UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Adaptación de algoritmos de aprendizaje automático para su ejecución sobre GPUs.

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Jesús Vieco Pérez

Tutor: Jon Ander Gómez Adrian

2015 -2016

“El logro más impresionante de la industria del software es su continua anulación de los constantes y asombrosos logros de la industria del hardware”

-- Henry Petroski

Resumen

En este proyecto trabajamos en el desarrollo de algunos algoritmos de aprendizaje automático usados para entrenar redes neuronales, concretamente en la implementación para ser ejecutada en las unidades de procesamiento gráfico (GPUs). Actualmente, las GPUs están disponibles en la mayoría de las tarjetas gráficas porque son utilizadas para realizar las operaciones en paralelo necesarias para obtener el color y la intensidad de los pixeles.

Gracias a la rápida evolución de las GPUs durante los últimos años podemos realizar eficientemente cálculos en paralelo. El objetivo principal de este proyecto es el diseño de algoritmos para ser ejecutados en las GPUs para aprovechar la ventaja de esto. Después se llevará a cabo un estudio comparativo sobre las dos versiones del mismo algoritmo (para las CPUs y para las GPUs). Los resultados del estudio serán representados y comentados en el proyecto. La comparativa se centrará en el efecto que algunos híper-parámetros pueden tener sobre las diferentes implementaciones. Para programar dichos algoritmos usaremos los lenguajes de programación C++ y CUDA. El primero para la implementación para CPU y el segundo para la implementación GPU.

Palabras clave: Aprendizaje automático, red neuronal, GPU, CUDA, C++.

Abstract

In this project we worked in the development of some machine learning algorithms used for training neural networks, in particular the implementation to be run on Graphical Processor Units (GPUs). Currently, GPUs are available on most graphic cards because they are used for performing in parallel the operations needed to get color and intensity of all pixels.

Thanks to the rapid evolution of GPUs during last years we can efficiently perform parallel computations. The main goal of this project is just the design of algorithms to be executed in GPUs in order to take advantage of them. Then, a comparative study of the two versions of same algorithm (for CPUs and for GPUs) will be carried out. The results of that study will be presented and discussed in this project. The study will focus on the effect some hyper-parameters can have on the different implementations. For programming these algorithms we will use the programming languages C++ and CUDA. The first one for the CPU implementation and the second one for the GPU implementation.

Keywords: Machine learning, neural network, GPU, CUDA, C++.

Resum

En este projecte hem treballat en el desenvolupament d'alguns algorismes d'aprenentatge automàtic utilitzats per entrenar xarxes neuronals. En particular la seua implementació per executar-se en GPUs (de l'anglés Graphical Processor Units). Actualment, tenim GPUs disponibles en moltes targetes gràfiques, perquè són utilitzades per operar en paral·lel a l'hora d'estimar el color i la intensitat de cada pixel.

Gràcies a la ràpida evolució de les GPUs als darrers anys, podem abordar càlculs en paral·lel de manera eficient. El principal objectiu d'este projecte és justament el disseny d'algorismes per executar-se en GPUs per treure profit d'elles. Aleshores, es durà a terme un estudi comparatiu de dues versions del mateix algorisme (una per CPUs i l'altra per GPUs). Els resultats d'este estudi es presentarà i discutirà en este projecte. L'estudi farà especial èmfasi en l'efecte que alguns del hiper-paràmetres poden tindre en les diferents implementacions. Per programar estos algorismes farem ús dels llenguatges de programació C++ i CUDA. El primer per executar-se en CPUs i el segon per GPUs.

Paraules clau: Aprenentatge automàtic, xarxa neuronal, GPU, CUDA, C++.

Índice general

Índice de figuras

Índice de tablas

Glosario

Aprendizaje automático: El aprendizaje automático es un campo de la informática que se basa en el reconocimiento de patrones y la teoría del aprendizaje computacional en la inteligencia artificial.

Arquitectura: es el diseño de los componentes y como se distribuyen en el procesador, es junto a la tecnología de fabricación lo que define las características y las posibles prestaciones.

Caché: Una memoria caché es una memoria pequeña integrada en el microprocesador cuyo acceso a los datos es mucho más rápido que al resto de memoria, pero también es mucho más pequeña.

CPU: Unidad central de proceso, es la encargada de realizar el procesamiento de los datos y además el control de las funciones del resto de los componentes del ordenador.

Función de activación: función matemática que se aplica a la entrada de una neurona, normalmente para que los valores recibidos por la neurona sean binarios (0 o 1)

GPU: Unidad de procesamiento gráfico, es un coprocesador dedicado al procesamiento de operaciones en coma flotante, está destinado a liberar carga computacional a la CPU en el cálculo de operaciones gráficas.

Hilo: en el ámbito de los sistemas operativos, un hilo es la unidad más pequeña que puede manejar un procesador, se utilizan frecuentemente en computación para paralelizar programas dividiendo las tareas en hilos o subprocesos.

Neurona: En aprendizaje automático es una función matemática concebida como un modelo de las neuronas biológicas. La neurona artificial recibe una o más entradas y devuelve una sola salida.

Operaciones en coma flotante: son todas las operaciones del juego de instrucciones de un procesador donde los operandos son números reales.

Red neuronal: En aprendizaje automático son una familia de modelos matemáticos inspirados en las redes neuronales biológicas, se utilizan para aproximar funciones en base a un gran número de entradas.

Topología de la red: organización de la red según el tamaño de sus capas de entrada, salida y ocultas además del número de capas ocultas que contiene.

Unidad de control: es uno de los tres bloques funcionales principales en los que se divide una CPU, su función es buscar las instrucciones en la memoria principal, decodificarlas y ejecutarlas.

1. Introducción

El presente trabajo de fin de grado pretende realizar un estudio, en un entorno de programación determinado, sobre el coste computacional de una serie de algoritmos de aprendizaje automático ejecutados sobre la unidad central de procesos (CPU) para entrenar y clasificar con redes neuronales; además de comprobar si es posible y eficiente un modelo de paralelismo mediante su computo en la unidad de procesamiento gráfico (GPU).

Para realizar este estudio se ejecutarán tres algoritmos utilizados en aprendizaje automático y se realizarán una serie de medidas para comprobar su comportamiento variando la cantidad y el tamaño de las capas ocultas de la red neuronal.

Se pretenden extraer conclusiones mediante la comparación de los resultados obtenidos tanto en la CPU como en la GPU, en esta última además se abordará otro estudio variando una serie de parámetros clave que se describirán más adelante.

Motivación

La causa de mayor importancia que lleva a realizar este estudio es la alta complejidad computacional que conllevan las redes neuronales. Pese a que el entrenamiento y clasificación en redes neuronales se basa en operaciones básicas en coma flotante, estas se repiten una gran cantidad de veces.

Dado que la CPU sin recurrir al paralelismo solo puede realizar una operación al mismo tiempo se ha optado por agrupar las operaciones de forma que en vez de ser individuales se conviertan en matriciales, éstas siguen siendo aún operaciones secuenciales. Para eliminar la ejecución de operaciones en serie se hará uso de una GPU dedicada, esto requiere un nuevo lenguaje de programación que nos permita interactuar con la GPU para paralelizar en todo lo posible los cálculos.

Planificación previa

Antes de llevar a cabo el desarrollo se ha realizado una planificación para desarrollar una red neuronal con tres algoritmos distintos: *fordward* (algoritmo de clasificación), *backPropagation* (algoritmo de retro propagación del error) y una variante de los anteriores con la aplicación de un coeficiente de *dropout* (coeficiente de apagado) estos serán explicados más adelante.

Para ello necesitamos un lenguaje de programación de alto nivel orientado a objetos, este será C++. Además para la ejecución en la GPU necesitamos otro lenguaje que nos permita utilizar las funciones de la GPU en C++, para ello utilizaremos CUDA.

Una vez realizada la implementación será necesario un conjunto de datos para el entrenamiento y la clasificación, recurriremos al conjunto de datos de MNIST (1) donde tenemos una base de datos de dígitos manuscritos que describiremos en un apartado posterior.

Estructura de la memoria

El presente trabajo se compone de siete capítulos, y en esta sección nos disponemos a realizar una breve descripción del trabajo realizado en cada uno de ellos.

- Introducción: En este capítulo se describe la motivación para llevar a cabo este proyecto junto con la planificación previa.

- Objetivos: Aquí se desarrollan los objetivos que se desean alcanzar.

- Tecnología utilizada: Este apartado se centra en describir tanto el software como el hardware utilizado.

- Algoritmos a implementar: Este capítulo describe la parte central del proyecto, la descripción del software a generar.

- Desarrollo del proyecto: Este capítulo consiste en la descripción de como se ha organizado el código explicado en la sección anterior y la presentación del conjunto de datos que se utilizará para las pruebas.

- Pruebas y mediciones: Esta es la parte más importante del trabajo, donde cuantificaremos las mediciones realizadas tanto en CPU como en GPU exponiendo los resultados obtenidos.

- Conclusiones: El capítulo recoge una serie de conclusiones extraídas a partir de los resultados del anterior apartado.

2. Objetivos

Para el desarrollo de este trabajo nos centraremos en dos objetivos concretos:

- La traducción de código de algoritmos de aprendizaje automático para ser ejecutados tanto en la CPU como en la GPU
- La cuantificación de la mejora en el tiempo de ejecución de dichos algoritmos tras su cómputo en ambas unidades de procesamiento.

El primer objetivo se centra en la programación de dos algoritmos como son el *forward* y el *back propagation*, que explicaremos más adelante, para el entrenamiento y la clasificación de redes neuronales, además de una aplicación posterior de una técnica conocida como *dropout*. Para ello utilizaremos los lenguajes de programación CUDA y C++.
En el segundo objetivo lo que queremos es realizar un estudio sobre la viabilidad y la mejora sobre el tiempo de ejecución al paralelizar en todo lo posible el código de una red neuronal y variar su configuración.

Como objetivos personales a la hora de realizar este proyecto nos planteamos:

- Aprender y mejorar la programación en los lenguajes C++ y CUDA.
- Ampliar el conocimiento sobre las redes neuronales y los algoritmos implicados en ellas.
- Realizar entrenamientos y pruebas con redes neuronales mediante el uso de conjuntos de datos reales como son los de la base de datos MNIST.

3. Tecnología utilizada

Para el desarrollo de los algoritmos se ha optado por definir una jerarquía de clases común tanto para la ejecución en la GPU como en la CPU, la veremos en el apartado "Desarrollo del proyecto". Esta jerarquía ha sido implementada en el lenguaje C++, además de ser ejecutada en la GPU gracias al enlace de CUDA con C++.

C++

Antes de hablar de C++, necesitamos explicar que es lenguaje de programación, es una herramienta que nos permite definir un algoritmo para que el ordenador realice una tarea específica. Un algoritmo es un conjunto de instrucciones o reglas definidas, ordenadas y finitas que permite realizar una actividad mediante pasos sucesivos que no generen dudas a quien deba realizar dicha actividad. Dados un estado inicial y una entrada, siguiendo los pasos sucesivos se llega a un estado final y se obtiene una solución.

Cada lenguaje de programación posee una sintaxis y un léxico particular, es decir, una forma de escribirse que es diferente en cada lenguaje por la forma que fue creado y por la forma que trabaja su compilador. Un compilador es un programa que se encarga de traducir un lenguaje de programación a otro, esto se utiliza para pasar lenguajes de programación de alto nivel a código máquina para su ejecución en la CPU.

C++ es un lenguaje de programación diseñado en la década de 1980 con la intención de extender el lenguaje C y añadirle orientación a objetos. Para trabajar con C++ elegimos el entorno de programación Visual Studio 2013 que además de contar con un editor de texto cuenta con un compilador de C++ lo que nos permite ejecutar el código en el mismo entrono de desarrollo que lo modificamos.

CUDA

Es una plataforma para la programación paralela creada por NVIDIA (2). Ésta plataforma permite a los desarrolladores utilizar la GPU para programación de propósito general, es decir, para realizar cualquier cálculo.

Inicialmente CUDA fue creado para programar en C, aunque actualmente soporta lenguajes como C++, PYTHON o FORTRAN. Para la programación en CUDA también utilizaremos Visual Studio 2013, con descargar el CUDA Toolkit desde (3) e instalarlo podremos compilar y ejecutar archivos .cu en Visual Studio al igual que hacemos con C++.

Una de las grandes ventajas de ejecutar CUDA en conjunto con C++ es que cuando realizamos la llamada a una función *kernel* desde C++, el programa de C++ continúa

sin esperar la respuesta de CUDA, cuando necesita algún dato que se calcula en ese *kernel* entonces entra en escena un mecanismo de sincronización que hace esperar al programa de C++ a que todos los hilos del *kernel* hayan terminado.

CPU

La unidad central de proceso, aquí será donde se medirán las ejecuciones en secuencial. Para ello utilizaremos un Intel Core i7-3610QM a una velocidad de 2.3GHz. Este procesador que contiene 4 núcleos físicos donde cada núcleo físico contiene 2 lógicos, 8 en total, ha sido probado con el software QwikMark y ha obtenido una potencia de 2.93 GFLOPS (10^9 operaciones en coma flotante por segundo) por núcleo y 23 GFLOPS totales, como la CPU se encargará de procesar los cálculos de forma secuencial el dato de 23 GFLOPS no es relevante para este estudio.

GPU

La unidad de procesamiento gráfico se encargará de ejecutar los algoritmos de forma paralela. En este caso utilizaremos una tarjeta gráfica Nvidia GeForce GT 630m que trabaja a una frecuencia de reloj de 800 MHz, es decir, la CPU tiene una velocidad un 235% superior a la GPU. Esta GPU cuenta con 96 núcleos CUDA con una potencia total de 307.20 GFLOPS teóricos (4).

