions/64197780/how-to-generalize-fast-matrix-multiplication-on-gpu-using-numba/64198479#64198479

https://realpython.com/intro-to-python-threading/

https://numba.readthedocs.io/en/stable/cuda/overview.html#requirements

https://www.vincent-lunot.com/post/an-introduction-to-cuda-in-python-part-1/

https://nyu-cds.github.io/python-numba/05-cuda/

https://numba.readthedocs.io/en/stable/cuda/examples.html

https://sahilchachra.medium.com/dive-into-basics-of-gpu-cuda-accelerated-programming-using-numba-in-python-a0be21aa00b7

https://towardsdatascience.com/cuda-by-numba-examples-1-4-e0d06651612f

https://thedatafrog.com/en/articles/cuda-kernel-python/

https://parzibyte.me/blog/2021/08/16/python-convertir-segundos-segundos-minutos-horas/

http://diagramas-de-flujo.blogspot.com/2013/01/convertir-de-segundos-horas-minutos-y-segundos-en-Python.html

ALEXNET Y REDES CONVOLUCIONALES

- https://petuum.medium.com/intro-to-distributed-deep-learning-systems-a2e45c6b8e7
- https://www.kdnuggets.com/2021/02/evaluating-deep-learning-models-confusion-matrix-accuracy-precision-recall.html
- https://petuum.medium.com/intro-to-distributed-deep-learning-systems-a2e45c6b8e7

- https://medium.com/analytics-vidhya/concept-of-alexnet-convolutional-neural-network-6e73b4f9ee30
- https://pytorch-ignite.ai/tutorials/intermediate/01-cifar10-distributed/
- https://neptune.ai/blog/distributed-training
- https://glassboxmedicine.com/2020/03/04/multi-gpu-training-in-pytorch-data-and-model-parallelism/
- https://theaisummer.com/distributed-training-pytorch/
- https://leimao.github.io/blog/PyTorch-Distributed-Training/
- PyTorch for Deep Learning & Machine Learning – Full Course - YouTube
- Deep Learning With PyTorch - Full Course - YouTube
- Build a Neural Network with Pytorch - PART 1 - YouTube

Paralelismo de datos

- https://pytorch.org/tutorials/intermediate/ddp_tutorial.html
- https://handbook.pytorch.wiki/chapter1/data_parallel_tutorial.html#
- https://towardsdatascience.com/writing-distributed-applications-with-pytorch-f5de4567ed3b
- https://stackoverflow.com/questions/71036271/in-the-pytorch-distributed-data-parallel-ddp-tutorial-how-does-setup-know-i
- https://leimao.github.io/blog/PyTorch-Distributed-Training/
- https://github.com/pytorch/examples/tree/main/distributed/ddp
- https://www.kaggle.com/code/residentmario/notes-on-parallel-distributed-training-in-pytorch/notebook
- https://pythonrepo.com/repo/rentainhe-pytorch-distributed-training
- https://towardsdatascience.com/how-to-scale-training-on-multiple-gpus-dae1041f49d2
- https://medium.com/intel-student-ambassadors/distributed-training-of-deep-learning-models-with-pytorch-1123fa538848
- https://towardsdatascience.com/how-to-scale-training-on-multiple-gpus-dae1041f49d2

Paralelismo de modelo

- https://towardsdatascience.com/model-parallelism-in-one-line-of-code-352b7de5645a
- https://medium.com/deelvin-machine-learning/model-parallelism-vs-data-parallelism-in-unet-speedup-1341bc74ff9e

- https://medium.com/@esaliya/model-parallelism-in-deep-learning-is-not-what-you-think-94d2f81e82ed
- https://leimao.github.io/blog/Data-Parallelism-vs-Model-Paralelism/
- https://mxnet.incubator.apache.org/versions/1.9.1/api/faq/model_parallel_lstm.html
- https://blog.paperspace.com/alexnet-pytorch/
- https://blog.paperspace.com/how-to-maximize-gpu-utilization-by-finding-the-right-batch-size/

Artículos consultados

- A guide to convolution arithmetic for Deep learning. Vincent Dumoulin and Francesco Visin MILA, Universite de Montreal, Politecnico di Milano.
- ImageNet Classification with Deep Convolutional Neural Networks. Alex Krizhevsky et al. University of Toronto.
- Gradient-Based Learning Applied to Document Recognition. Yann LeCun et al.
- Demystifing Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis.
- Machine learning applications in cancer prognosis and prediction. Konstatina Kourou

APENDICE A

COMANDOS ÚTILES DE CUDA

CUDA es una plataforma de cómputo paralelo de propósito general y modelos de programación para el motor de cómputo paralelo de GPU. El uso de las GPUs se ha extendido debido a su habilidad para poder acelerar tareas que son paralelizables tales como operaciones de algebra lineal y operaciones que requieren ciclos, que son operaciones muy utilizadas en el entrenamiento de redes neuronales y se consideran operaciones computacionalmente intensivas.

torch.cuda.is_available()

Determina si el ambiente de CUDA esta disponible

torch.cuda.device_count()

El conteo de dispositivos. Se refiere al número de nucleos disponibles.

torch.cuda.get_device_capability() –

Regresa una tupla la cual representa una caracteristica conocida como la capacidad de CUDA del dispositivo. La capacidad de CUDA son versions de Plataforma de hardware, y la capacidad regresada por el dispositivo sera dependiente de la generacion de cómputo.

La capacidad de cómputo de la GPU se puede consulta ren la página de NVIDIA en la siguiente página web.

https://developer.nvidia.com/cuda-gpus

torch.cuda.device -

Administrador de contexto que intercambia el contexto de ejecución a diferentes dispositivos.

torch.cuda.get_device_name -

Obtiene el nombre del dispositivo. Se puede revisar por medio del comando nvidia-smi.

watch -d “nvidia-smi”

Un comando que puede llegar a ser util para monitorear el uso de la GPU en el tiempo sobre la terminal. Este comando ejecuta una consulta a partir de la utilidad CLI nvidia-smi cada dos segundos.

$nvidia-smi --list-gpus

En caso de que la computadora tenga mas de un GPU mostrara una lista de las GPUs que se encuentren disponible.

Compatibilidad de CUDA y PyTorch

La versión de Pytorch con el CUDA toolkit versión 11.3 son compatibles

NEW ENVIRONMENT

$conda create -n ml python=3.8

$conda activate ml

first option to install cuda

$conda install -c anaconda cudatoolkit

second and best option

 # se instala junto con la versión más adecuada de cudatoolkit

$conda install -c pytorch pytorch

$conda install -c anaconda jupyter

APENDICE B

COMANDOS DE INSTALACION PYTORCH CUDA EN AMBIENTE LOCAL

Los siguientes comandos se utilizaron en una máquina virtual VirtualBox en el sistema operativo Ubuntu 20.04. El framework PyTorch permite la programación de redes neuronales dando un balance entre la especificación de funciones para la implementación de la red como versatilidad para facilitar la implementación. La herramienta Anaconda en su versión de software libre es un ambiente que incluye un conjunto de herramientas que permiten el desarrollo de proyectos relacionados al análisis de datos y proyectos de aprendizaje de máquina, aprendizaje profundo y computo numérico y científico.

```
$list physycal devices

$nvidia-smi --list-gpus

$conda create -n ml python=3.7

$conda activate ml

$conda install -c anaconda cudatoolkit

# install pytorch first

$conda install -c pytorch pytorch

$conda install -c anaconda jupyter

$jupyter notebook
```

APENDICE C

COMPILACION DE PROGRAMAS EN MPI E INSTALACION DE MPI4PY

Para la programación en MPI se realizan pruebas en ambientes LInux Ubuntu 20.04 en máquina virtual VirtualBox.

Para la compilación y ejecución de programas en ambiente local se hace con una computadora con procesador Intel Core i7 y 4 núcleos disponibles en la máquina virtual. La implementación que se usa requiere la instalación de MPI en C. Para esto se usa la implementación de openMPI. MPI4PY es una implementación que agrega las funcionalidades de MPI en el lenguaje Python por medio de una interfaz orientada a objetos

Se ingresan en la terminal de Ubuntu los siguientes comandos de instalación.

```
$sudo apt-get update
$sudo apt-get upgrade
$ sudo apt-get install libopenmpi-dev
$ sudo apt-get install openmpi-bin
```

COMANDOS DE COMPILACION Y EJECUCION PARA 4 PROCESOS

Compilación de MPI en C para ambiente Linux Ubuntu 20.04

```
compilación:   mpicc filename.c -o filename
ejecución    :   mpirun -np 4 ./filename
```

mpicc es un wrapper para el compilador de C. Este script es ejecutar algún programa. El wrapper simplifica la ejecución del compilador donde encontrar los archivos de cabecera necesarios en la cual las librerías
se enlazan con el archivo objeto.

Algunos sistemas también soportan la compilación por medio del comando mpiexec

En el siguiente ejemplo se ejecuta el programa con 4 procesos.

```
$ mpiexec -n 4 ./mpi_helloWorld
```

Una vez instalado MPI se hace la instalación de MPI4PY. Este paquete requiere la instalación previa de Python en su versión 3.8. En ocasiones no es necesario instalar en Ubuntu Python debido a que la version 20.04 viene con Python 3.8 ya instalado. Se recomienda hacer un ambiente virtual en caso de que se tenga una versión mayor a la 3.8

MPI4PY es un paquete que implementa el estándar MPI el cual funciona sobre la versión MPI de C. Este paquete provee comunicación para los objetos de Python, así como comunicación entre procesos.

Comandos de instalación MPI4PY Y Python 3.8

```
$sudo apt-get install python3
$sudo apt install python3-pip
$sudo apt install python3-mpi4py
```

Verificación de instalación del paquete MPI4PY con PIP

```
$python3 -m pip show mpi4py
```

La instalación de MPI4PY se puede instalar tanto de forma directa en Python como por medio de la herramienta Anaconda a través de los siguientes comandos. Se crea un nuevo ambiente con la bilbioteca junto con algunas otras librerías complementarias de MPI.

Instalación de MPI4PY en ambiente Anaconda

```
$conda create --name mpi python=3.8 mpi4py numpy scipy
```

Uso de instalación de clusters en miniconda.

```
$module load Langs/Python//miniconda
$conda create --name mpi python=3.8 mpi4py numpy scipy
```

Opcional. Instalación de Atom para la edición de código

El editor de Atom es de distribución Libre por medio de una licencia MIT y permite escribir programas con una sintaxis clara y a color con posibilidades de autocompletado. Es posible utilizar otros editores en Ubuntu tales como vi o nano para la edición de los programas. Una vez declarado el nuevo ambiente se puede comenzar a realizar pruebas para programas en MPI4PY.

Comandos para la ejecución de programas con MPI4PY

```
$mpirun -np 4 python3 ejemplo1.py
$gnome-screenshot -w     # permite hacer una captura de pantalla de la terminal
$clear     #limpia la pantalla de la terminal
```

En caso de que se quiera cancelar la ejecución del programa CTRL+c detiene la ejecución y la terminal permite abrir una nueva línea.

A continuación, se mencionan algunas definiciones importantes en la escritura de programas de MPI.