Como podemos observar, la GPU supera en el número máximo de GFLOPS por mucho a la CPU, pese a que la GPU tiene una menor frecuencia de reloj. Esto es debido a la arquitectura de la GPU ya que su propósito es maximizar el número de núcleos que se encargan de procesar los cálculos reduciendo su caché de datos y reduciendo la monitorización que realiza la unidad de control sobre el código que se está ejecutando. En la siguiente figura podemos ver cómo están constituidas las GPUs y las CPUs.

Figura 1: Arquitectura de la CPU y de la GPU

Esta diferencia no solo produce efectos positivos, limita el tipo de problemas que la GPU puede realizar, debido a la reducción de memoria caché debemos realizar operaciones de alta intensidad aritmética para reducir los accesos a memoria.

Para ejecutar código en la GPU debemos diseñar funciones en CUDA, estas funciones son llamadas *kernels*, utilizan la misma sintaxis de C y son precedidas por "__global__". Para realizar la llamada al código desde C++ se han de indicar el número de bloques y el tamaño de bloque.

CUDA está diseñado para poder ser ejecutado en diferentes GPUs con diferente número de procesadores llamados s*treaming multiprocessors* (SMs), para ello utiliza los bloques como podemos ver en la Figura 2, tenemos un programa definido con 8 bloques y el modelo de ejecución varía para una GPU con 2 SM que para una con 4 SM. Podemos deducir que cuantos más SMs menor será el tiempo de ejecución de un programa CUDA hasta que se supera el número de bloques, es decir, la distribución óptima para un programa CUDA será en una GPU con un número de SM mayor o igual que el número de bloques utilizado por dicho programa.

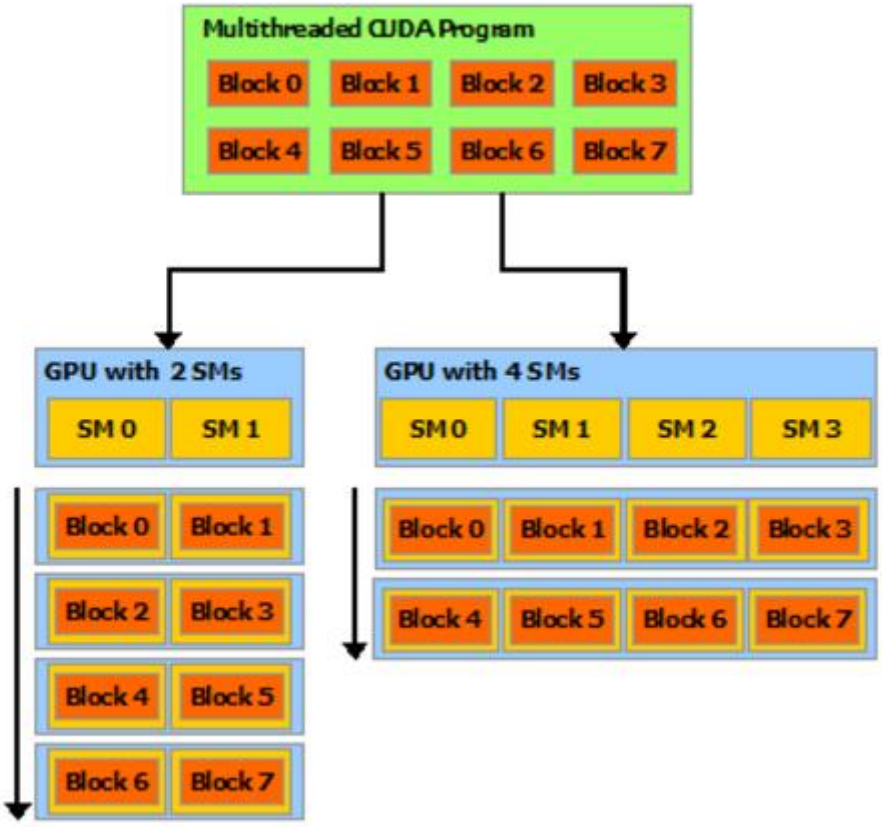

Figura 2: Distribución de un programa CUDA en diferentes GPUs

Internamente cada bloque se distribuye como podemos observar en la Figura 3, en este caso tendremos 6 bloques con un tamaño de bloque de 12 hilos, 4 en horizontal por cada 3 en vertical.

Para conocer la identidad del hilo al que estamos accediendo podemos hacer uso del identificador "*threadIdx*" `dentro del` *kernel*, este identificador tiene 3 valores "x",

"y" y "z" en nuestro ejemplo al ser de dos dimensiones el bloque solo utilizaremos dos donde *threadIdx*.x hará referencia a la posición en horizontal y *threadIdx.y* nos indicará la posición en vertical. Estas posiciones serán llamadas locales, porque hacen referencia a la posición dentro de cada bloque, para calcular la global solo tendremos que calcular el índice del bloque en el que estamos (*blockIdx)*, multiplicar ese índice por el tamaño de bloque (*blockDim)* y sumárselo a la posición local. Para este ejemplo de dos dimensiones la posición global en horizontal se calcularía como en la Tabla 1 y la posición vertical como en la Tabla 2.

```
blockIdx.x * blockDim.x + threadIdx.x;
```

Tabla 1: Cálculo de la posición global en horizontal

```
blockIdx.y * blockDim.y + threadIdx.y;
```

Tabla 2: Cálculo de la posición global en vertical

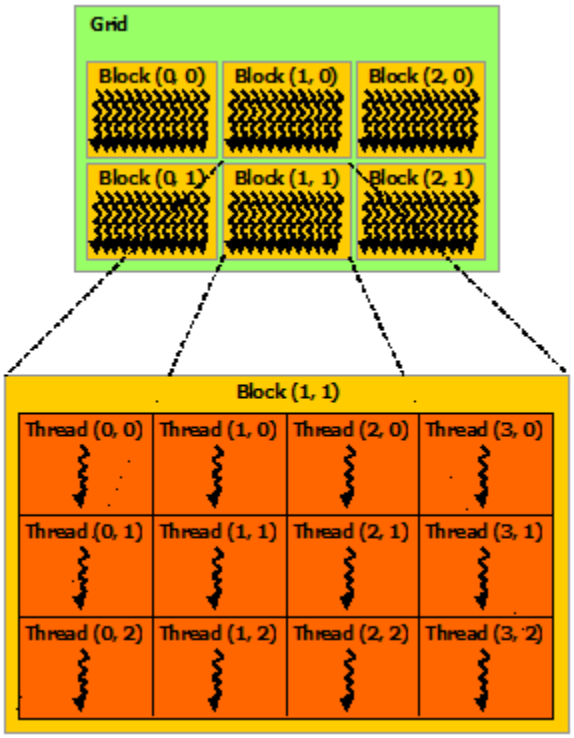

Figura 3: Distribución interna de los bloques en CUDA

Un ejemplo sencillo para ver la diferencia de ejecuciones entre la CPU y la GPU podría ser la suma de vectores, donde el código para la CPU en C++ sería el siguiente, suponiendo n el tamaño de los vectores A y B que se sumarán y se guardarán en C:

```
void sumaVectores(float & A, float & B, float & C)
{
	for (int i = 0;  i < n; i++)
	{
		C[i] = A[i] + B[i];
	}
}
```

Tabla 3: Función suma de vectores en C++

Donde la CPU ejecutará el cálculo de cada elemento de C de forma secuencial, es decir, uno detrás de otro, por lo que tardará n veces lo que tarde en realizar la suma y asignar ese valor a la memoria correspondiente.

En cambio para CUDA tendríamos el siguiente código:

```
__global__ void sumaVectoresCuda (float & A, float & B, float & C)
{
      int i = blockIdx.x * blockDim.x + threadIdx.x;
      C[i] = A[i] + B[i];
}
```

Tabla 4: Función suma de vectores en CUDA

Que será llamado desde C++ con el siguiente código:

```
dim3 tamañoDeBloque(BLOCK_SIZE, 1);
dim3 numeroDebloques(n / BLOCK_SIZE, 1);

sumaVectoresCuda  <<numeroDebloques, tamañoDeBloque>>
```

Tabla 5: Llamada a suma de vectores en CUDA desde C++

Donde BLOCK_SIZE es el tamaño que escogeremos de bloque y ha de ser un divisor de n, más adelante veremos cómo evitar que tenga que serlo. Por tanto la GPU en cada núcleo calculará solo un elemento de C, realizando todos a la vez con n núcleos, por tanto el tiempo correspondiente en la GPU serían lo que tarde en realizar la suma y asignar ese valor a la memoria correspondiente. Pero aquí encontramos el primer problema, la memoria RAM de la GPU no es la misma que la CPU, por tanto deberíamos copiar los vectores de la CPU a la memoria de vídeo (RAM de la GPU) y esto tiene un coste que estudiaremos en el apartado 6. Pruebas y mediciones. Para pasar de la memoria RAM (CPU) a la memoria de video usaremos la siguiente función:

```
cudaMemcpy(vectorGPU, vectorCPU, tamaño, cudaMemcpyHostToDevice);
```

Tabla 6: Función para copiar elementos desde la CPU a la GPU

Donde tenemos que tener reservada la memoria en la GPU con:

```
cudaMalloc( (void**)& vectorGPU, tamaño );
```

Tabla 7: Función reservar memoria en la GPU

Además una vez calculada la suma de vectores debemos devolver el resultado a la CPU con:

```
cudaMemcpy(vectorCPU, vectorGPU, tamaño, cudaMemcpyDeviceToHost);
```

Tabla 8: Función para copiar elementos desde la GPU a la CPU

Si tomamos como x el tiempo de la suma y la asignación en memoria tenemos que el tiempo en CPU es n*x y el tiempo en GPU es el envío de A y B a la memoria de vídeo + x + el tiempo de envío del vector C a la RAM. En un futuro apartado estudiaremos el coste de cada uno para diferentes tamaños.

Para solucionar la suma de vectores en la GPU cuando n no es divisible por el tamaño de bloque tenemos que realizar la siguiente modificación en el código de la llamada desde C++:

```
dim3 tamañoDeBloque (BLOCK_SIZE, 1);
```

```
dim3 numeroDebloques ((n + BLOCK_SIZE -1) / BLOCK_SIZE, 1);

sumaVectoresCuda  <<numeroDebloques, tamañoDeBloque>>
```

Tabla 9: Llamada a suma de vectores en CUDA desde C++ con n que no es divisible por el tamaño de bloque

Con esta modificación lo que hacemos es generar más hilos que elementos hay en la suma, por tanto tendremos que hacer una modificación al *kernel* de CUDA para que no intente acceder a un elemento del vector cuyo índice sea mayor que la longitud del vector:

```
__global__ void sumaVectoresCuda (float & A, float & B, float & C)
{
      int i = threadIdx.x;
      if (i < n)
      {
            C[i] = A[i] + B[i];
      }
}
```

Tabla 10: Función suma de vectores en CUDA para cualquier tamaño n

4. Algoritmos a implementar

Antes de describir los algoritmos y sobre que estructura han sido desarrollados debemos conocer qué son y cómo se gestó la idea de las redes neuronales. Para ello debemos describir que es un modelo matemático de una neurona y las conexiones que se han de realizar para que un grupo de éstas puedan llegar a formar lo que actualmente llamamos redes neuronales.

Historia y modelado de una neurona

El modelo matemático que fue generado en 1943 por McCulloch y Pitt junto con la regla descrita por Hebb en 1949 que modela el aprendizaje de las neuronas fueron la base para que en 1957 Rosenblat introduzca lo que hoy llamamos perceptrón. Una neurona es una célula especializada y caracterizada por poseer una cantidad de canales llamados dendritas y otro canal llamado axón, en nuestro modelo matemático las dendritas serían las entradas y el axón la salida.

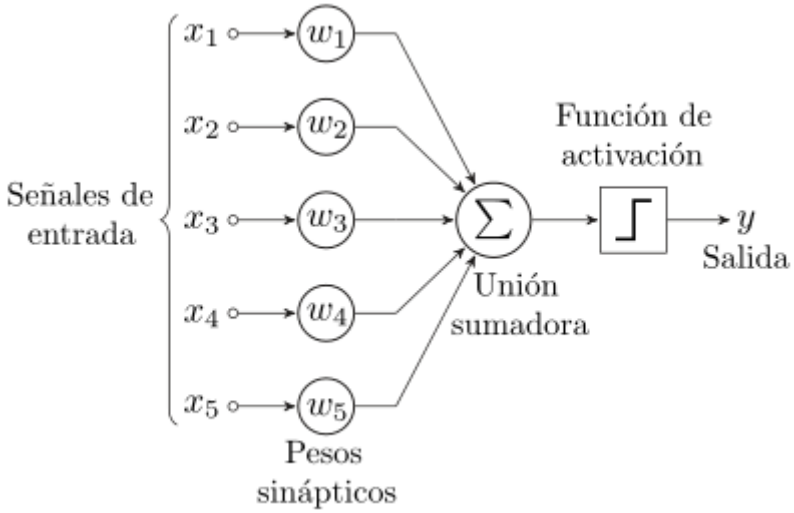

Figura 4: Esquema del perceptrón

Por tanto podemos definir que el perceptrón es una función que toma una serie de entradas y nos devuelve una salida. Realmente el perceptrón internamente lo que hace es multiplicar un valor w_i (llamados pesos sinápticos) para cada entrada i y sumar el resultado de todas estas operaciones, si adaptamos las w_i podemos hacer que el perceptrón nos devuelva un 0 o un 1 dependiendo de la función de activación como podemos ver en la Figura 4, con ello lo que tenemos es un clasificador en dos clases con una frontera de discriminación lineal.