- COMM
  El mundo de comunicación definido por medio de MPI

- RANK
  Un número de identificación dado a cada proceso interno para definir la comunicación

- SIZE
  número total de procesos asignados

- BROADCAST
  Comunicación uno a muchos

- SCATTER
  Distribución de datos de uno a muchos

- GATHER
  Distribución de datos muchos a uno

NOTAS DE COMANDOS BASICOS DE MPI4PY

```
from mpi4py import MPI

# inicializar el mundo de comunicación
comm = MPI.COMM_WORLD

# obtener el tamaño del mundo de comunicación
size = comm.Get_size()

# obtener el ID de proceso
rank = comm.Get_rank()
```

Las comunicaciones punto a punto se hacen por medio de los operadores send y recieve.

APENDICE D
CODIGOS DE PROGRAMAS
En este apéndice se encuentran los códigos de programas correspondientes al presente trabajo.

Código de Max Pooling GPU

```
REFERENCIAS DE MAX POOLING GPU
PARA CONSULTAR LAS REFERENCIAS COMPLETAS CONSULTAR LA SECCION DE BILBIOGRAFÍA Y REFERENCIAS
```

https://d2l.ai/chapter_convolutional-neural-networks/pooling.html
https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/
https://www.kaggle.com/questions-and-answers/59502
http://deeplearning.stanford.edu/tutorial/supervised/Pooling/
https://numpy.org/doc/stable/reference/generated/numpy.ndarray.max.html
https://stackoverflow.com/questions/33569668/numpy-max-vs-amax-vs-maximum
https://www.sharpsightlabs.com/blog/numpy-maximum/
https://sparkbyexamples.com/numpy/calculate-maximum-numpy-array/
https://www.tutorialsandyou.com/python/numpy-max-in-python-39.html
https://medium.com/aiguys/pooling-layers-in-neural-nets-and-their-variants-f6129fc4628b
https://blog.paperspace.com/pooling-in-convolutional-neural-networks/
https://medium.com/analytics-vidhya/matrix-multiplication-in-cuda-a-simple-guide-bab44bc1f8ab
https://www.youtube.com/watch?v=uihBwtPIBxM
https://medium.com/towards-data-science/tensorflow-for-computer-vision-how-to-implement-pooling-from-scratch-in-python-379cd7717de9
https://medium.com/analytics-vidhya/simple-cnn-using-numpy-part-iii-relu-max-pooling-softmax-c03a3377eaf2
https://medium.com/rapids-ai/the-life-of-a-numba-kernel-a-compilation-pipeline-taking-user-defined-functions-in-python-to-cuda-71cc39b77625
https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/
https://www.youtube.com/watch?v=sSWCeZVIN8w
https://hardzone.es/tutoriales/componentes/puertos-pcie-tipos-placas-base/
http://vision.stanford.edu/teaching/cs131_fall1920/slides/20_deep_cnns.pdf
https://dingyan89.medium.com/calculating-parameters-of-convolutional-and-fully-connected-layers-with-keras-186590df36c6
http://www.datasciencecourse.org/notes/matrices/
https://www.kdnuggets.com/2021/05/essential-linear-algebra-data-science-machine-learning.html
https://towardsdatascience.com/convolutional-neural-network-17fb77e76c05
https://cs.colby.edu/courses/F19/cs343/lectures/lecture11/Lecture11Slides.pdf

```
# -*- coding: utf-8 -*-
"""
# maxPoolingGPU.py

PROGRAMA DE MAX POOLING CON NUMBA CUDA TESINA UNAM IIMAS

$ python3 maxPoolingGPU.py

Se cambia range por np.arange


Multiplicacion de matrices por medio de Numba  CUDA


@author: Juan Carlos Lopez Nunez
"""
#import random
import math
import numpy as np
#import threading
import time

#from threading import Thread

#from time import sleep, perf_counter

import numba
#from numba import cuda
from numba import cuda, float32
import sys
import os
import cupy as cp

import warnings
from numba.core.errors import NumbaPerformanceWarning

import platform


### VARIABLES DE MAX POOLING

#imagen_entrada = []

#img_sinPadding = []  # copia de la imagen original para el padding

stride = 2
dim_entrada = 4

padding = 0

#imagen_Final = [] # matriz que guarda la imagen final

#### VARIABLES QUE INTERVIENEN EN LA PARALELIZACION MULT MATRICES
#filtro_imagen = []  # filtro
#resultado = []  # resultado del cual se extrae el max pooling


dim_mult = 2 # dimensiones por defecto de matrices de 2X2

numHilos = 1   #probar primero con un hilo

tiempoGPU = 0.0  # TIEMPO EN MSEG
numKernels = 0      # Numero de multiplicaciones (kernels extraidos)
```

```
# variable para generar matrices reproducibles
semillaNumAleatorio = None



# ------------------------------------------------------------------------------

def recorrerImagen(imagen_ent):
    global imagen_Final
    imagen_Final = []

    renglon_imagen = imagen_ent.shape[0]
    col_imagen = imagen_ent.shape[1]
    #

    for i in cp.arange(0, renglon_imagen, stride):
        for j in cp.arange(0,col_imagen, stride):

            i = int(i)
            j = int(j)

            # extrae el kernel correspondiente
            kernel = imagen_ent[i:i+dim_mult, j:j+dim_mult]

            # si el kernel es cuadrado extraerlo para enviarlo a la multiplicacion
            # de matrices en caso contrario ignorar el kernel
            if kernel.shape == (dim_mult, dim_mult):

               # aplicar la operacion de max pooling
               # -----------------------
               kernel = np.array(kernel)
               '''
               print("===kernel=====1")
               print(" " + str(kernel))
               print("==============2")
               '''
               ##................ PARALELIZACION .............##
               '''
               print("---------------6")
               print("valor_max= " + str(valor_max))
               print("---------------6")
               '''
               #para evitar un error logico en Max Pooling
               #resultado = np.zeros((dim_mult, dim_mult))
               #resultado = resultado.astype(int)

               valorMax = multGPU2(kernel)    #### ---> enviar el kernel extraido a la mult
de matrices



               imagen_Final.append(valorMax)


            # ............fin if
    # ..........fin for i-j

# ------------------------------------------------------------------------------
def ingresoMatriz():
    # intervienen en la paralelizacion
    global dim_entrada


    # variables e hiperparametros de Max Pooling
```

```
    #global dim_entrada
    #global stride
    #global padding

    #global semillaNumAleatorio


    print()
    print("ORDEN DE MATRIZ DE ENTRADA ")

    #dim_entrada = int(input("Ingresar la dimension de la imagen de entrada NXN : "))
    #numHilos = int(input("Ingresar el numero de hilos : "))
    #print()
    #print("AJUSTE DE HIPERPARAMETROS")
    #print()

    #dim_mult = int(input("Ingresar la dimension del kernel : "))
    #stride = int(input("Ingresar el stride : "))

    #padding = int(input("Ingresar el padding : "))


    while True:
        try:
            dim_entrada = input("Ingresar la dimension de la imagen de entrada NXN : ")
            dim_entrada = int(dim_entrada)

            break
        except ValueError:
            print("Entero no valido. Intente de nuevo ...")
    print("dim. de imagen entrada ingresado.")

    # ----------------------------------------
    menuOpciones()


# ------------------------------------------------------------------------
def ingresoHiperparametros():
    # intervienen en la paralelizacion
    global dim_mult

    # variables e hiperparametros de Max Pooling

    global stride
    global padding

    print()
    while True:
        try:
            dim_mult = input("Ingresar la dimension del kernel : ")
            dim_mult = int(dim_mult)
            break
        except ValueError:
            print("Entero no valido. Intente de nuevo ...")
    print("dim kernel ingresado.")

    while True:
        try:
            stride = input("Ingresar el stride (desplazamiento) : ")
            stride = int(stride)
            break
        except ValueError:
            print("Entero no valido. Intente de nuevo ...")
    print("stride ingresado.")
```

```
    while True:
        try:
            padding = input("Ingresar el padding : ")
            padding = int(padding)
            break
        except ValueError:
            print("Entero no valido. Intente de nuevo ...")
    print("Padding ingresado.")


# ------------------------------------------------------------------------------

# ----------------------------------------------------------------------

def imprimirMenu():
    print("1: Ingreso manual")
    print("2: Ingreso aleatorio")

# ----------------------------------------------------------------------

def menuOpciones():
    opcion = 0
    while (opcion == 0):
        imprimirMenu()
        opcion = ''
        try:
            opcion = int(input("Ingresar opcion : "))
        except:
            print('Opcion incorrecta ')

        if opcion == 1:
            ingresoManual()
            break
        elif opcion == 2:
            ingresoAleatorio()
            break
        else:
            print('Opcion invalida. Intente de nuevo-')

# ----------------------------------------------------------------------

def ingresoAleatorio():
    global imagen_entrada
    global semillaNumAleatorio

    while True:
        try:
            matricesRep = input("Genera matrices reproducibles?(0 ACEPTAR, otro valor
descartar )?: ")
            matricesRep = int(matricesRep)

            if matricesRep == 0:
                semillaNumAleatorio = True
                print("Matrices reproducibles (semilla)")
            else:
                print("Se generara una matriz aleatoria")
                semillaNumAleatorio == False

            break
        except ValueError:
            print("Entero no valido. Intente de nuevo ...")
    print("Opcion ingresada.")
```

```
    if semillaNumAleatorio == True:
        np.random.seed(35)

    valores = 10   # valores aleatorios desde el 0 hasta el 9
    imagen_entrada = np.random.randint(valores, size=(dim_entrada, dim_entrada))
    imagen_entrada = imagen_entrada.astype(int)



# ---------------------------------------------------------------------
def ingresoManual():
    global imagen_entrada

    imagen_entrada = np.zeros((dim_entrada, dim_entrada)).astype(int)

    renglon_imagen = imagen_entrada.shape[0]
    col_imagen = imagen_entrada.shape[1]

    numValores = imagen_entrada.size
    print("Num. valores a ingresar", numValores)
    print()

    cont = 1 # conteo de numero de valores
    print("Ingreso manual de la matriz")
    print("---------------------------")
    for i in np.arange(renglon_imagen):
        print("Renglon {}  ".format(i))
        print("-----------")
        for j in np.arange(col_imagen):
            print(f"# Valor={cont} | img[{i}][{j}]", end='')

            #imagen_entrada[i][j] = int(input(" = "))
            while True:
                try:
                    imagen_entrada[i][j] = input(" = ")
                    imagen_entrada[i][j] = int(imagen_entrada[i][j])
                    break
                except ValueError:
                    print("Entero no valido. Intente de nuevo ...")

            cont = cont + 1
        print()   # nueva linea

    # ----- fin for i-j


# ---------------------------------------------------------------------

# ---------------------------------------------------------------------------------
def inicializarMatrizEnt():
    global imagen_entrada
    global img_sinPadding


    #imagen_entrada = np.random.random((dim_entrada, dim_entrada))
    #imagen_entrada = imagen_entrada * 10

    #valores = 10  # rango de valores desde 0 hasta n


    #if semillaNumAleatorio == True:
    #    np.random.seed(35)
```

```
    # ================================================
    #### semilla para generar matrices reproducibles
    # genera la misma secuencia de numeros aleatorios

    # agregar imagen de entrada para la cual se puede agregar de forma manual una matriz
    #num = 3
    #imagen_entrada = np.full((dim_entrada, dim_entrada), num, npint)

    #imagen_entrada = np.random.randint(valores, size=(dim_entrada, dim_entrada))
    #imagen_entrada = imagen_entrada.astype(int)

    print()
    print("--Imagen de entrada--")
    imprimirMatriz(imagen_entrada)
    print("Imagen Ent: {} ".format(imagen_entrada.shape))
    print()

    if padding != 0:
       #
       img_sinPadding = imagen_entrada

       imagen_entrada = agregarPadding(imagen_entrada)
       print()
       print("Imagen con Padding= {}".format(padding))
       imprimirMatriz(imagen_entrada)
       print("Imagen Ent: {} ".format(imagen_entrada.shape))


    return imagen_entrada

# ------------------------------------------------------------------------

def agregarPadding(imagen_ent):
    global dim_padding
    #
    # si es padding = 1
    dim_padding = dim_entrada+padding*2

    imagenPadding = np.zeros((dim_padding, dim_padding))
    imagenPadding = imagenPadding.astype(int)

    renglon_imagen = imagen_ent.shape[0]
    col_imagen = imagen_ent.shape[1]

    for i in cp.arange(padding, renglon_imagen+padding, step=1):
        for j in cp.arange(padding,col_imagen+padding, step=1):

            i = int(i)
            j = int(j)

            # copiar valor de la imagen anterior en la nueva imagen
            imagenPadding[i][j] = int(imagen_entrada[i-padding][j-padding])


    # .........fin for i-j

    return imagenPadding

# ------------------------------------------------------------------------

def inicializarMatrizMult():

    global filtro_imagen
    global resultado
```

```
    filtro_imagen = np.identity(dim_mult)
    filtro_imagen = filtro_imagen.astype(int)

    #imprimirMatriz(filtro_imagen)

    resultado = np.zeros((dim_mult,dim_mult))
    resultado = resultado.astype(int)

# ------------------------------------------------------------------------
# REGION PARALELA

# ------------------------------------------------------------------------
#Impresion de las mnatrices
def impMatriz(matrizImpresion):

    for r in matrizImpresion:
        print(r)
    print()  # nueva linea

# ------------------------------------------------------------------

def imprimirMatriz(matriz):

    for i in range(len(matriz[0])):
        print("[", end="")
        for j in range(len(matriz[1])):
            print("%2d " %  (matriz[i][j]), end ="")
        print("]", end="")
        print()  # nueva linea
    # fin------- for i-j

# ------------------------------------------------------------------
# ---------------------------------------------------


def revisarVersionGPU():
    print("Version de Python : ", sys.version)
    print("Numpy version: ", np.__version__)
    print("Numba version: ", numba.__version__)
    #print("CUPY version: ", cp.__version__)

    warnings.simplefilter("ignore", category=NumbaPerformanceWarning)
    print()  # nueva linea

    if cuda.is_available() == True:
        # CUDA DISPONIBLE MOSTRAR INFORMACION DE LA GPU
        print("GPU DISPONIBLE")
        print("======================================================================")
        cuda.detect()
        #dev = cuda.current_context().device
        #print(dev.uuid)
        print("======================================================================")
        print()
    else:
        print("NO HAY GPU DISPONIBLE.\n")

# ---------------------------------------------------

# ---------------------------------------------------------

# ---------------------------------------------------
# realiza la multiplicacion de matrices cuadradas
```

```
# ----------------------------------------------------
# CUDA kernel HOST --> DEVICE
# ### ---------------------- CODIGO DE DEVICE
@cuda.jit
def matmul(A, B, C):
    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k,j]
        C[i, j] = tmp
# ----------------------------------------------------
# ----------------------------------------------------
# CUDA kernel HOST --> DEVICE
# ### ---------------------- CODIGO DE DEVICE
@cuda.jit
def matmul2(A, B, C, indices):
    # se obtienen las coordenadas de hilos unicas x, y en el grid 2D
    i, j = cuda.grid(2)

    # The above is equivalent to the following 2 lines of code:
    #i = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    #j = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y

    for i in range(i, dim_mult):
        for j in range(j, dim_mult):
            suma = 0
            for k in range(A.shape[1]):
                suma += A[i, k] * B[k,j]
                #indices[i][j] = i  + j/ 10
            C[i, j] = suma

    # fin for------------------ i.j-k
    # Escribir el indice-x seguido por un decimal y el indice-y
    #
    indices[i][j] = i + j / 10

# ----------------------------------------------------
# multiplicacion rapida de matrices
# por medio de memoria compartida la memoria compartida es por cada bloque
# ----------------------------------------------------
# CUDA kernel HOST --> DEVICE
# ### ---------------------- CODIGO DE DEVICE

# Controla los hilos por bloque y uso de memoria compartida
# los computos se haran sobre bloques de HPBXHPB elementos
# HPB no debe de ser mayorque 32 en este ejemplo

# DECLARACION DE HPB COMO VARIABLE GLOBAL

'''
NOTAS DE MATMUL3



'''

HPB = 32

@cuda.jit
def matmul3(A, C):
    '''
    Realiza la multiplicacion de matrices de C = A * B usando memoria compartida de CUDA
```

```
    '''
    B = cuda.const.array_like(filtro_imagen)

    # define un arreglo en la memoria compartida
    # el tamanio y el tipo de los arreglos deben ser conocidos en tiempo de compilacion
    sA = cuda.shared.array(shape=(HPB, HPB), dtype=float32)
    sB = cuda.shared.array(shape=(HPB, HPB), dtype=float32)

    # indices de la rejilla para problemas en 2D
    x, y = cuda.grid(2)

    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bpg = cuda.gridDim.x      # blocks per grid

    # cada hilo calcula un elemento en la matriz resultante
    # el producto punto es fragmentado en productos punto de vectores largos-HPB
    tmp = float32(0.)
    for i in range(bpg):
        # precargar datos en la memoria compartida
        sA[ty, tx] = 0
        sB[ty, tx] = 0

        if y < A.shape[0] and (tx + i * HPB) < A.shape[1]:
            sA[ty, tx] = A[y, tx + i * HPB]
        if x < B.shape[1] and (ty + i * HPB) < B.shape[0]:
            sB[ty, tx] = B[ty + i * HPB, x]

        # esperar hasta que todos los hilos terminen su precarga
        cuda.syncthreads()

        # calcular el producto parcial sobre la memoria compartida
        for j in range(HPB):
            tmp += sA[ty, j] * sB[j, tx]

        # esperar hasta que todos los hilos terminen de calcular
        cuda.syncthreads()
    if y < C.shape[0] and x < C.shape[1]:
        C[y, x] = tmp

### end matmul3()



'''
# ------------------------------------------------

def verificar_mult(res, kernel):

    multiplica = np.dot(kernel, filtro_imagen)
    print("--------VERIFICACION DE MULTIPLICACION------------#")
    imprimirMatriz(multiplica)
    print(multiplica.shape)
    print(multiplica.dtype)
    print("--------------------------------------------------#")
    #np.array_equal(resultado, multiplica) == True:

    if np.array_equal(multiplica, res):
        print("Las multiplicaciones coinciden. \n")
        print("Resultado obtenido con CUDA")
        print(res)
    else:
        print("Las multiplicaciones no coinciden.\n")
        print("Diferencia")
```

```
        print(res - multiplica)

'''


# ------------------------------------------------

# ------------------------------------------------
def multGPU2(kernel):

    global kernel_device
    #global filtro_device
    global resultado_device
    global resultado

    global tiempoGPU
    global numKernels
    global filtro_device



    # ### ---------------------- CODIGO DE HOST


    #revisarVersionGPU()

    #iniciarMatrices()
    # indices de la rejilla del kernel
    # GTX 1050     16, 16
    # TESLA T4
    # TESLA K20

    # CONTROLA LOS HILOS POR BLOQUE
    # el calculo se hara en bloques de  HPB X HPB elementos
    #HPB = 16


    # indices de la matriz de la GPU
    #indice_x = 8
    #indice_y = 8
    #indices = np.zeros((indice_x, indice_y))


    # ----------------------CONFIGURAR BLOQUE GPU ---------
    stream = cuda.stream()

    #### enviar GPU TRANSFERENCIA DE DATOS   host -> device
    kernel_device = cuda.to_device(kernel, stream=stream)
    #filtro_device = cuda.to_device(filtro_imagen)
    resultado_device = cuda.to_device(resultado)

    #indices_device = cuda.to_device(indices)

    # ----------------------------------------

    kernel_cupy = cp.asarray(kernel_device)
    resultado_cupy = cp.asarray(resultado_device)


    # configurar los bloques GPU : GTX1050 / Tesla T4 / K20
    threadsperblock = (HPB, HPB)

    # configurar bloques GPU para matmul1-if y matmul2-fori-j-k | grid(2)
```

```
    '''
    blockspergrid_x = math.ceil(resultado.shape[0] / threadsperblock[0])
    blockspergrid_y = math.ceil(resultado.shape[1] / threadsperblock[1])
    blockspergrid = (blockspergrid_x, blockspergrid_y)
    '''

    # configurar los bloques GPU para matmul3 (multiplicacion con memoria compartida)
    # ================================================================================
    grid_y_max = max(kernel.shape[0], filtro_imagen.shape[0])
    grid_x_max = max(kernel.shape[1], filtro_imagen.shape[1])
    blockspergrid_x = math.ceil(grid_x_max / threadsperblock[0])
    blockspergrid_y = math.ceil(grid_y_max / threadsperblock[1])

    blockspergrid = (blockspergrid_x, blockspergrid_y)

    # ================================================================================


    # ------------------------------------------------------
    resultado = np.zeros((dim_mult,dim_mult)).astype(int)   # reseteo de la multiplicacion


    # se crean dos eventos uno al inicio del computo y otro al final del computo
    inicioEvento = cuda.event()
    finEvento = cuda.event()

    # iniciar el kernel
    inicioEvento.record(stream=stream)

    '''
    print("blocks per grid =", blockspergrid)
    print("ThreadsPerBlock =", threadsperblock)
    '''
    # numero total debloques en el grid
    # numero total de hilos en el bloque

    #matmul2[blockspergrid, threadsperblock](kernel_device, filtro_device, resultado_device,
indices_device)

    matmul3[blockspergrid, threadsperblock](kernel_cupy , resultado_cupy)

    finEvento.record(stream=stream)


    # tareas futuras asignadas a stream esperaran hasta que este evento sea completado
    finEvento.wait(stream=stream)
    # sincronizar este evento con el CPU, para que pueda usar sus valores esperar a que
    # los procesos terminen
    finEvento.synchronize()


    # tiempo de ejecucion del kernel
    tiempo_medido = inicioEvento.elapsed_time(finEvento)   # medido en milisegundos
    #tiempo_seg = tiempo_medido / 1000.0                    # tiempo en seg

    #print()

    # tiempo Medido y numero de multiplicaciones
    tiempoGPU += tiempo_medido
    numKernels = numKernels + 1



    # -----------------fin-for -i
```

```
    #timing_cpu2 = timing_cpu

    # copiar el resultado de regreso al host

    #resultado = np.zeros((dim_mult, dim_mult))   # reseteo


    ### TRABSFERENCIA DE DATOS        DEVICE --> HOST


    #indices = indices_device.copy_to_host()

    # impresion de los indices de la GPU
    '''
    print("Indices")
    print("-------")
    print(indices)
    '''

    #print()
    resultado = resultado_device.copy_to_host()

    # ------------------------------------------------------------
    # IMPRIMIR EL RESULTADO DE LA MULTIPLICACION DE MATRICES
    # ------------------------------------------------------------
    '''
    print("resultado")
    imprimirMatriz(resultado)
    print(resultado.shape)
    print(resultado.dtype)
    print()

    '''

    # obtener el valor maximo a partir de la multiplicacion de matrices

    #valorMax = np.max(resultado)    # version de Numpy para obtener valor mayor
    valorMax = mayorValor(resultado)   # version definida en el programa

    # ------------------------------------------------
    # MEDICIONES DE LA MULTIPLICACION DE MATRICES
    # ------------------------------------------------
    '''
    print("ValorMax =", valorMax)
    print()
    verificar_mult(resultado)

    #print("Tiempo de ejecucion: ", timing_cpu)
    print()
    print("============================================================")
    print()
    print(f"Tiempo de ejecucion del kernel {tiempo_medido:.4f} mseg |  {tiempo_seg:.4f}  seg
")
    print()
    print("============================================================")
    '''
    return valorMax

# ------------------------------------------------
# encontrar el valor mayor

def mayorValor(arr):
    n = arr.shape[0]  # obtener dimensiones renglon / columna para la matriz cuadrada
```

```
    max_elem = arr[0][0]   # incializar elemento maximo con el primer valor

    for i in range(0, n):
        for j in range(0, n):
            if arr[i][j] > max_elem:
                max_elem = arr[i][j]

    ## fin for i-j
    return max_elem


# ------------------------------------------------

def info_CPU():
    numProc = os.cpu_count()
    print("================================================")
    print("Numero de procesadores CPU : ", numProc)
    print(platform.processor() )  # regresa el nombre del procesador como cadena
    print("Arquitectura :" + platform.machine())   # tipo de maquina
    print("Sistema Operativo : " + platform.system() +" "+ platform.version())
    print("================================================")


# ------------------------------------------------------

def tiempoTotal(segs):
    horas = int(segs / 60 / 60)
    segs -= horas * 3600
    minutos = int(segs / 60)
    segs -= minutos * 60

    return f" {horas:} Horas : {minutos:} Minutos : {segs:4f} segundos"

# -------------------------------------------------

def obtenerMaxPooling():
    global imagen_Final

    global tiempoGPU
    global numKernels

    tiempoGPU = 0.0  # reseteo del tiempo de la mult matrices GPU (en mseg)
    numKernels = 0   # reseteo de contador


    ingresoMatriz()
    ingresoHiperparametros()


    imagen = inicializarMatrizEnt()
    inicializarMatrizMult()


    tiempo_inicio = time.time()
    recorrerImagen(imagen)

    tiempo_fin = time.time()
    print()
    tiempoExperimento = tiempo_fin - tiempo_inicio



    # calcular el tamanio de salida para 'imagen_Final'
    n_ent = dim_entrada
```

```
    p = padding                     # tamanio del padding
    k = dim_mult                    # tamanio del kernel de convolucion
    s = stride                      # tamanio del stride de convolucion
    #print("--------")
    #print(n_ent)
    #print(p)
    #print(k)
    #print(s)

    #print("------")

    n_salida = math.floor(((n_ent + 2*p -k)/s)) + 1
    # math.floor devuelve el numero entero mas cercano menor o igual a un numero dado
    #print(n_salida)


    print()
    print("----MAX POOLING CON GPU----")
    print()
    # informacion de la CPU , GPU y Sist. Operativo
    #info_CPU()
    print()
    #revisarVersionGPU()
    print()
    print("   RESULTADO MAX POOLING")
    print()

    #imagen_Final = np.array(imagen_Final).reshape((n_salida, n_salida))
    imagen_Final = np.array(imagen_Final)
    imagen_Final = imagen_Final.reshape((n_salida, n_salida))
    imprimirMatriz(imagen_Final)
    print("Imagen Final: {} ".format(imagen_Final.shape))
    print()
    print("RESULTADOS DEL EXPERIMENTO")
    print("-------------------------------------------------------------")


    if padding != 0:
       #
       print("Img. Entrada      : {0:1d}X{1:1d} | # Elementos : {2:1d}   | Memoria : {3:2d}
bytes.".format(dim_entrada, dim_entrada, img_sinPadding.size, img_sinPadding.nbytes))
       print("Img. Entrada con Padding    : {0:1d}X{1:1d} | # Elementos : {2:1d}   | Memoria
:      {3:2d}       bytes.".format(dim_padding,       dim_padding,       imagen_entrada.size,
imagen_entrada.nbytes))
    else:
       print("Img. Entrada      : {0:1d}X{1:1d} | # Elementos : {2:1d}   | Memoria : {3:2d}
bytes.".format(dim_entrada, dim_entrada, imagen_entrada.size, imagen_entrada.nbytes))


    print()



    print("Img. Max Pooling : {0:1d}X{1:1d} | # Elementos : {2:1d}   | Memoria : {3:2d}
bytes.".format(n_salida, n_salida, imagen_Final.size, imagen_Final.nbytes))
    print()

    tiempo_seg = tiempoGPU / 1000.0                    # tiempo en seg
    #print("DimEnt= {} | Dim_Sal= {} |Tiempo= {:,.6f} seg | # Hilos= {}".format(dim_entrada,
n_salida, tiempoExperimento, numHilos))
    print("Tiempo medido desde CPU = {:.6f} seg | #".format( tiempoExperimento))
    print("Tiempo mult con GPU = {:.6f} mseg |  {:.6f} seg ".format( tiempoGPU, tiempo_seg))

    print()
```

```
    print("Tiempo Total CPU + GPU")
    print("----------------------")
    cadena1 = tiempoTotal( tiempoExperimento  )
    print(cadena1)
    print()
    print("Numero de kernels extraidos : {}".format(numKernels))
    print("------------------------------------------------------------")
    print()
    print("HIPERPARAMETROS MAX POOLING ")
    print("Kernel :  {}X{}".format(dim_mult, dim_mult))
    print("Stride :  ({},{})".format(stride, stride))
    print("Padding:  {}".format(padding))
    print()

    info_CPU()
    print()
    revisarVersionGPU()
    # la escala se reduce a la mitad de la imagen de entrada con p = 0, k = 2, s = 2

# ------------------------------------------------------------------------------------

def main():
    print("PROGRAMA DE MAX POOLING CON GPU TESINA CAR UNAM IIMAS")
    print("ING. JUAN CARLOS LOPEZ NUNEZ")

    if cuda.is_available() == True:
        #
        print("GPU Disponible ")

        #print()
        #revisarVersionGPU()
        print()
        obtenerMaxPooling()
    else:
        print("NO HAY GPU DISPONIBLE. \n SE NECESITA DE UNA GPU PARA ESTE PROGRAMA. \n")


# ------------------------------------------------------------------------------------
if __name__=="__main__":
    main()


# ......................fin main
```