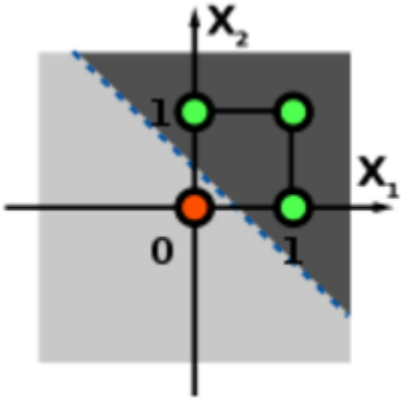

Figura 5: Ejemplo de perceptrón en dos dimensiones

Como podemos ver en la Figura 5, el perceptrón ha sido entrenado para recibir 2 entradas (las dos coordenadas del punto) y es capaz de separar unas muestras de otras de forma lineal. El aprendizaje del perceptrón se consigue mediante la clasificación y corrección de los pesos mediante una serie de muestras para buscar los w_i que minimicen el número de errores al clasificar. Para explicar el algoritmo definimos algunas variables primero:

- x_i denota el elemento en la posición i en el vector de la entrada
- w_i el elemento en la posición i en el vector de peso
- y denota la salida de la neurona
- δ denota la salida esperada
- α es una constante tal que 0< α <1

Por tanto debemos repetir tantas veces como sea necesario y con cada muestra de aprendizaje la Ecuación 1, donde w_i' es el nuevo peso w_i. Hemos de tener un criterio de parada en el caso de que las muestras no sean linealmente separables, como puede ser un número prefijado de iteraciones.

$$w_i' = w_i + \alpha \ (\delta - y) * x(j)$$

Ecuación 1: Ajuste de pesos en el perceptrón

Un perceptrón de dos capas consiste en una combinación de funciones discriminantes lineales junto con su función de activación agrupadas en 2 capas (capa oculta y capa de salida) más una capa de entrada, la función de activación en el perceptrón multicapa pasa a ser llamada función de transferencia dado que ese dato se transfiere a todas las neuronas de la siguiente capa. En la Figura 6 podemos observar un perceptrón multicapa con una capa oculta de tamaño m, una capa de salida de tamaño 1 y una capa de entrada de tamaño n. A este perceptrón se le llama perceptrón de una capa oculta.

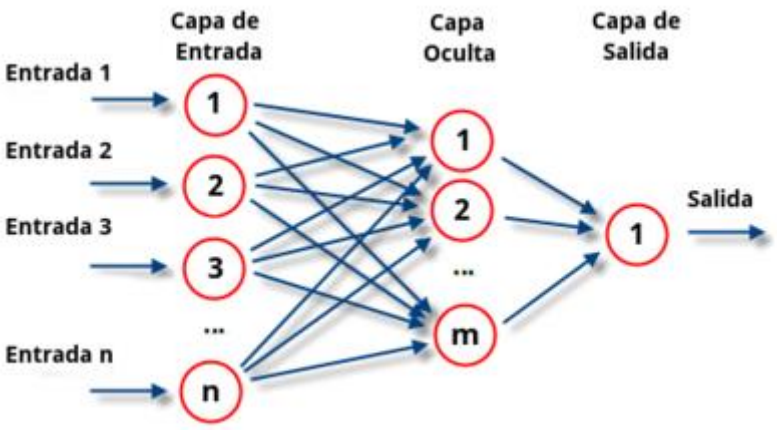

Figura 6: Perceptrón multicapa

Forward

Para nuestra implementación de la red neuronal partiremos del perceptrón multicapa descrito anteriormente, pero en este caso cada capa tendrá su propia función de transferencia, que puede ser la misma que de otras capas, y su propio tamaño. Dado que entrenaremos con la base de datos MNIST la capa de entrada tendrá un tamaño de 784 (las imágenes son de 28x28 pixeles) y una capa de salida de tamaño 10 (reconocerá dígitos del 0 al 9).

Para cada neurona dentro de las capas ocultas tendremos n pesos w donde n es el tamaño de entrada de esa capa (el número de neuronas de la capa anterior). Por tanto, cada neurona de la capa m recibirá el dato que devuelve cada neurona de la capa m-1, lo multiplicará por el peso correspondiente, sumará el resultado de esta operación con todos los datos recibidos, aplicará la función de transferencia y enviará el resultado a todas las neuronas de la capa m+1. Este es el algoritmo *forward*, es el encargado de realizar las operaciones necesarias para clasificar un conjunto de datos de entrada.

Como podemos observar las operaciones de cada neurona son un producto escalar (el sumatorio para $i \in [0$, tamaño de la capa m-1] del resultado de la neurona i de la capa m-1 por w_i de la capa m) y aplicar la función de transferencia al resultado. Si realizamos las operaciones de una capa a la vez agrupando los pesos w en forma de matriz tendremos que ese producto escalar se convierte en un producto matricial, esto no cambia nada en la ejecución en secuencial dado que solo se puede calcular un dato por instrucción, en cambio cuando lo paralelizamos es aquí donde basaremos nuestra optimización para reducir el tiempo de ejecución. Con esta modificación y al tener la misma función de transferencia en toda la capa podemos ejecutarla en paralelo con el resultado de la operación matricial.

La operación matricial nos devolverá un vector de tamaño n_m, donde n_m es el tamaño de la capa m. Para poder optimizar aún más las operaciones en paralelo no solo procesaremos una muestra, sino que lo haremos con varias a la vez. Esta técnica se conoce en el ámbito del aprendizaje automático y se dice que en vez de clasificar una muestra se clasifica un *batch*, es decir, un conjunto de muestras de un tamaño prefijado

e invariante durante todo el entrenamiento. Por tanto si utilizamos esta técnica en vez de recibir un vector al realizar el *forward* de una capa recibiremos una matriz de tamaño n_m x k, donde n_m es el tamaño de la capa m y k es el tamaño del *batch*. Así pues reduciremos aún más el tiempo de ejecución en paralelo.

backPropagation

Este término proviene de la contracción en inglés de *backward propagation of errors*, es decir, propagación hacia atrás de los errores. Esta técnica se basa en ir adaptando los pesos de cada neurona de la red según esta clasifique una serie de muestras de las que ya conocemos la salida que tenemos que recibir, por tanto debemos tener un conjunto de muestras previamente etiquetado de forma correcta.

El objetivo es encontrar los pesos adecuados para que tras realizar el algoritmo *forward* para todas las capas, el valor resultado coincida con las etiquetas de nuestras muestras de entrenamiento, esto en la mayoría de los casos es imposible dado que las muestras han de ser claramente separables y esto depende directamente de la topología de la red. Para solucionar esto se requieren métodos de optimización, el más utilizado en este algoritmo es el de descenso por gradiente.

La técnica de descenso por gradiente se resume en la Ecuación 2, done `ρk` es un valor llamado factor de aprendizaje (con valor mayor que `0` y menor o igual que 1) y `∇q(θ(k))` es el gradiente de la función `q` en el punto `θ(k)`, vector formado por las derivadas parciales de la función calculadas en `θ(k)`. Se basa en un proceso iterativo hasta alcanzar un mínimo local de la función.

```
θ(1) = aleatorio
θ(k + 1) = θ(k) – ρk * ∇q(θ(k))
```

Ecuación 2: Descenso por gradiente

Para calcular el error de clasificación debemos restar la salida que obtenemos de la última capa tras realizar el *forward*, en nuestro caso como tenemos 10 neuronas en la última capa recibiremos un vector de 10 elementos (si el tamaño de *batch* es 1) y como salida esperada será un vector a 0 donde el elemento i sea un 1, siendo i la clase a la que queremos clasificar esos datos, por ejemplo si queremos clasificarlo en la clase 3, la salida esperada será [0,0,0,1,0,0,0,0,0,0].

Podemos ver el algoritmo descrito en la Ecuación 3, una vez obtenido el error de la capa de salida, éste se ha de propagar hacia las capas inferiores, para ello calculamos δ multiplicando cada elemento i,j del error de la capa anterior por el elemento i,j del resultado de aplicar la derivada de la función de transferencia al resultado de aplicar el algoritmo *forward* en la capa actual. Después calculamos el error de la capa actual realizando el producto matricial entre los pesos de esa capa (haciendo su matriz transpuesta) y δ. A continuación calculamos ΔW multiplicando el factor de aprendizaje de la capa por el resultado del producto matricial entre δ y el resultado de aplicar el algoritmo *forward* en la capa actual. Por último solo tenemos que actualizar los pesos restándole a la matriz de pesos de cada capa su ΔW correspondiente.

```
ErrorCapaDeSalida = resultadoForwardUltimaCapa – resultadoEsperado

δ = prodElementoPorElemento( ErrorCapaAnterior ,
derivadaFuncionTransferencia(resultadoForwardCapaActual) )

ErrorCapaActual = prodMatricial( (pesosCapaActual).traspuesta(), δ )

ΔW = factorAprendizaje * prodMatricial(δ, resultadoForwardCapaActual)
```

Ecuación 3: Algoritmo backPropagation

Este algoritmo se ha de repetir el número necesario de veces hasta que la red se pueda considerar entrenada, esto es uno de los puntos más difíciles. Existen diferentes métodos para elegir el criterio de parada, en nuestro caso utilizaremos un número prefijado de iteraciones para que la comparación entre CPU y GPU sea más clara y sencilla. Otro criterio de parada sería por ejemplo que el error de clasificación de las muestras de entrenamiento entre una iteración y la siguiente fuese menor que un número fijado.

Variante con dropout

El coeficiente de *dropout* lo podemos traducir como coeficiente de apagado, es una técnica desarrollada por investigadores de la Universidad de Toronto (5), muy sencilla que se basa en realizar el entrenamiento con menos neuronas en las capas ocultas de las que tiene la red, por eso lo llamamos coeficiente de apagado, dado que desactiva ciertas neuronas. Este coeficiente puede ser diferente para cada capa y solo se aplica en las capas ocultas, dado que si lo aplicamos a la capa de entrada estaríamos eliminando características de la muestra y si lo hacemos en la capa de salida eliminaríamos clases a las que puede pertenecer.

Como podemos ver en la Figura 7, se desactivan ciertas neuronas que serán diferentes en cada *batch* de entrenamiento, las neuronas se seleccionan de forma aleatoria, dado un coeficiente entre 0 y 1 escogeremos las neuronas de manera que se acerque al tamaño de capa * *dropout*. Una vez realizado el entrenamiento, al realizar la clasificación debemos activar todas las neuronas y para compensar el tamaño de los pesos, al ser entrenados con menos neuronas su peso es mayor que si lo hubiésemos hecho con todas, debemos corregirlo multiplicando la salida de cada neurona de una capa por el *dropout* de ésta.

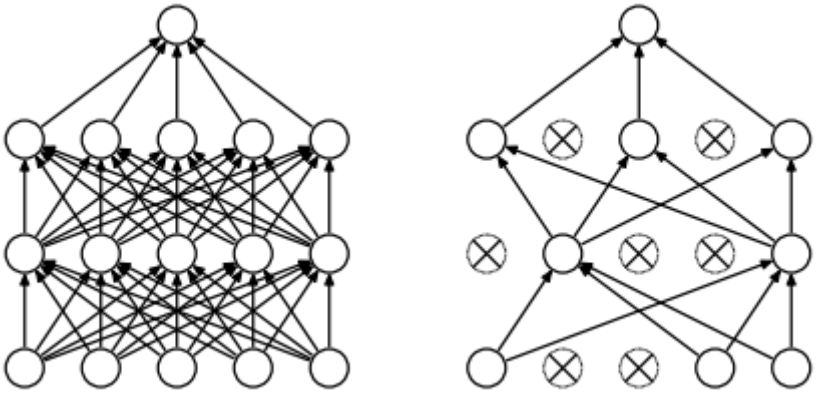

Figura 7: Red neuronal con neuronas activadas y con neuronas desactivadas

Esta técnica se utiliza para evitar que una función realice *overfitting*, es decir, que una función se adapte demasiado a las muestras de aprendizaje lo que puede dar lugar a errores de clasificación. Como podemos ver en la Figura 8, podemos distinguir tres gráficas donde se busca el ajuste de una función discriminante a una serie de datos de entrenamiento, en la primera por la izquierda podemos ver que la función apenas se ajusta a los datos ocasionando muchos errores de clasificación. En la imagen central podemos ver una función que se ajusta bastante bien a los datos, tiene algunos errores pero pueden ser admisibles ya que las muestras pueden haber variado debido al ruido u otros factores. A la derecha podemos ver una función que tiene un sobreajuste con respecto a los datos, podemos observar que no tiene errores en la clasificación, pero esto no es del todo aceptable debido a que la función se ajusta demasiado a los datos de entrenamiento con lo que puede aumentar el número de errores en la clasificación si no hemos seleccionado unos datos de entrenamiento adecuados.

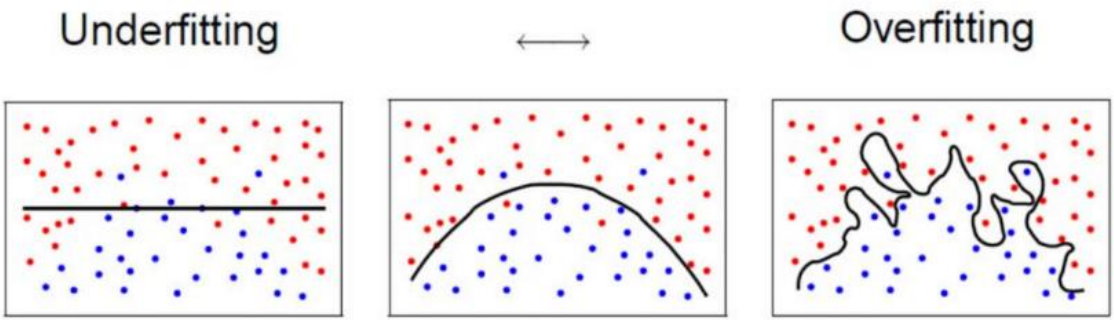

Figura 8: Ajuste de funciones discriminantes

La aplicación del *dropout* se utiliza durante el entrenamiento tanto en el algoritmo *forward* como en el *backPropagation*, esto conlleva un cierto tiempo extra para el entrenamiento y clasificación pero considerablemente inferior a otras técnicas como la *max-norm* o las regularizaciones l1 y l2, de las cuales no vamos a entrar en detalle. Dado que utilizar el *dropout* es beneficioso para la red, las mediciones de tiempos se realizarán aplicando esta variante.