Código de Max Pooling serial

REFERENCIAS MAX POOLING SERIAL
PARA LA CONSULTA DE REFERENCIAS COMPLETAS CONSULTAR LA SECCION DE BILBIOGRAFÍA Y REFERENCIAS

https://d2l.ai/chapter_convolutional-neural-networks/pooling.html
http://deeplearning.stanford.edu/tutorial/supervised/Pooling/
https://numpy.org/doc/stable/reference/generated/numpy.ndarray.max.html
https://www.sharpsightlabs.com/blog/numpy-maximum/
https://sparkbyexamples.com/numpy/calculate-maximum-numpy-array/
https://www.tutorialsandyou.com/python/numpy-max-in-python-39.html
https://medium.com/aiguys/pooling-layers-in-neural-nets-and-their-variants-f6129fc4628b
https://blog.paperspace.com/pooling-in-convolutional-neural-networks/

https://medium.com/analytics-vidhya/matrix-multiplication-in-cuda-a-simple-guide-bab44bc1f8ab
https://www.youtube.com/watch?v=uihBwtPIBxM
https://medium.com/towards-data-science/tensorflow-for-computer-vision-how-to-implement-pooling-from-scratch-in-python-379cd7717de9
https://medium.com/analytics-vidhya/simple-cnn-using-numpy-part-iii-relu-max-pooling-softmax-c03a3377eaf2
https://medium.com/rapids-ai/the-life-of-a-numba-kernel-a-compilation-pipeline-taking-user-defined-functions-in-python-to-cuda-71cc39b77625
https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/

```
# -*- coding: utf-8 -*-
"""
# maxPooling.py
PROGRAMA DE MAX POOLING SERIAL TESINA UNAM IIMAS

$ python3 maxPooling1_6serial.py

Se cambia range por np.arange


@author: Juan Carlos Lopez Nunez
"""
import random
import math
import numpy as np
import threading
import time
from threading import Thread

from time import sleep, perf_counter

import platform
import os

### VARIABLES DE MAX POOLING

imagen_entrada = []

stride = 2
dim_entrada = 4

padding = 0

imagen_Final = [] # matriz que guarda la imagen final

#### VARIABLES QUE INTERVIENEN EN LA PARALELIZACION MULT MATRICES
filtro_imagen = []  # filtro
resultado = []  # resultado del cual se extrae el max pooling


dim_mult = 2 # dimensiones por defecto de matrices de 2X2

numHilos = 1   #probar primero con un hilo

matrizI = []
contKernels = 1

# ------------------------------------------------------------------------------
```

```
def recorrerImagen(imagen_ent):
    global tiempoMult  # contador de tiempo de las multiplicaciones de matrices

    renglon_imagen = imagen_ent.shape[0]
    col_imagen = imagen_ent.shape[1]
    #

    for i in np.arange(0, renglon_imagen, stride):
        for j in np.arange(0,col_imagen, stride):

            # extrae el kernel correspondiente
            kernel = imagen_ent[i:i+dim_mult, j:j+dim_mult]

            # si el kernel es cuadrado extraerlo para enviarlo a la multiplicacion
            # de matrices en caso contrario ignorar el kernel
            if kernel.shape == (dim_mult, dim_mult):

               # aplicar la operacion de max pooling
               # -----------------------
               kernel = np.array(kernel)
               '''
               print("===kernel=====1")
               print(" " + str(kernel))
               print("==============2")
               '''
               ##................ PARALELIZACION .............##
               '''
               print("---------------6")
               print("valor_max= " + str(valor_max))
               print("---------------6")
               '''
               #para evitar un error logico en Max Pooling
               #resultado = np.zeros((dim_mult, dim_mult))
               #resultado = resultado.astype(int)
               tiempo_mult1 = time.time()

               valorMax = multMatrizParalela(kernel)

               tiempo_mult2 = time.time()
               tiempoExperimento2 = tiempo_mult2 - tiempo_mult1
               tiempoMult += tiempoExperimento2

               imagen_Final.append(valorMax)


            # ............fin if
    # ..........fin for i-j

# ------------------------------------------------------------------------------
def ingresoMatriz():
    # intervienen en la paralelizacion
    global dim_mult
    global numHilos  # dimension de la mult (ventana)

    # variables e hiperparametros de Max Pooling
    global dim_entrada
    global stride
    global padding

    global contKernels
    global tiempoMult
    tiempoMult = 0.0
    contKernels = 0 # reseteo de contador
```

```
    print("PROGRAMA DE MAX POOLING SERIAL TESINA CAR UNAM IIMAS")
    print("ING. JUAN CARLOS LOPEZ NUNEZ")
    print()
    print("ORDEN DE MATRIZ DE ENTRADA ")

    dim_entrada = int(input("Ingresar la dimension de la imagen de entrada NXN : "))
    #numHilos = int(input("Ingresar el numero de hilos : "))
    print()
    print("AJUSTE DE HIPERPARAMETROS")
    print()

    dim_mult = int(input("Ingresar la dimension del kernel : "))
    stride = int(input("Ingresar el stride : "))

    padding = int(input("Ingresar el padding : "))


# ------------------------------------------------------------------------------
def inicializarMatrizEnt():
    global imagen_entrada

    #imagen_entrada = np.random.random((dim_entrada, dim_entrada))
    #imagen_entrada = imagen_entrada * 10

    valores = 10  # rango de valores desde 0 hasta n

    imagen_entrada = np.random.randint(valores, size=(dim_entrada, dim_entrada))
    imagen_entrada = imagen_entrada.astype(int)

    print()
    print("--Imagen de entrada--")
    imprimirMatriz(imagen_entrada)
    print("Imagen Ent: {} ".format(imagen_entrada.shape))
    print()

    if padding != 0:
       imagen_entrada = agregarPadding(imagen_entrada)
       print()
       print("Imagen con Padding= {}".format(padding))
       imprimirMatriz(imagen_entrada)
       print("Imagen Ent: {} ".format(imagen_entrada.shape))


    return imagen_entrada

# ------------------------------------------------------------------------

def agregarPadding(imagen_ent):
    #
    # si es padding = 1
    dim_padding = dim_entrada+padding*2

    imagenPadding = np.zeros((dim_padding, dim_padding))
    imagenPadding = imagenPadding.astype(int)

    renglon_imagen = imagen_ent.shape[0]
    col_imagen = imagen_ent.shape[1]

    for i in np.arange(padding, renglon_imagen+padding, step=1):
        for j in np.arange(padding,col_imagen+padding, step=1):

            # copiar valor de la imagen anterior en la nueva imagen
            imagenPadding[i][j] = int(imagen_entrada[i-padding][j-padding])
```

```
    # ..........fin for i-j

    return imagenPadding

# ----------------------------------------------------------------------

def inicializarMatrizMult():

    global filtro_imagen
    global resultado


    filtro_imagen = np.identity(dim_mult)
    filtro_imagen = filtro_imagen.astype(int)

    #imprimirMatriz(filtro_imagen)

    resultado = np.zeros((dim_mult,dim_mult))
    resultado = resultado.astype(int)

# ----------------------------------------------------------------------

# ----------------------------------------------------------------------

def multMatrizParalela(kernel):
    # reseteo de resultado
    global contKernels
    resultado = np.zeros((dim_mult,dim_mult)).astype(int)

    for i in range(len(kernel)):
        for j in range(len(filtro_imagen[0])):
            #print("Renglon [{}][{}] {}".format(i, j, k))
            #print("-------")
            for k in range(len(filtro_imagen)):
                resultado[i][j] += int(kernel[i][k] * filtro_imagen[k][j])
                #print("  res[{}][{}]  =  {}".format(i,  j,  resultado[i][j] ))    #  imprimir
elementos linea de diagnostico

            #print("-----------")

            #matrizI += ('|')
    #print("------")
    #print(resultado)
    #print("--------")
    contKernels += 1

    return np.max(resultado)


# ----------------------------------------------------------------------
#Impresion de las mnatrices
def imprimirMatriz(matrizImpresion):

    for r in matrizImpresion:
        print(r)
    print()  # nueva linea
# ----------------------------------------------------------------------
def info_CPU():
    numProc = os.cpu_count()
    print("================================================")
    print("Numero de procesadores CPU : ", numProc)
    print(platform.processor() )  # regresa el nombre del procesador como cadena
```

```
    print("Arquitectura :" + platform.machine())   # tipo de maquina
    print("Sistema Operativo : " + platform.system() +" "+ platform.version())
    print("================================================")

# -----------------------------------------------------
if __name__=="__main__":

    ingresoMatriz()
    imagen = inicializarMatrizEnt()
    inicializarMatrizMult()


    tiempo_inicio = time.time()
    recorrerImagen(imagen)

    tiempo_fin = time.time()
    print()
    tiempoExperimento = tiempo_fin - tiempo_inicio


    # calcular el tamanio de salida para 'imagen_Final'
    n_ent = dim_entrada
    p = padding                    # tamanio del padding
    k = dim_mult                   # tamanio del kernel de convolucion
    s = stride                     # tamanio del stride de convolucion
    #print("--------")
    #print(n_ent)
    #print(p)
    #print(k)
    #print(s)

    #print("------")

    n_salida = math.floor(((n_ent + 2*p -k)/s)) + 1
    # math.floor devuelve el numero entero mas cercano menor o igual a un numero dado
    #print(n_salida)


    print()
    print("----Max Pooling----")
    #imagen_Final = np.array(imagen_Final).reshape((n_salida, n_salida))
    imagen_Final = np.array(imagen_Final)
    imagen_Final = imagen_Final.reshape((n_salida, n_salida))
    imprimirMatriz(imagen_Final)
    print("Imagen Final: {} ".format(imagen_Final.shape))
    print()
    print("RESULTADOS DEL EXPERIMENTO")
    print("-------------------------------------------------------------")
    print("Img. Entrada       : {0:1d}X{1:1d} | # Elementos : {2:1d}   | Memoria : {3:2d}
bytes.".format(dim_entrada, dim_entrada, imagen_entrada.size, imagen_entrada.nbytes))
    print("Img. Max Pooling : {0:1d}X{1:1d} | # Elementos : {2:1d}   | Memoria : {3:2d}
bytes.".format(n_salida, n_salida, imagen_Final.size, imagen_Final.nbytes))
    print()

    #print("DimEnt= {} | Dim_Sal= {} |Tiempo= {:,.6f} seg | # Hilos= {}".format(dim_entrada,
n_salida, tiempoExperimento, numHilos))

    print("Tiempo Mult Matrices : {:,.6f} seg |".format(tiempoMult))
    print("Tiempo    Experimento=    {:,.6f}    seg    |    #    Kernels         extraidos
{}".format(tiempoExperimento, contKernels))
    print("-------------------------------------------------------------")
    print()
    print("HIPERPARAMETROS MAX POOLING ")
    print("Kernel :  ({}X{})".format(dim_mult, dim_mult))
```

```
    print("Stride :  {},{}".format(stride, stride))
    print("Padding:  {}".format(padding))
    print()
    # la escala se reduce a la mitad de la imagen de entrada con p = 0, k = 2, s = 2
    info_CPU()


# .......................fin main

# --------------------------------------------------------------------------------
```

Código de Multiplicación de matrices GPU

REFERENCIAS DE CÓDIGO DE MULTIPLICACION DE MATRICES GPU
PARA LA CONSULTA DE REFERENCIAS COMPLETAS CONSULTAR LA SECCION DE BILBIOGRAFÍA Y REFERENCIAS

https://python-course.eu/python-tutorial/formatted-output.php
https://www.tutorialspoint.com/python/python_multithreading.htm
https://stackoverflow.com/questions/10613131/how-to-access-list-elements
http://www.adeveloperdiary.com/data-science/computer-vision/how-to-implement-sobel-edge-detection-using-python-from-scratch/
https://stackoverflow.com/questions/64197780/how-to-generalize-fast-matrix-multiplication-on-gpu-using-numba/64198479#64198479
https://realpython.com/intro-to-python-threading/
https://numba.readthedocs.io/en/stable/cuda/overview.html#requirements
https://www.vincent-lunot.com/post/an-introduction-to-cuda-in-python-part-1/
https://nyu-cds.github.io/python-numba/05-cuda/
https://numba.readthedocs.io/en/stable/cuda/examples.html
https://sahilchachra.medium.com/dive-into-basics-of-gpu-cuda-accelerated-programming-using-numba-in-python-a0be21aa00b7
https://towardsdatascience.com/cuda-by-numba-examples-1-4-e0d06651612f
https://thedatafrog.com/en/articles/cuda-kernel-python/

```
 1 '''
 2
 3 MULTIPLICACION DE MATRICES
 4
 5 TESINA UNAM CAR IIMAS
 6 @author: Juan Carlos Lopez
 7
 8 '''
 9
10 import numba
11 from numba import cuda
12 import math
13 import sys
14 import numpy as np
```

```
15
16 import warnings
17 from numba.core.errors import NumbaPerformanceWarning
18
19 kernel = []
20 filtro_imagen = []
21 resultado = []
22
23
24
25 dim_mult = 2
26
27 @cuda.jit
28 def matmul(A, B, C, out):
29     i, j = cuda.grid(2)
30
31     out[i][j] = 1 + j / 10
32
33
34     for i in range(i, dim_mult):
35         for j in range(j, dim_mult):
36             suma = 0
37             for k in range(A.shape[1]):
38                 suma += A[i, k] * B[k,j]
39             C[i, j] = suma
40
41     ## fin for i-j-k
42
43 ### end matmul()
44
45 def revisarVersionGPU():
46     print("Version de Python : ", sys.version)
47     print("Numpy version : ", np.__version__)
48     print("Numba version : ",numba.__version__)
49
50     warnings.simplefilter("ignore", category=NumbaPerformanceWarning)
51     print()
52
53     if cuda.is_available() == True:
54         print("GPU DISPONIBLE")
55         print("================================")
56         cuda.detect()
57         print("================================")
58         print()
59     else:
60         print("NO HAY GPU DISPONIBLE. \n")
61
62
63 ### end revisarVersionGPU()
64
65 def imprimirMatriz(matriz):
66     for i in matriz:
67         print(i)
68
69 ### end imprimirMatriz()
70
71 def iniciarMatrices():
72     global kernel
73     global filtro_imagen
74     global resultado
75
76     global dim_mult
77     global indices
78
```

```
 79     dim_mult = int(input("Ingresar la dimension de la multiplicacion : "))
 80
 81     valores = 10
 82
 83     kernel = np.random.randint(valores, size=(dim_mult, dim_mult))
 84     kernel = kernel.astype(int)
 85
 86     indices = np.zeros((dim_mult, dim_mult))
 87
 88     print("kernel = ")
 89     imprimirMatriz(kernel)
 90     print(kernel.shape)
 91     print()
 92
 93     filtro_imagen = np.identity(dim_mult).astype(int)
 94     print("filtro imagen =")
 95     imprimirMatriz(filtro_imagen)
 96     print(filtro_imagen.shape)
 97     print()
 98
 99     resultado = np.zeros((dim_mult, dim_mult)).astype(int)
100
101 ### end iniciarMatrices()
102
103
104 def configBloqueGPU():
105     global kernel_device
106     global filtro_device
107     global resultado_device
108     global stream
109
110     global indices
111     global indices_device
112
113     stream = cuda.stream()
114
115     ### ENVIAR GPU TRANSFERENCIA DE DATOS HOST --> DEVICE
116     kernel_device = cuda.to_device(kernel, stream=stream)
117     filtro_device = cuda.to_device(filtro_imagen)
118     resultado_device = cuda.to_device(resultado)
119
120     indices_device = cuda.to_device(indices)
121
122     ## HILOS POR BLOQUE (BLOCKSIZE)
123     HPB_X = 16
124     HPB_Y = 16
125
126     # CONFIGURACION DE LOS BLOQUES GPU
127     threadsperblock = (HPB_X, HPB_Y)
128     #print(threadsperblock)
129
130     blockspergrid_x = math.ceil(resultado.shape[0] / threadsperblock[0])
131     blockspergrid_y = math.ceil(resultado.shape[1] / threadsperblock[1])
132
133     blockspergrid = (blockspergrid_x, blockspergrid_y)
134
135     return blockspergrid, threadsperblock
136
137 ### end configBloqueGPU()
138
139 def multGPU():
140     global kernel_device
141     global filtro_device
142     global resultado_device
```

```
143
144     global indices_device
145
146     revisarVersionGPU()
147
148     iniciarMatrices()
149     blockspergrid, threadsperblock = configBloqueGPU()
150
151     inicioEvento = cuda.event()
152     finEvento = cuda.event()
153
154     inicioEvento.record(stream=stream)
155
156     print("gridsize ", blockspergrid)
157     print("blocksize ", threadsperblock)
158     print("=========================================")
159     print("Numero total de hilos x = ", blockspergrid[0] * threadsperblock[0] )
160     print("Numero total de hilos y = ", blockspergrid[1] * threadsperblock[1] )
161     print("=========================================")
162
163
164     matmul[blockspergrid,       threadsperblock](kernel_device,       filtro_device,
resultado_device, indices_device)
165
166     finEvento.record(stream=stream)
167
168     finEvento.wait(stream=stream)
169
170     finEvento.synchronize()
171
172     tiempo_medido = inicioEvento.elapsed_time(finEvento)
173     tiempo_seg = tiempo_medido / 1000.0
174
175     indices = indices_device.copy_to_host()
176
177     print("----INDICES---#")
178     print(indices)
179     print(indices.shape)
180     print(indices.size)
181     print("-------#")
182
183     resultado = resultado_device.copy_to_host()
184     print("resultado")
185     imprimirMatriz(resultado)
186     print(resultado.shape)
187     print(resultado.dtype)
188     print()
189
190     valorMax = np.max(resultado)
191
192     print("ValorMax = ", valorMax)
193     print()
194
195     verificar_mult(resultado)
196
197     print()
198     print("==============================")
199     print()
200     print(f"Tiempo de ejecucion del kernel {tiempo_medido:.4f} mseg | {tiempo_seg:.4f}
seg")
201     print("==============================")
202
203 ### end multGPU()
204
```

```
205
206 def verificar_mult(res):
207
208     multiplica = np.dot(kernel, filtro_imagen)
209     print("------------VERIFICACION DE RESULTADOS.----------")
210     imprimirMatriz(multiplica)
211     print(multiplica.shape)
212     print(multiplica.dtype)
213     print("-------------------------------------------------")
214
215     if np.array_equal(multiplica, res):
216         print("Las multiplicaciones coinciden. \n ")
217         print("Resultado obtenido con CUDA")
218         print(res)
219     else:
220         print("Las multiplicaciones no coinciden.\n")
221         print("Diferencia")
222         print(res-multiplica)
223
224 ### end verificar_mult()
225
226 def main():
227
228     if cuda.is_available() == True:
229         print("MULTIPLICACION DE MATRICES EN GPU CON NUMBA CUDA ")
230         print()
231         multGPU()
232     else:
233         print("NO HAY GPU DISPONIBLE \n SE NECESITA DE UNA GPU PARA ESTE PROGRAMA.
\n")
234
235
236 ### end main()
237
238 if __name__ == '__main__':
239     main()
240
241
```