5. Desarrollo del proyecto

Para el desarrollo del proyecto necesitamos definir una estructura de clases para representar la red neuronal como las matrices que vamos a utilizar. Para ello las definiremos en C++ utilizando Visual Studio 2013 las enlazaremos unas con otras para su compilación y posterior ejecución.

Organización del código

Para realizar la implementación de los algoritmos citados anteriormente hemos decidido diseñar una jerarquía de clases, Figura 9, en la que tenemos representada la siguiente estructura implementada mayoritariamente en C++ (archivos que terminan en .h y .cpp):

- Matrix.h: La clase que define la estructura que vamos a utilizar para la representación de matrices.
- Functions.h: La declaración de las operaciones matriciales que vamos a utilizar para ser ejecutadas en la CPU.
- Functions.cpp: Implementación de las funciones descritas en Functions.h.
- Kernels.cuh: El código correspondiente a las funciones implementadas en la clase descrita anteriormente modificadas para ser ejecutadas en la GPU.
- Functional.h: Declaración de un objeto que representará a una capa en la red neuronal.
- Functional.cpp: Implementación de la representación de una capa, ya sea de entrada, oculta o de salida.
- NN.h: Declaración de la estructura que representa íntegramente a la red neuronal.
- NN.cpp: Implementación de la clase descrita anteriormente, hace uso de la clase Functional para representar cada capa en la red.
- MNISTParser.h:El código que se encarga de extraer y organizar la información del fichero de muestras, ya sean de aprendizaje o de test.
- Main.cu: La clase que se encargará de unir todas en una, aquí tenemos la implementación de los algoritmos, tanto para CUDA como para C++, y el método *main* que se encargará de entrenar y clasificar con los datos que le indiquemos.

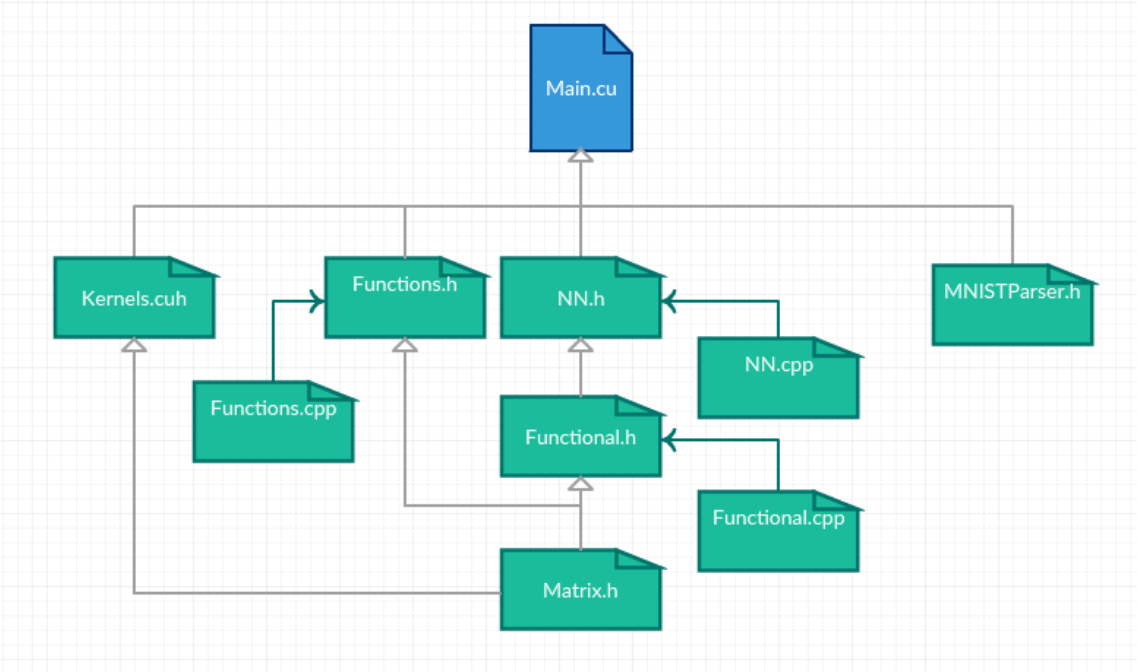

Figura 9: Diseño de clases

Muestras para el entrenamiento y test

Para realizar el aprendizaje de nuestra red necesitamos un conjunto de muestras que podamos separar para realizar el entrenamiento y el test. Para ello hemos optado por el conjunto de datos de MNIST (1) creada por Yann LeCun, Corinna Cortes y Christopher J.C. Burges. Es un conjunto de dígitos escritos a mano, las muestras son imágenes de 28x28 píxeles en escala de grises. Esta base de datos es una ampliación de otra anterior llamada NIST de los que se han conservado 10000 muestras.

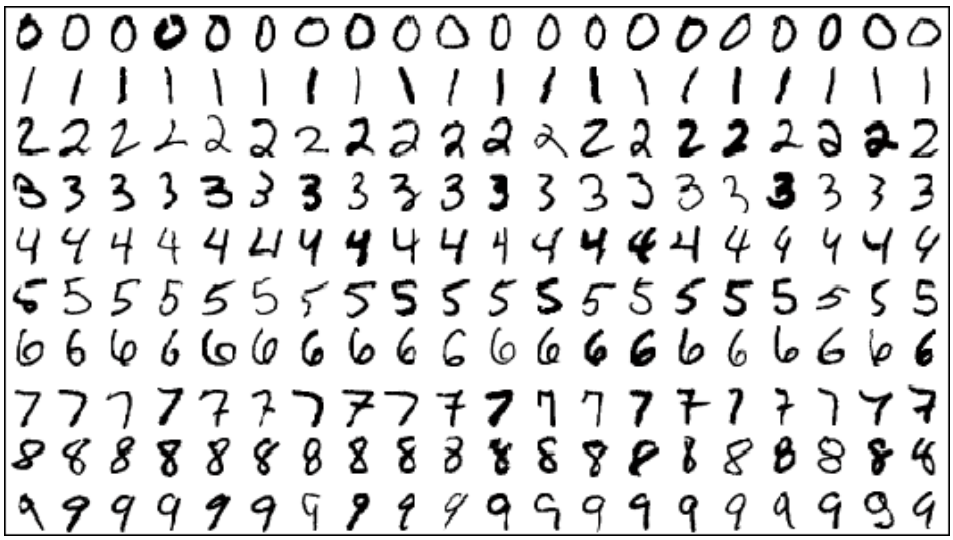

Figura 10: Ejemplo de datos de MNIST

Una tarea difícil en la selección de las muestras es el porcentaje que vamos a utilizar para entrenamiento y cual para test, los creadores de esta base de datos decidieron que 60000 muestras serían para entrenamiento y 10000 para test, con lo que podemos descargar 4 archivos en la página web, las muestras y las etiquetas para el entrenamiento y las muestras y las etiquetas para el test. Los archivos tienen una codificación especial para su fácil procesado, para ello se utiliza el archivo que hemos creado con el nombre de MNISTParser.h.

6. Pruebas y mediciones

Este proyecto se centra en este apartado, donde realmente mediremos si hemos conseguido lo que estábamos buscando, reducir el tiempo de entrenamiento y clasificación de una red neuronal independientemente de la estructura de la misma. Para ello debemos realizar distintas ejecuciones variando la configuración de la red neuronal: En CUDA dependemos del tamaño de bloque, por ello debemos hacer una medición previa para elegir el tamaño que mejor comportamiento tenga, además dependemos del traspaso de datos entre la memoria de la CPU y la memoria de video.

Estudio referente a la GPU

En la Figura 11 podemos ver la jerarquía de memoria para CUDA, donde cada hilo contiene su propia memoria, cada bloque contiene una memoria que comparten todos los hilos de ese bloque y todos los bloques tienen acceso a la memoria de video. Cuanto más compartida es una memoria más lento es el acceso a datos dentro de esta. Utilizando NSIGHT (6), una herramienta diseñada por NVIDIA para depurar (*debugging*) y realizar análisis sobre el rendimiento del código (*profiling*), podemos conocer las características de nuestra tarjeta gráfica, en nuestro caso la GT 630m.

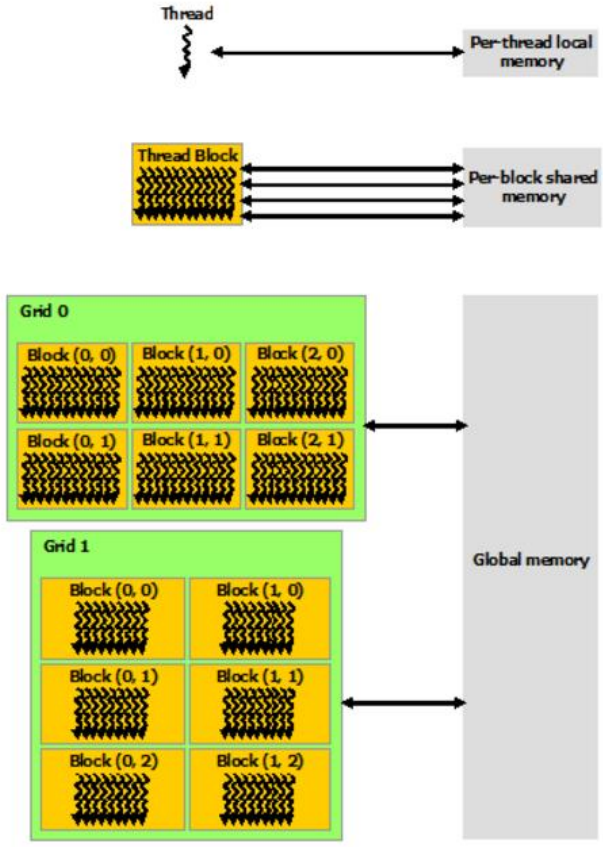

Figura 11: Jerarquía de memoria en CUDA

En el anexo 1 (0) podemos encontrar las características que nos devuelve NSIGHT sobre la tarjeta, pese a ser una gran cantidad de valores hemos de resaltar algunos como los siguientes, dado que nos limitarán a la hora de ejecutar nuestro programa:

- MAX_THREADS_PER_BLOCK: Esto nos indica el número máximo de hilos que puede contener un bloque (eje X * eje Y * eje Z), en este caso 1024. Como trabajamos en 2 dimensiones, el tamaño de bloque ha de ser menor de 32.
- TOTAL_MEMORY: Memoria de video total que tenemos, en nuestro caso 2GB, 2147483648 Bytes.
- MAX_GRID_DIM_X: El número máximo de bloques que podemos ejecutar, en este caso se refiere al eje X pero es un número global, con lo que está condicionado a utilizar solo ese eje. El máximo es 65535, si utilizásemos los 3 ejes, el tamaño máximo para cada eje sería 40 ($\sqrt[3]{65535} = 40.3173$). Como utilizamos solo dos dimensiones, el tamaño máximo será 255 ($\sqrt[2]{65535} = 255.9980$).
- MAX_SHARED_MEMORY_PER_BLOCK : Tamaño de la memoria compartida para cada bloque, 49152 Bytes, 48 MB.

El resto de parámetros están más dirigidos a la programación gráfica, sobre todo para el cálculo de texturas sobre superficies, por lo que no vamos si quiera a nombrarlos.

Dado la gran cantidad de accesos a memoria que realiza la multiplicación de matrices, NVIDIA propone en su guía de programación (7) utilizar la memoria compartida para reducir el acceso a memoria de video, para ello tenemos que modificar el algoritmo que multiplica como en la Figura 12, para que todos los hilos de un mismo bloque utilicen la memoria compartida para guardar y consultar los elementos que necesitan de la matriz.

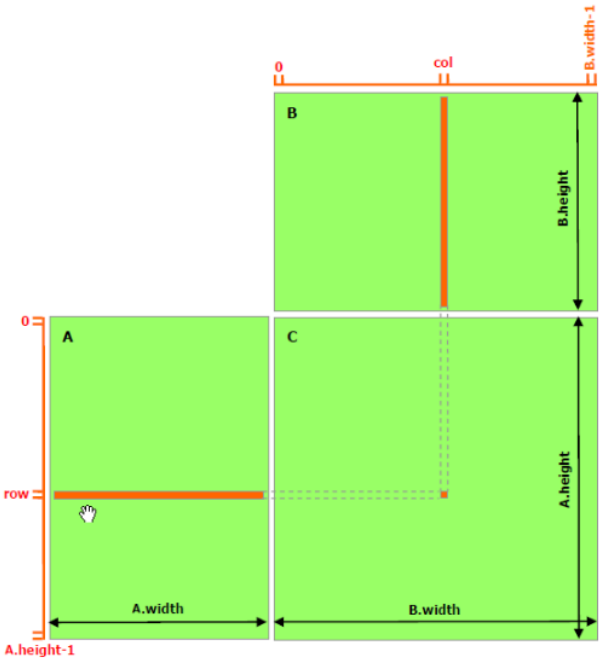

Figura 12: Multiplicación de matrices

Para realizar la multiplicación utilizando memoria compartida usaremos el código del anexo 2 (0), crearemos dos matrices compartidas (As y Bs) de tamaño BLOCK_SIZExBLOCK_SIZE, iremos recorriendo de izquierda a derecha la matriz, desde la fila que nos indique el bloque en el que nos encontramos. Cada hilo del bloque guardará el elemento que le corresponda de A en As y de B en Bs de la submatriz que obtenemos desde el elemento que se encuentra en [blockIdx.y*BLOCK_SIZE, blockIdx.x*BLOCK_SIZE] hasta el elemento en [blockIdx.y*BLOCK_SIZE + BLOCK_SIZE, blockIdx.x*BLOCK_SIZE + BLOCK_SIZE] una vez guardado cada elemento, los hilos se sincronizan para evitar condiciones de carrera y calculan el elemento de la multiplicación matricial entre las submatrices As y Bs que les correspondería. Repiten este proceso para cada bloque que encuentran en horizontal y suman el resultado de cada uno para obtener el dato calculado de la matriz C en el punto [blockIdx.y*BLOCK_SIZE + threadIdx.y, blockIdx.x*BLOCK_SIZE + threadIdx.x].