Código de red convolucional AlexNet

REFERENCIAS DE CÓDIGO DE RED CONVOLUCIONAL ALEXNET
PARA LA CONSULTA DE REFERENCIAS COMPLETAS CONSULTAR LA SECCION DE BILBIOGRAFÍA Y REFERENCIAS

https://github.com/bentrevett/pytorch-image-classification

https://github.com/MLDawn/MLDawn-Projects/blob/main/Pytorch/Brain-Tumor-Detector/MRI-Brain-Tumor-Detecor.ipynb

https://discuss.pytorch.org/t/how-do-i-check-the-number-of-parameters-of-a-model/4325/23?page=2

https://pytorch.org/docs/stable/generated/torch.set_num_threads.html#torch.set_num_threads

https://www.geeksforgeeks.org/python-program-to-detect-the-edges-of-an-image-using-opencv-sobel-edge-detection/

http://www.adeveloperdiary.com/data-science/computer-vision/how-to-implement-sobel-edge-detection-using-python-from-scratch/

https://github.com/TylerYep/torchinfo

==

Clasificación de tumores cerebrales por medio de base de datos de imágenes en MRI

Universidad Nacional Autónoma de México UNAM IIMAS

Proyecto de tesina Especialidad de Cómputo de alto rendimiento

Tutor Dr. Hector Pérez Benitez.

Juan Carlos López Núñez.

Descripción

El siguiente programa busca realizar una clasificación de un conjunto de imágenes MRI dividido en 3 clases

- anatomical.
- functional.
- metaanalytic.

Se tiene un total de 245 imágenes como conjunto de entrenamiento en formato JPG.

Para el proceso de clasificación se usó una red convolucional AlexNet con un tamaño de lote de 256 y 25, 50 épocas para reducir la tasa de error.

Las pruebas se realizaron con un CPU Intel Core i-7 de 4 núcleos con 8 hilos en Windows 10 y una GPU NVIDIA GTX 1050 para laptop. Se utilizo Anaconda junto con Jupyter Notebook con Python 3.8.13 para este experimento.

==

```
import numpy as np

import glob

import matplotlib.pyplot as plt

from sklearn.metrics import confusion_matrix, accuracy_score

import random

import cv2

import sys

import os
```

```
from pathlib import Path
#import os
#from PIL import Image
#from glob import glob

%matplotlib inline
# ================================================================================
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.optim.lr_scheduler import _LRScheduler
import torch.utils.data as data

import torchvision.transforms as transforms
import torchvision.datasets as datasets
from torchvision import models

from sklearn import decomposition
from sklearn import manifold
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
from tqdm.notebook import tqdm, trange

import copy
import time
# ================================================================================
torch.__version__
# ================================================================================
torchvision.__version__
# ================================================================================
!python --version
# ================================================================================
# https://superfastpython.com/number-of-cpus-python/
```

```
# https://www.programcreek.com/python/example/2473/multiprocessing.cpu_count
# Number of Logical CPU cores
os.cpu_count()
# ============================================================================
#informacion de tarjeta NVIDIA version de CUDA
# https://discuss.pytorch.org/t/requirements-txt-for-multiple-cuda-architectures/135734
# https://www.gcptutorials.com/post/check-cuda-version-in-pytorch
#nvcc --version
!nvcc -V
# ============================================================================
# Returns the number of threads used for parallelizing CPU operations
torch.get_num_threads()
# ============================================================================
# Returns the number of threads used for inter-op parallelism on CPU (e.g. in JIT interpreter)
torch.get_num_interop_threads()
#torch.set_num_interop_threads() # Inter-op parallelism
#torch.set_num_threads() # Intra-op parallelism
# ============================================================================
# https://pytorch.org/docs/stable/generated/torch.set_num_threads.html#torch.set_num_threads
# Sets the number of threads used for intraop parallelism on CPU.

# https://pytorch.org/docs/stable/notes/cpu_threading_torchscript_inference.html
"""
To ensure that the correct number of threads is used, set_num_threads must be called before
running eager, JIT or autograd code.
-------------------------------------------------------------------------------------------

intra-operand and inter-operand

If you have 4 cores and need to do, say, 8 matrix multiplications (with separate data)
you could use 4 cores to do each matrix multiplication (intra-op-parallelism).
Or you could use a single core for each op and run 4 of them in parallel (inter-op-parallelism).
In training, you also might want to have some cores for the dataloader, for inference, the JIT
can parallelize things (I think).
"""
```

```
NUM_HILOS = 4
torch.set_num_threads(NUM_HILOS)  # numero maximo de hilos soportados por el procesador
# ================================================================================
# Returns the number of threads used for parallelizing CPU operations
torch.get_num_threads()
# ================================================================================
SEED = 1234


random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic=True
# ================================================================================
if torch.cuda.is_available():
    device = torch.device('cuda:0')
    print("CUDA disponible")
else:
    device = torch.device('cpu')
    print("CPU disponible")
# ================================================================================
# leer las imagenes
# ---------------
# OPCIONAL
# -------------
anatomical = []
functional = []
metaanalytic = []


for f in glob.iglob("./datasetMRI/entrenamiento/anatomical/*.jpg"):
    img = cv2.imread(f)
    img = cv2.resize(img,(128,128))
    b, g, r = cv2.split(img)
    img = cv2.merge([r,g,b])
    #img = Image.open(f).convert("RGB")
```

```
    #print(img, np.asarray(img).shape)

    anatomical.append(img)

for f in glob.iglob("./datasetMRI/entrenamiento/functional/*.jpg"):
    img = cv2.imread(f)
    img = cv2.resize(img,(128,128))
    b, g, r = cv2.split(img)
    img = cv2.merge([r,g,b])
    #img = Image.open(f).convert("RGB")
    #print(img, np.asarray(img).shape)

    functional.append(img)

for f in glob.iglob("./datasetMRI/entrenamiento/metaanalytic/*.jpg"):
    img = cv2.imread(f)
    img = cv2.resize(img,(128,128))
    b, g, r = cv2.split(img)
    img = cv2.merge([r,g,b])
    #img = Image.open(f).convert("RGB")
    #print(img, np.asarray(img).shape)

    metaanalytic.append(img)
# ================================================================================
# --------------
# OPCIONAL
# -------------
directorio = '../datasetMRI'

# pasar a arreglos numpy
anatomical = np.array(anatomical)
functional = np.array(functional)
metaanalytic = np.array(metaanalytic)

#concatenar
```

```
conjuntoCompleto = np.concatenate((anatomical, functional, metaanalytic))
# pasarlas a tensores
anatomical_t = torch.from_numpy(anatomical)
functional_t = torch.from_numpy(functional)
metaanalytic_t = torch.from_numpy(metaanalytic)
conjuntoCompleto_t = torch.from_numpy(conjuntoCompleto)
# ============================================================================
anatomical_t.shape
# ============================================================================
functional_t.shape
# ============================================================================
# ============================================================================
metaanalytic_t.shape
# ============================================================================
# --------------
# OPCIONAL
# --------------
# la siguiente funcion despliega las imagenes previamente mostradas


def plot_images(anatomical, functional, metaanalytic, num= 5):
    anatomical_imgs = anatomical[np.random.choice(anatomical.shape[0], num, replace=False)]
    functional_imgs = functional[np.random.choice(functional.shape[0], num, replace=False)]
    metaanalytic_imgs = metaanalytic[np.random.choice(metaanalytic.shape[0], num, replace=False)]


    plt.figure(figsize=(16,9))
    for i in range(num):
        plt.subplot(1, num, i+1)
        plt.title('anatomical')
        plt.imshow(anatomical_imgs[i])


    plt.figure(figsize=(16,9))
    for i in range(num):
        plt.subplot(1, num, i+1)
        plt.title('functional')
        plt.imshow(functional_imgs[i])
```

```
    plt.figure(figsize=(16,9))
    for i in range(num):
        plt.subplot(1, num, i+1)
        plt.title('metaanalytic')
        plt.imshow(metaanalytic_imgs[i])
# ========================================================================
#caminos de conjuntos de prueba y entrenamiento
#from pathlib import Path

datos_entrenamiento = Path('datasetMRI/entrenamiento/')
datos_prueba = Path('datasetMRI/prueba/')
# ========================================================================
# definicion de transforms
train_transforms = transforms.Compose([
                    transforms.RandomRotation(5),
                    transforms.Resize(size=(32,32)),
                    transforms.RandomHorizontalFlip(0.5),
                    transforms.RandomCrop(32, padding=2),
                    transforms.ToTensor(),
                    #transforms.Normalize([0.491, 0.482, 0.447], [0.247, 0.243, 0.261])
                    #transforms.Normalize(mean=means,std=stds
                    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))

                ])
test_transforms = transforms.Compose([
                    transforms.ToTensor(),
                    transforms.Resize(size=(32,32)),
                    #transforms.Normalize([0.491, 0.482, 0.447], [0.247, 0.243, 0.261])
                    #transforms.Normalize(mean=means, std=stds)
                    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                   ])

datos_entrenamiento = datasets.ImageFolder(root=datos_entrenamiento, # target folder of images
                        transform=train_transforms, # transforms to perform on data (images)
```

```
                        )
datos_prueba = datasets.ImageFolder(root=datos_prueba,
                        transform=test_transforms)

print(f"Datos de entrenamiento:\n{datos_entrenamiento}\nDatos de prueba:\n{datos_prueba}")
# ================================================================================
# crear un conjunto de validacion a partir del conjunto de entrenamiento
VALID_RATIO = 0.9

n_train_examples = int(len(datos_entrenamiento) * VALID_RATIO)
n_valid_examples = len(datos_entrenamiento) - n_train_examples

datos_entrenamiento, valid_data = data.random_split(datos_entrenamiento,
                                [n_train_examples, n_valid_examples])
# ================================================================================
# ...and ensure our validation set uses the test transforms.

valid_data = copy.deepcopy(valid_data)
valid_data.dataset.transform = test_transforms
# ================================================================================

print(f'Numero de ejemplos de entrenamiento: {len(datos_entrenamiento)}')
print(f'Numero de ejemplos de validacion: {len(valid_data)}')
print(f'Numero de ejemplos de prueba: {len(datos_prueba)}')
# ================================================================================
# plotear imagenes del conjunto de datos (alexnet)
def plot_images(images, labels, classes, normalize=False):

    n_images = len(images)

    rows = int(np.sqrt(n_images))
    cols = int(np.sqrt(n_images))

    fig = plt.figure(figsize=(10, 10))
```

```
    for i in range(rows*cols):

        ax = fig.add_subplot(rows, cols, i+1)

        image = images[i]

        if normalize:
            image_min = image.min()
            image_max = image.max()
            image.clamp_(min=image_min, max=image_max)
            image.add_(-image_min).div_(image_max - image_min +1e-5)

        ax.imshow(image.permute(1,2,0).cpu().numpy())
        ax.set_title(classes[labels[i]])
        ax.axis('off')
# ======================================================================
N_IMAGES = 25

images, labels = zip(*[(image, label) for image, label in
                [datos_entrenamiento[i] for i in range(N_IMAGES)]])

classes = datos_prueba.classes

plot_images(images, labels, classes)
# ======================================================================
# A solution to this is to renormalize the images so each pixel is between  [0,1] .
# This is done by clipping the pixel values between the maximum and minimum within
# an image and then scaling each pixel between  [0,1]  using these maximum and minimums.

plot_images(images, labels, classes, normalize=True)
# ======================================================================
# normalize the images
def normalize_image(image):
    image_min = image.min()
    image_max = image.max()
```

```
    image.clamp_(min=image_min, max=image_max)
    image.add_(-image_min).div_(image_max - image_min + 1e-5)
    return image
# ====================================================================
# As before, we'll check what images look like with Sobel filters applied to them.
# the filters are 2 dimensional
def plot_filter(images, filter, normalize=True):

    images = torch.cat([i.unsqueeze(0) for i in images], dim=0).cpu()
    filter = torch.FloatTensor(filter).unsqueeze(0).unsqueeze(0).cpu()
    filter = filter.repeat(3,3,1,1)

    n_images = images.shape[0]

    filtered_images = F.conv2d(images, filter)

    images = images.permute(0,2,3,1)
    filtered_images = filtered_images.permute(0,2,3,1)

    fig = plt.figure(figsize=(25,5))

    for i in range(n_images):

        image = images[i]

        if normalize:
            image = normalize_image(image)

        ax = fig.add_subplot(2, n_images, i+1)
        ax.imshow(image)
        ax.set_title('Original')
        ax.axis('off')

        image = filtered_images[i]
```

```
    if normalize:
        image = normalize_image(image)

    ax = fig.add_subplot(2, n_images, n_images+i+1)
    ax.imshow(image)
    ax.set_title('Filtered')
    ax.axis('off')

# ======================================================================
# FILTROS DE SOBEL
# filtro para detectar lineas horizontales
# http://www.adeveloperdiary.com/data-science/computer-vision/how-to-implement-sobel-edge-detection-using-python-from-scratch/
# https://www.geeksforgeeks.org/python-program-to-detect-the-edges-of-an-image-using-opencv-sobel-edge-detection/

N_IMAGES = 10

images = [image for image, label in [datos_entrenamiento[i] for i in range(N_IMAGES)]]

horizontal_filter = [[-1, -2, -1],
                     [ 0,  0,  0],
                     [ 1,  2,  1]]

plot_filter(images, horizontal_filter)

# ======================================================================
# filtro para detectar lineas verticales
vertical_filter = [[-1, 0, 1],
                   [-2, 0, 2],
                   [-1, 0, 1]]

plot_filter(images, vertical_filter)

# ======================================================================
#subsampling /pooling
```

```
def plot_subsample(images, pool_type, pool_size, normalize=True):

    images = torch.cat([i.unsqueeze(0) for i in images], dim=0).cpu()

    if pool_type.lower() == 'max':
        pool = F.max_pool2d
    elif pool_type.lower() in ['mean', 'avg']:
        pool = F.avg_pool2d
    else:
        raise ValueError(f'pool_type must be either max or mean, got: {pool_type}')

    n_images = images.shape[0]

    pooled_images = pool(images, kernel_size=pool_size)

    images = images.permute(0, 2, 3, 1)
    pooled_images = pooled_images.permute(0, 2, 3, 1)

    fig = plt.figure(figsize=(25, 5))

    for i in range(n_images):

        image = images[i]

        if normalize:
            image = normalize_image(image)

        ax = fig.add_subplot(2, n_images, i+1)
        ax.imshow(image)
        ax.set_title('Original')
        ax.axis('off')

        image = pooled_images[i]

        if normalize:
```

```
        image = normalize_image(image)