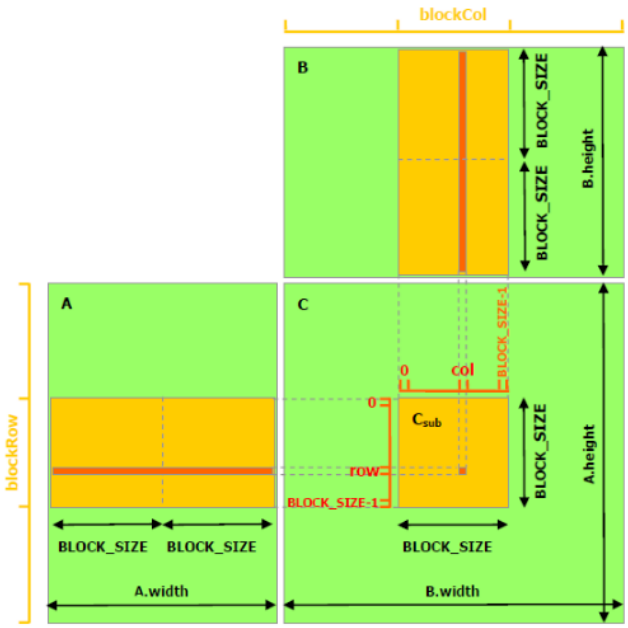

Figura 13: Multiplicación de matrices utilizando memoria compartida

Para comprobar el cambio que se produce en el tiempo de ejecución de la multiplicación utilizando memoria compartida frente a no utilizarla, código del anexo 3 (0), para ello realizaremos una serie de ejecuciones variando el tamaño de las matrices que se van a multiplicar, además realizaremos una comparativa en base al tamaño de bloque. Se han realizado diversas pruebas de las cuales solo se muestran una pequeña parte, en el anexo 4 (0) podemos encontrar más datos sobre otras pruebas realizadas que no se han incluido en este apartado.

Una vez realizadas las pruebas hemos obtenidos los resultados expuestos en la Tabla 11 donde podemos ver el tiempo de cada ejecución en memoria global frente al tiempo utilizando memoria compartida, se han realizado pruebas variando tanto el tamaño de las matrices como el tamaño de bloque, este último se ha visto limitado a un tamaño máximo de bloque de 29 dada la implementación de la multiplicación en memoria compartida ya que un tamaño mayor de bloque consumiría más memoria de la que disponemos para cada bloque en esta GPU.

BLOCK-SIZE	tamaño de las matrices	memoria global (ms)	memoria compartida (ms)	speedup
8x8	300x300 * 300x300	113,099	112,617	1,00
8x8	300x600 * 600x1200	252,462	271,905	0,93
8x8	600x600 * 600x600	250,645	272,152	0,92
8x8	600x900 * 900x1200	752,885	811,152	0,93
8x8	900x900 * 900x900	852,653	926,324	0,92
8x8	1200x1200 * 1200x1200	2001,570	2155,210	0,93
16x16	300x300 * 300x300	92,889	72,076	1,29
16x16	300x600 * 600x1200	220,958	156,752	1,41
16x16	600x600 * 600x600	226,487	159,040	1,42
16x16	600x900 * 900x1200	735,303	461,867	1,59
16x16	900x900 * 900x900	825,659	529,087	1,56
16x16	1200x1200 * 1200x1200	2174,540	1198,420	1,81
24x24	300x300 * 300x300	31,905	27,036	1,18
24x24	300x600 * 600x1200	206,843	161,691	1,28
24x24	600x600 * 600x600	201,365	157,392	1,28
24x24	600x900 * 900x1200	603,005	471,098	1,28
24x24	900x900 * 900x900	685,156	541,814	1,26
24x24	1200x1200 * 1200x1200	1586,910	1225,810	1,29
29x29	300x300 * 300x300	39,626	26,249	1,51
29x29	300x600 * 600x1200	252,585	155,504	1,62
29x29	600x600 * 600x600	244,028	149,272	1,63
29x29	600x900 * 900x1200	746,945	446,416	1,67
29x29	900x900 * 900x900	846,206	515,178	1,64
29x29	1200x1200 * 1200x1200	1997,350	1160,660	1,72

Tabla 11: tiempos diferentes multiplicaciones en GPU

En la última columna podemos observar lo que en computación paralela se conoce como *speedup*, aceleración en castellano, es el ratio (en tanto por uno) que varía el tiempo de una magnitud respecto a otra, en nuestro caso es el tiempo utilizando memoria compartida frente al tiempo utilizando memoria global, dividiendo la segunda por la primera.

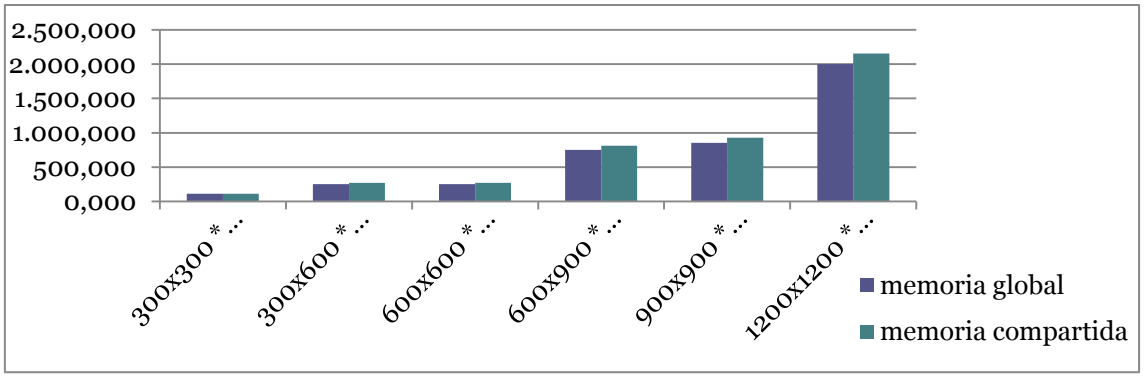

Figura 14: Memoria global vs compartida, tamaño de bloque 8x8

Podemos ver en la Figura 14 y en la columna de *speedup* que el tiempo de ejecución con el tamaño de bloque de 8x8 es algo mejor en la ejecución con memoria global, esto es debido a que al ser tan pequeño el tamaño de bloque las operaciones que se necesitan para guardar los datos en la memoria compartida requieren más tiempo del que ahorramos en accesos a memoria global.

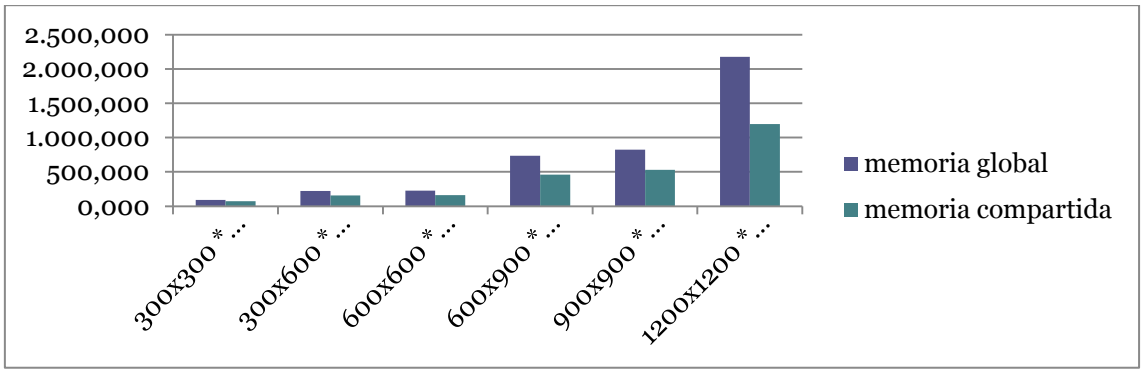

Figura 15: Memoria global vs compartida, tamaño de bloque 16x16

En cambio, en la Figura 15 y en la Figura 16 nos encontramos la situación a la inversa, es aquí donde la utilización de memoria compartida mejora en los tiempos frente a la memoria global, esto es debido a la reducción de accesos a la memoria de video.

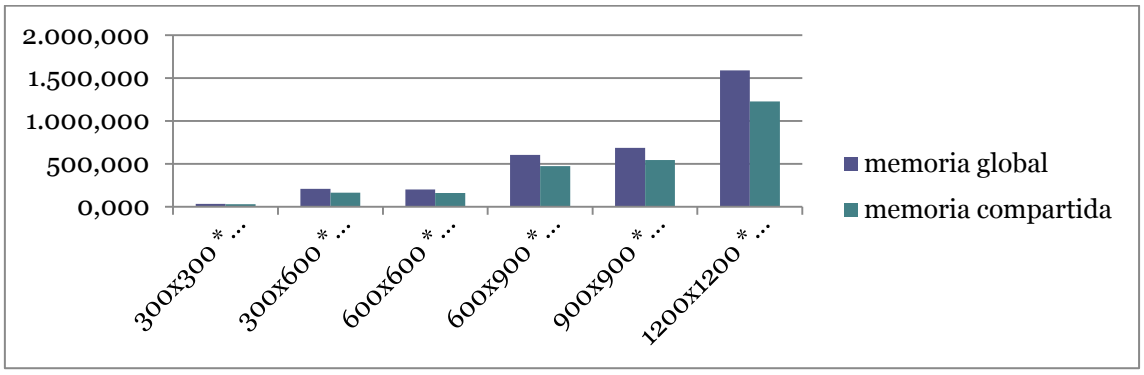

Figura 16: Memoria global vs compartida, tamaño de bloque 24x24

Por último tenemos la Figura 17 donde el tamaño de bloque es 29x29, este será el tamaño de bloque que utilizaremos para el desarrollo de otras futuras pruebas dado que su *speedup* es de media el mayor frente al resto de tamaños de bloque.

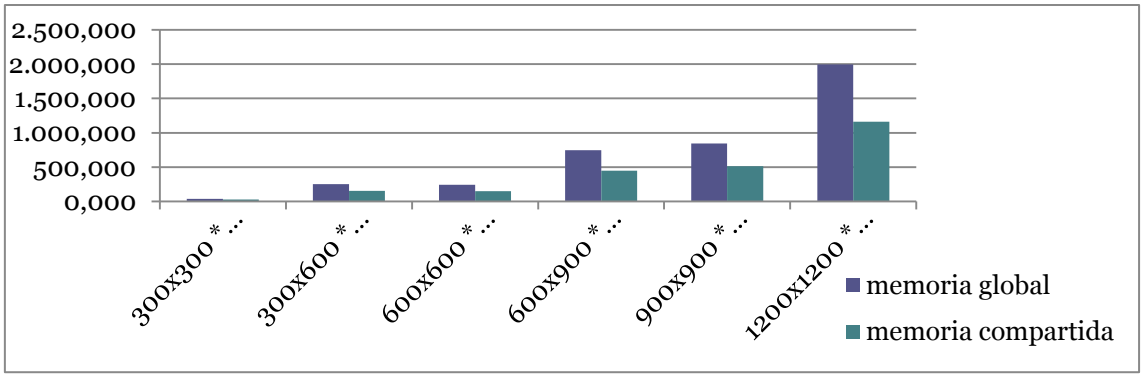

Figura 17: Memoria global vs compartida, tamaño de bloque 29x29

Por último en la Figura 18 tenemos representadas todas las ejecuciones utilizando memoria compartida, coloreadas de forma diferente para cada tamaño de matriz y separadas por el tamaño de bloque en cada ejecución.

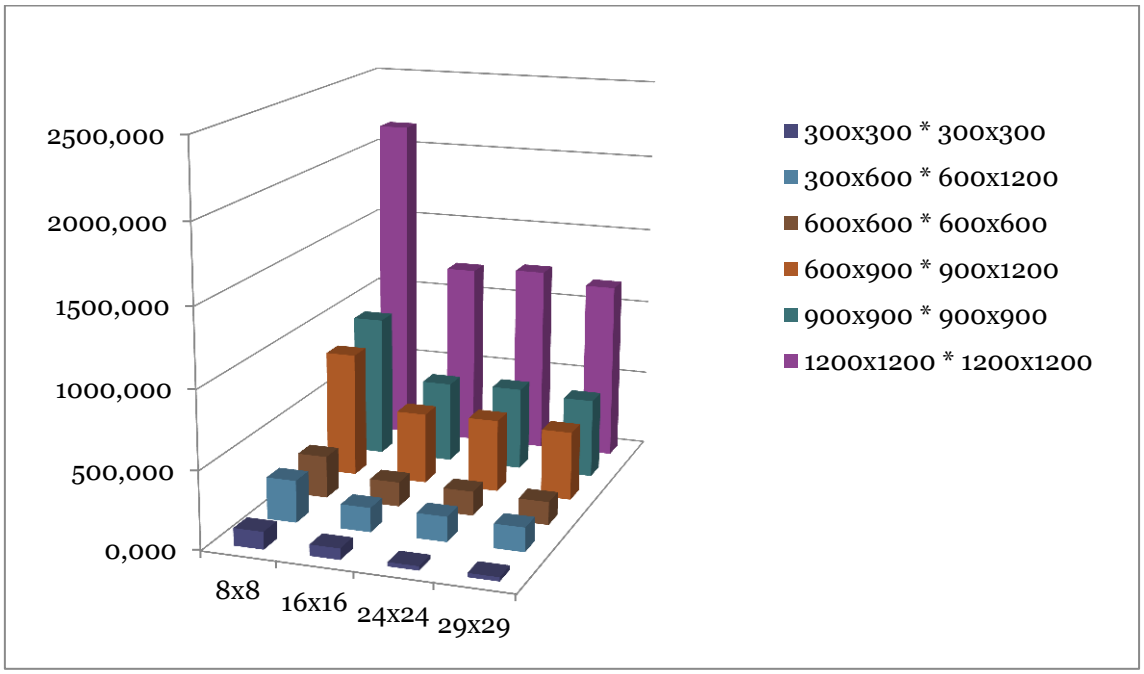

Figura 18: Memoria global vs compartida

Estudio de la GPU frente a la CPU

Una vez realizado el estudio dentro de la GPU podemos proseguir con la comparación entre multiplicaciones, esta vez la multiplicación de matrices de la Tabla 12 realizada en C++ y ejecutada sobre la CPU frente a la versión implementada en CUDA utilizando memoria compartida y un tamaño de bloque de 29x29.

```
void MultipliccionCPU(Matrix A, Matrix B, Matrix C)
{
```

```
      for (int i = 0; i < C.rows; i++) {
            for (int j = 0; j < C.columns; j++) {

                  float value = 0.0;

                  for (int k = 0; k < A.columns; k++) {
                        value  +=  A.elements[i  *  A.stride  +  k]  *
B.elements[k * B.stride + j];
                  }
                  C.elements[i * C.stride + j] = value;
            }
      }
}
```

Tabla 12: Multiplicación de matrices en CPU

En la Tabla 13 tenemos los tiempos resultantes de las ejecuciones con matrices con el mismo tamaño que en el apartado anterior. En el anexo 5 (0) encontraremos más ejecuciones con distinto tamaño de matrices. Dado que los tiempos han sido medidos con dos métodos distintos, los tiempos para la CPU no tienen la misma precisión que los de la GPU, por ello en la CPU el menor tiempo que podemos medir es un milisegundo (ms) y en la GPU una centésima de milisegundo. Pese a ello el estudio no se ve apenas alterado.

tamaño de las matrices	CPU (ms)	GPU (ms)	speedup
300x300 * 300x300	158,00	24,93	6,34
300x600 * 600x1200	1639,00	146,81	11,16
600x600 * 600x600	1379,00	137,99	9,99
600x900 * 900x1200	4755,00	438,63	10,84
900x900 * 900x900	5127,00	500,75	10,24
1200x1200 * 1200x1200	14630,00	1068,63	13,69

Tabla 13: Multiplicación en la CPU vs en la GPU

En la Figura 19 tenemos la representación de los datos, podemos ver que la mejora de la GPU frente a la CPU es considerable. Para la medición de tiempos en la GPU también se ha tenido en cuenta el tiempo necesario para reservar la memoria de video necesaria para realizar la multiplicación y enviar los datos desde la memoria RAM a la de la propia GPU. Hemos de considerar que en una red neuronal se realizan una gran cantidad de multiplicaciones matriciales, por lo tanto un speedup de tal magnitud puede suponer una gran aceleración en el entrenamiento y clasificación de dicha red.

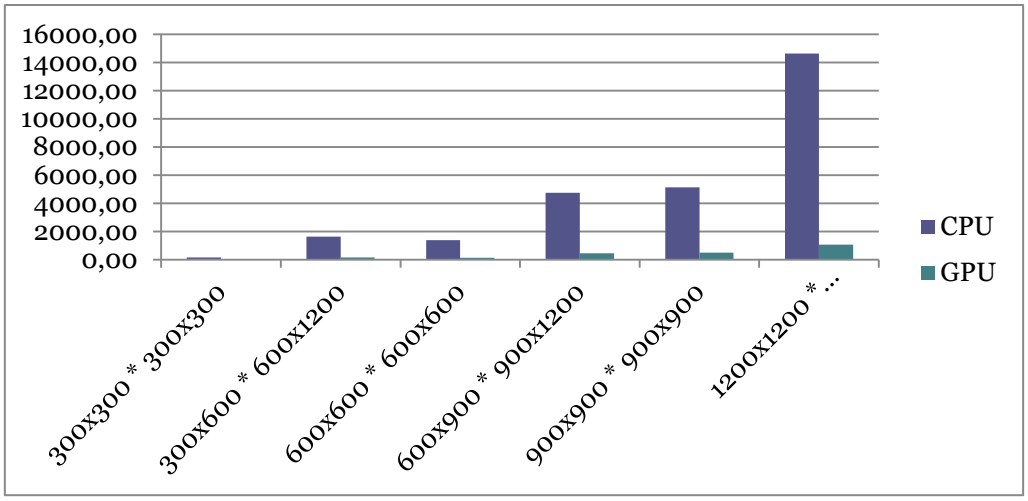

Figura 19: Multiplicación en CPU vs GPU

Estudio sobre las redes neuronales

Una vez hemos realizado la comparativa entre multiplicaciones el siguiente paso que tenemos que dar es hacia los algoritmos implementados. Una red neuronal se basa en dos funciones principales, el entrenamiento y la clasificación. La clasificación de una muestra en una red neuronal se realiza mediante el algoritmo *forward*, anteriormente comentado, en cambio el entrenamiento necesita también hacer uso del *backward* para modificar los pesos en cada neurona. Es por ello que no es igual el tiempo de ejecución, además de que en la parte de entrenamiento hemos de repetir el *forward* y el *backward* una serie de veces hasta considerar la red entrenada para reducir el error de clasificación.

tamaño	batch 100			batch 50		
	1 it.	50 it.	100 it.	1 it.	50 it.	100 it.
3 capas de 1024	6861,92	337460,12	675361,03	16879,20	844125,42	1687953,16
3 capas de 512	4415,81	220843,49	441493,13	4673,82	233737,74	467473,15
3 capas de 256	1754,48	87463,50	175501,20	1697,26	84879,45	169761,44

Tabla 14: Entrenamiento en CPU

En la Tabla 14 tenemos los tiempos de ejecución para 1, 50 y 100 iteraciones en una red neuronal ejecutada en la CPU variando tanto el tamaño de las capas ocultas como el tamaño de *batch*, el tiempo está representado en segundos. Por ejemplo, para realizar una iteración con una red de 3 capas ocultas de tamaño 1024 y un tamaño de *batch* 100, su ejecución ha tardado 1 hora, 54 minutos y 21.92 segundos, para una red neuronal es un tiempo bastante lento, pero hemos de tener en cuenta que la red neuronal y los algoritmos que ejecutan han sido creados desde cero, sin importar librería alguna ni realizar optimizaciones en los cálculos. La decisión de no utilizar ninguna librería externa ha sido para asegurarnos que la comparativa se realizaba con el mayor grado de similitud posible.

tamaño	batch 100			batch 50		
	1 it.	50 it.	100 it.	1 it.	50 it.	100 it.
3 capas de 1024	410,72	20539,96	41072,90	477,97	23898,92	47558,61
3 capas de 512	144,23	7212,75	14483,10	175,58	8778,90	17559,77
3 capas de 256	59,11	2956,42	5904,50	72,57	3524,16	7258,30

Tabla 15: Entrenamiento en GPU

En la Tabla 15 encontramos los tiempos de las ejecuciones correspondientes realizadas en la GPU todas ellas con un tamaño de bloque de 29, que como comentamos en el capítulo anterior es el que mejor resultados obtenía en la comparativa, los tiempos también se muestran en segundos. Para realizar 1 iteración con la misma configuración que hemos propuesto antes (3 capas de 1024 y tamaño de *batch* 100) en la GPU necesitamos 6 minutos y 50.72 segundos, alrededor de 1 hora y 48 minutos menos en una sola iteración.

tamaño	batch 100			batch 50		
	1 it.	50 it.	100 it.	1 it.	50 it.	100 it.
3 capas de 1024	16,71	16,43	16,44	35,31	35,32	35,49
3 capas de 512	30,62	30,62	30,48	26,62	26,62	26,62
3 capas de 256	29,68	29,58	29,72	23,39	24,09	23,39

Tabla 16: speedup GPU vs CPU

En la Tabla 16 podemos ver los *speedups* de las diferentes ejecuciones, dado que el sistema para paralelizar las operaciones en la GPU depende directamente del tamaño de las operaciones que se van a realizar, los *speedup* son diferentes para cada tamaño de capa oculta y para cada tamaño de *batch*, es por ello por lo que debemos variar las diferentes configuraciones para poder comparar el comportamiento.

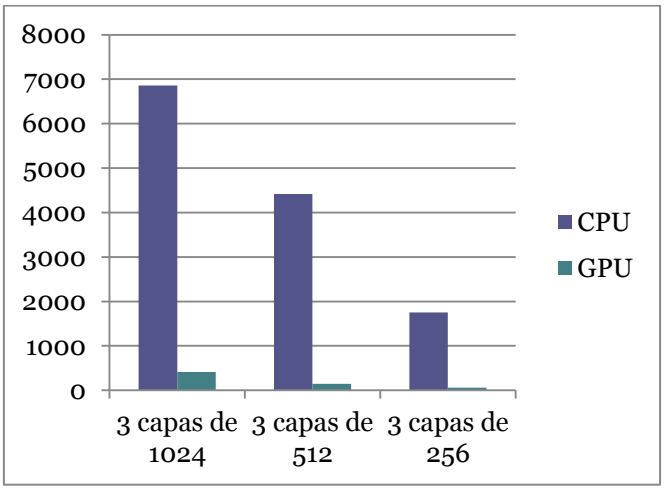

Figura 20: GPU vs CPU, 1 iteración, tamaño de batch 100

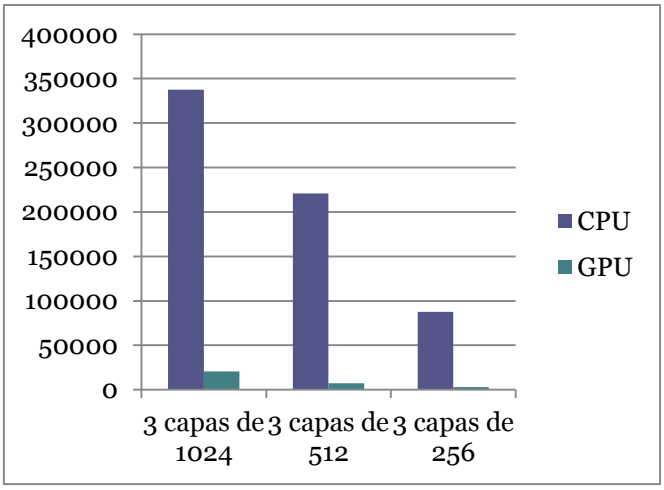

Figura 21: GPU vs CPU, 50 iteraciones, tamaño de batch 100

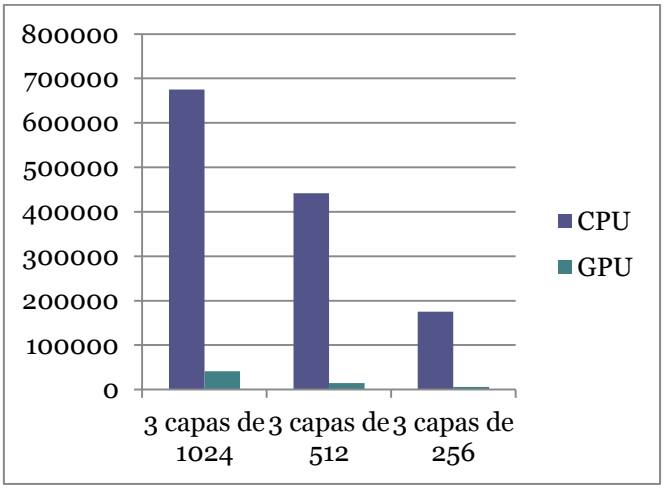

Figura 22: GPU vs CPU, 100 iteraciones, tamaño de batch 100

En las figuras anteriores podemos ver la representación del tiempo que tarda la GPU y la CPU en realizar 1, 50 y 100 iteraciones para cada uno de los tamaños de capa oculta propuestos usando un tamaño de *batch* de 100. Si nos fijamos en la Tabla 16 podemos ver que los *speedup* para las tres capas de 512 y las tres capas de 256 son mucho mayores que para las tres de 1024, esto es difícilmente visible en las figuras dada la gran magnitud del tiempo que hemos medido en la CPU y lo pequeño que es en comparación el de la GPU.