    ax = fig.add_subplot(2, n_images, n_images+i+1)
    ax.imshow(image)
    ax.set_title('Subsampled')
    ax.axis('off')

# ================================================================================
plot_subsample(images, 'max', 2)

# ================================================================================
# ==================================
# FIN DEL PROPRECESAMIENTO DE DATOS
# CREANDO LOS ITERADORES
# ==================================

# TAM DE LOTE
# 16, 32, 64, 128, 256, 512, 1024, 2048
BATCH_SIZE = 256  # 256

iterador_entrenamiento = data.DataLoader(datos_entrenamiento,
                        shuffle=True,
                        batch_size=BATCH_SIZE)

iterador_validacion = data.DataLoader(valid_data,
                        batch_size=BATCH_SIZE)

iterador_prueba = data.DataLoader(datos_prueba,
                       batch_size=BATCH_SIZE)

# ================================================================================
# ==================================
# DEFINICION DEL MODELO
# IMPLEMENTACION DE ALEXNET
# ==================================
```

```
class AlexNet(nn.Module):
    def __init__(self, output_dim):
        super().__init__()

        self.features = nn.Sequential(
            nn.Conv2d(3, 64, 3, 2, 1),  # in_channels, out_channels, kernel_size, stride, padding
            nn.MaxPool2d(2),  # kernel_size
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 192, 3, padding=1),
            nn.MaxPool2d(2),
            nn.ReLU(inplace=True),
            nn.Conv2d(192, 384, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, 3, padding=1),
            nn.MaxPool2d(2),
            nn.ReLU(inplace=True)
        )

        self.classifier = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(256*2*2, 4096),    #
            #nn.AdaptiveMaxPool2d((28,7680)),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(4096, 4096),
            #nn.AdaptiveMaxPool2d((7680,7680)),
            nn.ReLU(inplace=True),
            nn.Linear(4096, output_dim),
        )

    def forward(self, x):
        x = self.features(x)
        h = x.view(x.shape[0], -1)
```

```
        h = x.view(x.size(0), -1)  #flatten the vector

        x = self.classifier(h)

        return x, h

# ================================================================================

# INSTANCIA DEL MODELOS CON UN NUMERO DEFINIDO DE CLASES
OUTPUT_DIM = 10

model = AlexNet(OUTPUT_DIM)
# ================================================================================
# CONTEO DE PARAMETROS
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)



print(f'El modelo tiene {count_parameters(model):,} parametros entrenables')
# ================================================================================
from prettytable import PrettyTable
# https://discuss.pytorch.org/t/how-do-i-check-the-number-of-parameters-of-a-model/4325/23?page=2

def count_parameters(model):
    table = PrettyTable(["Modules", "Parameters"])
    total_params = 0
    for name, parameter in model.named_parameters():
        if not parameter.requires_grad:
            continue
        param = parameter.numel()
        table.add_row([name, param])
        total_params+=param
    print(table)
    print(f"Total Trainable Params: {total_params}")
    return total_params
```

```
count_parameters(model)

# ========================================================================

# ========================
# ENTRENAR EL MODELO
# ========================
def initialize_parameters(m):
    if isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(m.weight.data, nonlinearity='relu')
        nn.init.constant_(m.bias.data, 0)
    elif isinstance(m, nn.Linear):
        nn.init.xavier_normal_(m.weight.data, gain=nn.init.calculate_gain('relu'))
        nn.init.constant_(m.bias.data, 0)



# ========================================================================
model.apply(initialize_parameters)
# ========================================================================
modelAlexNet = model
param_size = 0
for param in modelAlexNet.parameters():
    param_size += param.nelement() * param.element_size()
buffer_size = 0
for buffer in modelAlexNet.buffers():
    buffer_size += buffer.nelement() * buffer.element_size()

size_all_mb = (param_size + buffer_size) / 1024**2

print('model size: {:.3f}MB'.format(size_all_mb))
# ========================================================================
#ESTADISTICAS
# https://github.com/TylerYep/torchinfo
from torchinfo import summary
summary(modelAlexNet, input_size=(BATCH_SIZE ,3 , 32, 32))
```

```
# ========================================================================

class LRFinder:

    def __init__(self, model, optimizer, criterion, device):


        self.optimizer = optimizer

        self.model = model

        self.criterion = criterion

        self.device = device


        torch.save(model.state_dict(), 'init_params.pt')


    def range_test(self, iterator, end_lr=10, num_iter=100,

                   smooth_f=0.05, diverge_th=5):


        lrs = []

        losses = []

        best_loss = float('inf')


        lr_scheduler = ExponentialLR(self.optimizer, end_lr, num_iter)


        iterator = IteratorWrapper(iterator)


        for iteration in range(num_iter):


            loss = self._train_batch(iterator)


            lrs.append(lr_scheduler.get_last_lr()[0])


            # update lr

            lr_scheduler.step()


            if iteration > 0:

                loss = smooth_f * loss + (1 - smooth_f) * losses[-1]


            if loss < best_loss:
```

```
            best_loss = loss

        losses.append(loss)

        if loss > diverge_th * best_loss:
            print("Stopping early, the loss has diverged")
            break

    # reset model to initial parameters
    model.load_state_dict(torch.load('init_params.pt'))

    return lrs, losses

def _train_batch(self, iterator):

    self.model.train()

    self.optimizer.zero_grad()

    x, y = iterator.get_batch()

    x = x.to(self.device)
    y = y.to(self.device)

    y_pred, _ = self.model(x)

    loss = self.criterion(y_pred, y)

    loss.backward()

    self.optimizer.step()

    return loss.item()
```

```
class ExponentialLR(_LRScheduler):
    def __init__(self, optimizer, end_lr, num_iter, last_epoch=-1):
        self.end_lr = end_lr
        self.num_iter = num_iter
        super(ExponentialLR, self).__init__(optimizer, last_epoch)

    def get_lr(self):
        curr_iter = self.last_epoch
        r = curr_iter / self.num_iter
        return [base_lr * (self.end_lr / base_lr) ** r
                for base_lr in self.base_lrs]


class IteratorWrapper:
    def __init__(self, iterator):
        self.iterator = iterator
        self._iterator = iter(iterator)

    def __next__(self):
        try:
            inputs, labels = next(self._iterator)
        except StopIteration:
            self._iterator = iter(self.iterator)
            inputs, labels, *_ = next(self._iterator)

        return inputs, labels

    def get_batch(self):
        return next(self)
# ====================================================================
START_LR = 1e-7

optimizer = optim.Adam(model.parameters(), lr=START_LR)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
criterion = nn.CrossEntropyLoss()
```

```
model = model.to(device)
criterion = criterion.to(device)
# ================================================================================
END_LR = 10
NUM_ITER = 100

lr_finder = LRFinder(model, optimizer, criterion, device)
lrs, losses = lr_finder.range_test(iterador_entrenamiento, END_LR, NUM_ITER)

# ================================================================================
# plot the learning rate against the loss
def plot_lr_finder(lrs, losses, skip_start=5, skip_end=5):

    if skip_end == 0:
        lrs = lrs[skip_start:]
        losses = losses[skip_start:]
    else:
        lrs = lrs[skip_start:-skip_end]
        losses = losses[skip_start:-skip_end]

    fig = plt.figure(figsize=(16, 8))
    ax = fig.add_subplot(1,1,1)
    ax.plot(lrs, losses)
    ax.set_xscale('log')
    ax.set_xlabel('Learning rate')
    ax.set_ylabel('Loss')
    ax.grid(True, 'both', 'x')
    plt.show()

# ================================================================================
plot_lr_finder(lrs, losses)
# ================================================================================
FOUND_LR = 1e-3
optimizer = optim.Adam(model.parameters(), lr=FOUND_LR)
```

```
# ========================================================================
# funcion para calcular la precision
def calculate_accuracy(y_pred, y):
    top_pred = y_pred.argmax(1, keepdim=True)
    correct = top_pred.eq(y.view_as(top_pred)).sum()
    acc = correct.float() / y.shape[0]
    return acc
# ========================================================================
def train(model, iterator, optimizer, criterion, device):

    epoch_loss = 0
    epoch_acc = 0

    model.train()   # encender dropout

    for (x, y) in tqdm(iterator, desc="Entrenando", leave=False):

        x = x.to(device)
        y = y.to(device)

        optimizer.zero_grad()

        y_pred, _ = model(x)

        loss = criterion(y_pred, y)

        acc = calculate_accuracy(y_pred, y)

        loss.backward()

        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()
```

```
    return epoch_loss / len(iterator), epoch_acc / len(iterator)

# ================================================================================

def evaluate(model, iterator, criterion, device):


    epoch_loss = 0

    epoch_acc = 0


    model.eval()  # apagar dropout


    with torch.no_grad():


        for (x, y) in tqdm(iterator, desc="Evaluando", leave=False):


            x = x.to(device)

            y = y.to(device)


            y_pred, _ = model(x)


            loss = criterion(y_pred, y)


            acc = calculate_accuracy(y_pred, y)


            epoch_loss += loss.item()

            epoch_acc += acc.item()


    return epoch_loss / len(iterator), epoch_acc / len(iterator)

# ================================================================================

# funcion para medir cuanto tarda una epoca

def epoch_time(start_time, end_time):

    elapsed_time = end_time - start_time

    elapsed_mins = int(elapsed_time / 60)

    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))

    return elapsed_mins, elapsed_secs


# ================================================================================
```

```
"""
-------------------------
PRUEBAS DE ENTRENAMIENTO
-------------------------



Epoca: 25 | Tiempo epoca: 0m 0s
        Entren Perdida: 0.526 | Entren Precision: 80.82%
        Valid Perdida: 0.275 | Valid Precision: 89.29%
-----------------------------------------------
Tiempo de entrenamiento: 26.42 segundos
-----------------------------------------------



Epoca: 50 | Tiempo epoca: 0m 1s
        Entren Perdida: 0.394 | Entren Precision: 86.94%
        Valid Perdida: 0.111 | Valid Precision: 100.00%
-----------------------------------------
Tiempo de entrenamiento: 89.37 segundos
-----------------------------------------


Epoca: 100 | Tiempo epoca: 0m 0s
        Entren Perdida: 0.338 | Entren Precision: 87.76%
        Valid Perdida: 0.076 | Valid Precision: 100.00%
-----------------------------------------------


Tiempo de entrenamiento: 107.81 segundos
-----------------------------------------------
"""


EPOCHS = 25 # 25, 50, 100


best_valid_loss = float('inf')


inicioEntrenamiento = time.time() #inicio de entrenamiento
```

```
for epoch in trange(EPOCHS, desc="Epocas"):

    start_time = time.monotonic()

    train_loss, train_acc = train(model, iterador_entrenamiento, optimizer, criterion, device)
    valid_loss, valid_acc = evaluate(model, iterador_validacion, criterion, device)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'tut3-model.pt')

    end_time = time.monotonic()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    print(f'Epoca: {epoch+1:02} | Tiempo epoca: {epoch_mins}m {epoch_secs}s')
    print(f'\tEnt Perdida: {train_loss:.3f} | Ent Precision: {train_acc*100:.2f}%')
    print(f'\tVal Perdida: {valid_loss:.3f} | Val Precision: {valid_acc*100:.2f}%')

# fin de entrenamiento
finEntrenamiento = time.time()
tiempo = (finEntrenamiento - inicioEntrenamiento)
print('-----------------------------------------------')
print(f"Tiempo de entrenamiento: {tiempo:.2f} segundos")
print('-----------------------------------------------\n')
# ================================================================================
model.load_state_dict(torch.load('tut3-model.pt'))

test_loss, test_acc = evaluate(model, iterador_prueba, criterion, device)
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
# ================================================================================
#EXAMINAR EL MODELO
def get_predictions(model, iterator, device):

    model.eval()
```

```
    images = []
    labels = []
    probs = []

    with torch.no_grad():

        for (x, y) in iterator:

            x = x.to(device)

            y_pred, _ = model(x)

            y_prob = F.softmax(y_pred, dim=-1)

            images.append(x.cpu())
            labels.append(y.cpu())
            probs.append(y_prob.cpu())

    images = torch.cat(images, dim=0)
    labels = torch.cat(labels, dim=0)
    probs = torch.cat(probs, dim=0)

    return images, labels, probs

# ========================================================================
images, labels, probs = get_predictions(model, iterador_prueba, device)

# ========================================================================
# por cada prediccion se obtiene la clase predecida
pred_labels = torch.argmax(probs, 1)
# ========================================================================
# matriz de confusion
def plot_confusion_matrix(labels, pred_labels, classes):
```

```
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(1, 1, 1)
    cm = confusion_matrix(labels, pred_labels)
    cm = ConfusionMatrixDisplay(cm, display_labels=classes)
    cm.plot(values_format='d', cmap='Blues', ax=ax)
    plt.xticks(rotation=20)
# ================================================================================
plot_confusion_matrix(labels, pred_labels, classes)
# ================================================================================
```