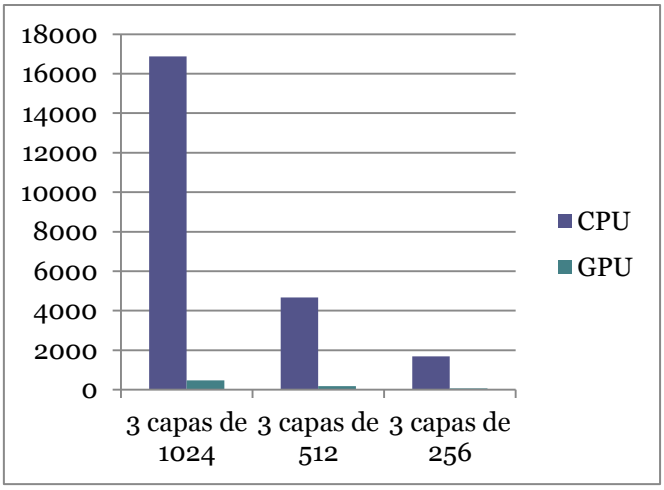

Figura 23: GPU vs CPU, 1 iteración, tamaño de batch 50

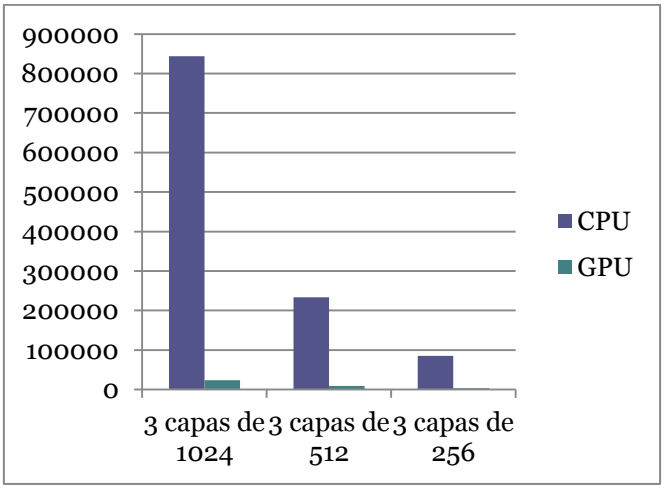

Figura 24: GPU vs CPU, 50 iteraciones, tamaño de batch 50

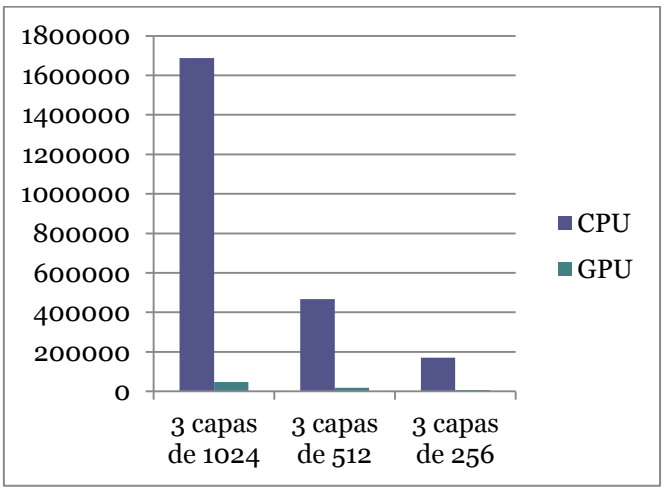

Figura 25: GPU vs CPU, 100 iteraciones, tamaño de batch 50

En las figuras 23, 24 y 25 podemos apreciar la misma comparativa, esta vez para un tamaño de *batch* de 50, en este caso la mejora de la GPU contra la CPU es mayor aún, por lo que casi no podemos apreciar las columnas relativas al tiempo de la GPU. Pese a que el *speedup* con un tamaño de *batch* 50 sea mayor que con 100, el tiempo de

entrenamiento con un *batch* de 50 es mayor que con uno de 100, esto es debido a que aunque se reducen el tamaño de las operaciones, se duplican la cantidad que hay que realizar de estas.

7. Conclusiones

Al inicio del proyecto nos planteamos dos objetivos:

- La traducción de código de algoritmos de aprendizaje automático para ser ejecutados tanto en la CPU como en la GPU
- La cuantificación de la mejora en el tiempo de ejecución de dichos algoritmos tras su cómputo en ambas unidades de procesamiento.

Una vez realizados ambos podemos concluir que la programación de una red neuronal para su ejecución en la GPU es parcialmente posible, es decir, hemos desarrollado las funciones matemáticas que realiza una red, pero la estructura y el control de su ejecución se ha de realizar en la CPU, además de la lectura de muestras y etiquetas para su entrenamiento y clasificación.

En el objetivo de la cuantificación de la mejora del tiempo de ejecución podemos ver en el apartado anterior, donde comparamos los tiempos de ejecución de la red neuronal, que podemos llegar a conseguir tiempos mucho mejores en la GPU, llegando a ser hasta 35 veces inferior que el de la CPU. Una de las preguntas que nos puede surgir es como puede llegar a ser esto posible si en el apartado de la comparación entre multiplicaciones el mayor *speedup* es cercano a 13. Esto es debido a que la red neuronal no solo utiliza multiplicaciones, además de que en ese apartado tuvimos en cuenta el coste de transferir los objetos de memoria RAM a memoria de video y viceversa, además el modelo de ejecución de CUDA permite que la CPU llame a un *kernel* y continúe la ejecución del programa hasta llegar a un punto donde necesite los datos calculados en dicho *kernel*, donde los esperará si este no ha terminado sus cálculos.

8. Bibliografía

1. **Yann LeCun, Corinna Cortes, Christopher J.C. Burges.** THE MNIST DATABASE. [En línea] Yann LeCun. http://yann.lecun.com/exdb/mnist/.

2. CUDA Parallel Computing Platform. [En línea] NVIDIA. http://www.nvidia.com/object/cuda_home_new.html.

3. CUDA NVIDIA downloads. [En línea] NVIDIA. https://developer.nvidia.com/cuda-downloads.

4. Techpowerup NVIDIA geforce gt 630m. [En línea] Techpowerup. https://www.techpowerup.com/gpudb/353/geforce-gt-630m.

5. **Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov.** Department of Computer Science University of Toronto. [En línea] 6 de 2014. https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf.

6. NVIDIA Nsight Visual Studio Edition. [En línea] NVIDIA. https://developer.nvidia.com/nvidia-nsight-visual-studio-edition.

7. Cuda C programming guide. *Guia de programación v7.5*. [En línea] NVIDIA, 1 de Septiembre de 2015. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

9. Anexos

Características de la tarjeta gráfica GT 630m

GeForce GT 630M	
MAX_THREADS_PER_BLOCK	1024
MAX_BLOCK_DIM_X	1024
MAX_BLOCK_DIM_Y	1024
MAX_BLOCK_DIM_Z	64
MAX_GRID_DIM_X	65535
MAX_GRID_DIM_Y	65535
MAX_GRID_DIM_Z	65535
MAX_SHARED_MEMORY_PER_BLOCK	49152
TOTAL_CONSTANT_MEMORY	65536
WARP_SIZE	32
MAX_PITCH	2147483647
MAX_REGISTERS_PER_BLOCK	32768
CLOCK_RATE	950000
TEXTURE_ALIGNMENT	512
GPU_OVERLAP	1
MULTIPROCESSOR_COUNT	2
KERNEL_EXEC_TIMEOUT	1
INTEGRATED	0
CAN_MAP_HOST_MEMORY	1
COMPUTE_MODE	0

MAXIMUM_TEXTURE1D_WIDTH	65536
MAXIMUM_TEXTURE2D_WIDTH	65536
MAXIMUM_TEXTURE2D_HEIGHT	65535
MAXIMUM_TEXTURE3D_WIDTH	2048
MAXIMUM_TEXTURE3D_HEIGHT	2048
MAXIMUM_TEXTURE3D_DEPTH	2048
MAXIMUM_TEXTURE2D_LAYERED_WIDTH	16384
MAXIMUM_TEXTURE2D_LAYERED_HEIGHT	16384
MAXIMUM_TEXTURE2D_LAYERED_LAYERS	2048
SURFACE_ALIGNMENT	512
CONCURRENT_KERNELS	1
ECC_ENABLED	0
PCI_BUS_ID	1
PCI_DEVICE_ID	0
TCC_DRIVER	0
MEMORY_CLOCK_RATE	900000
GLOBAL_MEMORY_BUS_WIDTH	128
L2_CACHE_SIZE	131072
MAX_THREADS_PER_MULTIPROCESSOR	1536
ASYNC_ENGINE_COUNT	1
UNIFIED_ADDRESSING	0

MAXIMUM_TEXTURE1D_LAYERED_WIDTH	16384
MAXIMUM_TEXTURE1D_LAYERED_LAYERS	2048
CAN_TEX2D_GATHER	1
MAXIMUM_TEXTURE2D_GATHER_WIDTH	16384
MAXIMUM_TEXTURE2D_GATHER_HEIGHT	16384
MAXIMUM_TEXTURE3D_WIDTH_ALTERNATE	0
MAXIMUM_TEXTURE3D_HEIGHT_ALTERNATE	0
MAXIMUM_TEXTURE3D_DEPTH_ALTERNATE	0
PCI_DOMAIN_ID	0
TEXTURE_PITCH_ALIGNMENT	32
MAXIMUM_TEXTURECUBEMAP_WIDTH	16384
MAXIMUM_TEXTURECUBEMAP_LAYERED_WIDTH	16384
MAXIMUM_TEXTURECUBEMAP_LAYERED_LAYERS	2046
MAXIMUM_SURFACE1D_WIDTH	65536
MAXIMUM_SURFACE2D_WIDTH	65536

MAXIMUM_SURFACE2D_HEIGHT	32768
MAXIMUM_SURFACE3D_WIDTH	65536
MAXIMUM_SURFACE3D_HEIGHT	32768
MAXIMUM_SURFACE3D_DEPTH	2048
MAXIMUM_SURFACE1D_LAYERED_WIDTH	65536
MAXIMUM_SURFACE1D_LAYERED_LAYERS	2048
MAXIMUM_SURFACE2D_LAYERED_WIDTH	65536
MAXIMUM_SURFACE2D_LAYERED_HEIGHT	32768
MAXIMUM_SURFACE2D_LAYERED_LAYERS	2048
MAXIMUM_SURFACECUBEMAP_WIDTH	32768
MAXIMUM_SURFACECUBEMAP_LAYERED_WIDTH	32768
MAXIMUM_SURFACECUBEMAP_LAYERED_LAYERS	2046
MAXIMUM_TEXTURE1D_LINEAR_WIDTH	134217728
MAXIMUM_TEXTURE2D_LINEAR_WIDTH	65000
MAXIMUM_TEXTURE2D_LINEAR_HEIGHT	65000

MAXIMUM_TEXTURE2D_LINEAR_PITCH	1048544
MAXIMUM_TEXTURE2D_MIPMAPPED_WIDTH	16384
MAXIMUM_TEXTURE2D_MIPMAPPED_HEIGHT	16384
MAXIMUM_TEXTURE1D_MIPMAPPED_WIDTH	16384
STREAM_PRIORITIES_SUPPORTED	0
GLOBAL_L1_CACHE_SUPPORTED	1
LOCAL_L1_CACHE_SUPPORTED	1
MAX_SHARED_MEMORY_PER_MULTIPROCESSOR	49152
MAX_REGISTERS_PER_MULTIPROCESSOR	32768
MANAGED_MEMORY	0
MULTI_GPU_BOARD	0
MULTI_GPU_BOARD_GROUP_ID	0
DISPLAY_NAME	GeForce GT 630M
COMPUTE_CAPABILITY_MAJOR	2
COMPUTE_CAPABILITY_MINOR	1
TOTAL_MEMORY	2147483648
RAM_TYPE	7
RAM_LOCATION	1

GPU_PCI_DEVICE_ID	233378014
GPU_PCI_SUB_SYSTEM_ID	340201539
GPU_PCI_REVISION_ID	161
GPU_PCI_EXT_DEVICE_ID	3561
GPU_PCI_EXT_GEN	1
GPU_PCI_EXT_GPU_GEN	1
GPU_PCI_EXT_GPU_LINK_RATE	5000
GPU_PCI_EXT_GPU_LINK_WIDTH	16
GPU_PCI_EXT_DOWNSTREAM_LINK_RATE	8000
GPU_PCI_EXT_DOWNSTREAM_LINK_WIDTH	16

Multiplicación en CUDA con memoria compartida

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
      int Row = blockIdx.y*BLOCK_SIZE + threadIdx.y;
      int Col = blockIdx.x*BLOCK_SIZE + threadIdx.x;

      float CValue = 0;

      __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
      __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

      for (int k = 0; k < (BLOCK_SIZE + A.columns - 1) / BLOCK_SIZE;
k++) {

            if (k*BLOCK_SIZE + threadIdx.x < A.columns && Row <
A.rows)
                  As[threadIdx.y][threadIdx.x]                        =
GetElement(A,Row,k*BLOCK_SIZE + threadIdx.x);
            else
                  As[threadIdx.y][threadIdx.x] = 0.0;

            if (k*BLOCK_SIZE + threadIdx.y < B.rows && Col <
B.columns)
```

```
                  Bs[threadIdx.y][threadIdx.x]                                =
GetElement(B,(k*BLOCK_SIZE + threadIdx.y),Col);
            else
                  Bs[threadIdx.y][threadIdx.x] = 0.0;

            __syncthreads();

            for (int n = 0; n < BLOCK_SIZE; ++n)
                  CValue += As[threadIdx.y][n] * Bs[n][threadIdx.x];

            __syncthreads();
      }

      if (Row < C.rows && Col < C.columns)
            SetElement(C,  blockIdx.y  *  BLOCK_SIZE  +  threadIdx.y,
(blockIdx.x*BLOCK_SIZE) + threadIdx.x, CValue);
}
```

Multiplicación de matrices en CUDA

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
      float Cvalue = 0;
      int col = blockIdx.y * BLOCK_SIZE + threadIdx.y;
      int row = blockIdx.x * BLOCK_SIZE + threadIdx.x;

      if (row < C.rows && col < C.columns){
            for (int e = 0; e < A.columns; ++e)
                  Cvalue += GetElement(A, row, e) *  GetElement(B,  e,
col);
            SetElement(C, row, col, Cvalue);
      }
}
```

Comparación de multiplicaciones en la GPU

Time on global_memory= 38,3932 msec, Time on sharedl_memory= 42,3326 msec , dim = 300 x 300 x 300 block_size = 8 elements = 90000 speed 0,906942 errors 0
Time on global_memory= 75,6201 msec, Time on sharedl_memory= 82,6049 msec , dim = 300 x 300 x 600 block_size = 8 elements = 180000 speed 0,915443 errors 0
Time on global_memory= 113,099 msec, Time on sharedl_memory= 112,617 msec , dim = 300 x 300 x 900 block_size = 8 elements = 270000 speed 1,00428 errors 0
Time on global_memory= 132,659 msec, Time on sharedl_memory= 144,455 msec , dim = 300 x 300 x 1200 block_size = 8 elements = 360000 speed 0,918344 errors 0
Time on global_memory= 67,513 msec, Time on sharedl_memory= 71,6852 msec , dim = 300 x 600 x 300 block_size = 8 elements = 90000 speed 0,941797 errors 0
Time on global_memory= 127,181 msec, Time on sharedl_memory= 138,823 msec , dim = 300 x 600 x 600 block_size = 8 elements = 180000 speed 0,916136 errors 0
Time on global_memory= 190,452 msec, Time on sharedl_memory= 206,165 msec , dim = 300 x 600 x 900 block_size = 8 elements = 270000 speed 0,923783 errors 0

Time on global_memory= 252,462 msec, Time on sharedl_memory= 271,905 msec , dim = 300 x 600 x 1200 block_size = 8 elements = 360000 speed 0,928494 errors 0
Time on global_memory= 98,9418 msec, Time on sharedl_memory= 106,748 msec , dim = 300 x 900 x 300 block_size = 8 elements = 90000 speed 0,926868 errors 0
Time on global_memory= 190,926 msec, Time on sharedl_memory= 208,948 msec , dim = 300 x 900 x 600 block_size = 8 elements = 180000 speed 0,91375 errors 0
Time on global_memory= 288,237 msec, Time on sharedl_memory= 313,23 msec , dim = 300 x 900 x 900 block_size = 8 elements = 270000 speed 0,920207 errors 0
Time on global_memory= 378,905 msec, Time on sharedl_memory= 410,943 msec , dim = 300 x 900 x 1200 block_size = 8 elements = 360000 speed 0,922036 errors 0
Time on global_memory= 129,749 msec, Time on sharedl_memory= 142,048 msec , dim = 300 x 1200 x 300 block_size = 8 elements = 90000 speed 0,913415 errors 0
Time on global_memory= 253,385 msec, Time on sharedl_memory= 275,323 msec , dim = 300 x 1200 x 600 block_size = 8 elements = 180000 speed 0,920319 errors 0
Time on global_memory= 379,841 msec, Time on sharedl_memory= 414,461 msec , dim = 300 x 1200 x 900 block_size = 8 elements = 270000 speed 0,916471 errors 0
Time on global_memory= 504,288 msec, Time on sharedl_memory= 549,795 msec , dim = 300 x 1200 x 1200 block_size = 8 elements = 360000 speed 0,917229 errors 0
Time on global_memory= 65,6861 msec, Time on sharedl_memory= 71,6719 msec , dim = 600 x 300 x 300 block_size = 8 elements = 180000 speed 0,916484 errors 0
Time on global_memory= 127,75 msec, Time on sharedl_memory= 139,281 msec , dim = 600 x 300 x 600 block_size = 8 elements = 360000 speed 0,917215 errors 0
Time on global_memory= 192,302 msec, Time on sharedl_memory= 208,307 msec , dim = 600 x 300 x 900 block_size = 8 elements = 540000 speed 0,923166 errors 0
Time on global_memory= 253,543 msec, Time on sharedl_memory= 274,415 msec , dim = 600 x 300 x 1200 block_size = 8 elements = 720000 speed 0,923941 errors 0
Time on global_memory= 129,273 msec, Time on sharedl_memory= 139,43 msec , dim = 600 x 600 x 300 block_size = 8 elements = 180000 speed 0,927157 errors 0
Time on global_memory= 250,645 msec, Time on sharedl_memory= 272,152 msec , dim = 600 x 600 x 600 block_size = 8 elements = 360000 speed 0,920972 errors 0
Time on global_memory= 375,705 msec, Time on sharedl_memory= 407,698 msec , dim = 600 x 600 x 900 block_size = 8 elements = 540000 speed 0,921527 errors 0
Time on global_memory= 500,269 msec, Time on sharedl_memory= 538,104 msec , dim = 600 x 600 x 1200 block_size = 8 elements = 720000 speed 0,929689 errors 0
Time on global_memory= 192,305 msec, Time on sharedl_memory= 208,879 msec , dim = 600 x 900 x 300 block_size = 8 elements = 180000 speed 0,920651 errors 0
Time on global_memory= 375,8 msec, Time on sharedl_memory= 409,779 msec , dim = 600 x 900 x 600 block_size = 8 elements = 360000 speed 0,917081 errors 0
Time on global_memory= 566,66 msec, Time on sharedl_memory= 617,787 msec , dim = 600 x 900 x 900 block_size = 8 elements = 540000 speed 0,91724 errors 0
Time on global_memory= 752,885 msec, Time on sharedl_memory= 811,152 msec , dim = 600 x 900 x 1200 block_size = 8 elements = 720000 speed 0,928167 errors 0
Time on global_memory= 255,91 msec, Time on sharedl_memory= 278,353 msec , dim = 600 x 1200 x 300 block_size = 8 elements = 180000 speed 0,919374 errors 0
Time on global_memory= 499,583 msec, Time on sharedl_memory= 542,568 msec , dim = 600 x 1200 x 600 block_size = 8 elements = 360000 speed 0,920775 errors 0
Time on global_memory= 754,734 msec, Time on sharedl_memory= 821,836 msec , dim = 600 x 1200 x 900 block_size = 8 elements = 540000 speed 0,918351 errors 0
Time on global_memory= 999,802 msec, Time on sharedl_memory= 1081,58 msec , dim = 600 x 1200 x 1200 block_size = 8 elements = 720000 speed 0,924386 errors 0
Time on global_memory= 97,8561 msec, Time on sharedl_memory= 106,98 msec , dim = 900 x 300 x 300 block_size = 8 elements = 270000 speed 0,914712 errors 0

Time on global_memory= 191,521 msec, Time on sharedl_memory= 207,776 msec , dim = 900 x 300 x 600 block_size = 8 elements = 540000 speed 0,921765 errors 0
Time on global_memory= 288,525 msec, Time on sharedl_memory= 313,683 msec , dim = 900 x 300 x 900 block_size = 8 elements = 810000 speed 0,919797 errors 0
Time on global_memory= 380,052 msec, Time on sharedl_memory= 411,53 msec , dim = 900 x 300 x 1200 block_size = 8 elements = 1080000 speed 0,923511 errors 0
Time on global_memory= 192,433 msec, Time on sharedl_memory= 208,3 msec , dim = 900 x 600 x 300 block_size = 8 elements = 270000 speed 0,923828 errors 0
Time on global_memory= 374,165 msec, Time on sharedl_memory= 407,682 msec , dim = 900 x 600 x 600 block_size = 8 elements = 540000 speed 0,917787 errors 0
Time on global_memory= 566,235 msec, Time on sharedl_memory= 614,829 msec , dim = 900 x 600 x 900 block_size = 8 elements = 810000 speed 0,920963 errors 0
Time on global_memory= 748,97 msec, Time on sharedl_memory= 810,626 msec , dim = 900 x 600 x 1200 block_size = 8 elements = 1080000 speed 0,92394 errors 0
Time on global_memory= 287,873 msec, Time on sharedl_memory= 314,611 msec , dim = 900 x 900 x 300 block_size = 8 elements = 270000 speed 0,915012 errors 0
Time on global_memory= 565,518 msec, Time on sharedl_memory= 616,857 msec , dim = 900 x 900 x 600 block_size = 8 elements = 540000 speed 0,916773 errors 0
Time on global_memory= 852,653 msec, Time on sharedl_memory= 926,324 msec , dim = 900 x 900 x 900 block_size = 8 elements = 810000 speed 0,920469 errors 0
Time on global_memory= 1128,37 msec, Time on sharedl_memory= 1221,9 msec , dim = 900 x 900 x 1200 block_size = 8 elements = 1080000 speed 0,92345 errors 0
Time on global_memory= 383,83 msec, Time on sharedl_memory= 417,797 msec , dim = 900 x 1200 x 300 block_size = 8 elements = 270000 speed 0,918699 errors 0
Time on global_memory= 750,723 msec, Time on sharedl_memory= 818,974 msec , dim = 900 x 1200 x 600 block_size = 8 elements = 540000 speed 0,916663 errors 0
Time on global_memory= 1134,07 msec, Time on sharedl_memory= 1231,25 msec , dim = 900 x 1200 x 900 block_size = 8 elements = 810000 speed 0,92107 errors 0
Time on global_memory= 1502,94 msec, Time on sharedl_memory= 1623,38 msec , dim = 900 x 1200 x 1200 block_size = 8 elements = 1080000 speed 0,925808 errors 0
Time on global_memory= 129,539 msec, Time on sharedl_memory= 141,184 msec , dim = 1200 x 300 x 300 block_size = 8 elements = 360000 speed 0,917521 errors 0
Time on global_memory= 253,768 msec, Time on sharedl_memory= 276,95 msec , dim = 1200 x 300 x 600 block_size = 8 elements = 720000 speed 0,916293 errors 0
Time on global_memory= 382,65 msec, Time on sharedl_memory= 414,772 msec , dim = 1200 x 300 x 900 block_size = 8 elements = 1080000 speed 0,922556 errors 0
Time on global_memory= 506,684 msec, Time on sharedl_memory= 547,037 msec , dim = 1200 x 300 x 1200 block_size = 8 elements = 1440000 speed 0,926232 errors 0
Time on global_memory= 253,897 msec, Time on sharedl_memory= 276,121 msec , dim = 1200 x 600 x 300 block_size = 8 elements = 360000 speed 0,919513 errors 0
Time on global_memory= 499,396 msec, Time on sharedl_memory= 542,787 msec , dim = 1200 x 600 x 600 block_size = 8 elements = 720000 speed 0,920059 errors 0
Time on global_memory= 750,827 msec, Time on sharedl_memory= 813,785 msec , dim = 1200 x 600 x 900 block_size = 8 elements = 1080000 speed 0,922636 errors 0
Time on global_memory= 995,65 msec, Time on sharedl_memory= 1072,71 msec , dim = 1200 x 600 x 1200 block_size = 8 elements = 1440000 speed 0,928163 errors 0
Time on global_memory= 381,499 msec, Time on sharedl_memory= 416,804 msec , dim = 1200 x 900 x 300 block_size = 8 elements = 360000 speed 0,915295 errors 0
Time on global_memory= 751,586 msec, Time on sharedl_memory= 818,488 msec , dim = 1200 x 900 x 600 block_size = 8 elements = 720000 speed 0,918262 errors 0
Time on global_memory= 1134,29 msec, Time on sharedl_memory= 1230,54 msec , dim = 1200 x 900 x 900 block_size = 8 elements = 1080000 speed 0,921783 errors 0

Time on global_memory= 1502,47 msec, Time on sharedl_memory= 1621,14 msec , dim = 1200 x 900 x 1200 block_size = 8 elements = 1440000 speed 0,926796 errors 0
Time on global_memory= 507,35 msec, Time on sharedl_memory= 554,571 msec , dim = 1200 x 1200 x 300 block_size = 8 elements = 360000 speed 0,914852 errors 0
Time on global_memory= 997,524 msec, Time on sharedl_memory= 1085,11 msec , dim = 1200 x 1200 x 600 block_size = 8 elements = 720000 speed 0,919285 errors 0
Time on global_memory= 1506,79 msec, Time on sharedl_memory= 1634,88 msec , dim = 1200 x 1200 x 900 block_size = 8 elements = 1080000 speed 0,921655 errors 0
Time on global_memory= 2001,57 msec, Time on sharedl_memory= 2155,21 msec , dim = 1200 x 1200 x 1200 block_size = 8 elements = 1440000 speed 0,928712 errors 0

Time on global_memory= 31,8557 msec, Time on sharedl_memory= 24,711 msec , dim = 300 x 300 x 300 block_size = 16 elements = 90000 speed 1,28913 errors 0
Time on global_memory= 62,8986 msec, Time on sharedl_memory= 48,6138 msec , dim = 300 x 300 x 600 block_size = 16 elements = 180000 speed 1,29384 errors 0
Time on global_memory= 92,889 msec, Time on sharedl_memory= 72,0763 msec , dim = 300 x 300 x 900 block_size = 16 elements = 270000 speed 1,28876 errors 0
Time on global_memory= 107,321 msec, Time on sharedl_memory= 84,0291 msec , dim = 300 x 300 x 1200 block_size = 16 elements = 360000 speed 1,27719 errors 0
Time on global_memory= 57,8944 msec, Time on sharedl_memory= 42,8071 msec , dim = 300 x 600 x 300 block_size = 16 elements = 90000 speed 1,35245 errors 0
Time on global_memory= 115,023 msec, Time on sharedl_memory= 81,5709 msec , dim = 300 x 600 x 600 block_size = 16 elements = 180000 speed 1,41009 errors 0
Time on global_memory= 162,638 msec, Time on sharedl_memory= 119,408 msec , dim = 300 x 600 x 900 block_size = 16 elements = 270000 speed 1,36204 errors 0
Time on global_memory= 220,958 msec, Time on sharedl_memory= 156,752 msec , dim = 300 x 600 x 1200 block_size = 16 elements = 360000 speed 1,4096 errors 0
Time on global_memory= 88,6666 msec, Time on sharedl_memory= 61,6416 msec , dim = 300 x 900 x 300 block_size = 16 