APENDICE E

Comandos útiles para diagnóstico de CUDA y cancelación de procesos.

$nvidia-smi

Este comando muestra la información general de la o las GPUs disponibles en el sistema. Esta información muestra de forma general la información de la GPU con datos útiles tales como temperatura, versión de CUDA, modelo de la GPU, entre otras características importantes de la operación de la GPU.

El cuadro de información en la parte inferior muestra los procesos realizados por la GPU en el momento en que se ejecute este comando donde cada proceso tiene un PID asignado. Para poder cancelar un programa en ejecución en la GPU se puede utilizar el comando $sudo kill -9 PID.

$nvidia-smi -q

Despliega una lista completa de las características de la GPU.

$nvidia-smi -L

Muestra la UUID de la GPU que sirve como un número de identificación único para la GPU. Este comando además despliega en una lista las GPUs instaladas en el sistema. Los índices de la GPU comienzan a partir de 0.

$nvidia-smi -l

Esta opción muestra el cuadro de nvidia-smi cada segundo. Para poder cancelar esta opción usar CTRL+C para salir de esta opción desde la terminal.

$nvidia-smi dmon

Muestra una tabla en la cual se muestra una tabla con una serie de características tales como memoria, temperatura entre otros en forma de tabla la cual se imprime cada segundo en la terminal en formato de tabla.

$watch -n 1 nvidia-smi

Muestra la tabla que se puede obtener por medio de nvidia-smi pero con actualizaciones cada determinado tiempo, en este ejemplo se reflejan cambios cada segundo junto con un reloj de la computadora. Este comando es útil para monitorear la actividad de la GPU en pruebas extensas, en especial si se busca monitorear la memoria disponible del dispositivo y la temperatura de la GPU durante el experimento. Esta opción solamente está disponible con Linux.

$ nvidia-smi –query-gpu= … --format=csv

Esta opción despliega una tabla con las características de la GPU las cuales se muestran en forma de tabla en formato csv en la terminal. Este comando es personalizable en la selección de las características que se buscan consultar de la GPU. Se recomienda ver la tabla completa de características de la GPU para determinar cuál es la característica que se busca.

Para información adicional referirse al manual de nvidia-smi.

https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf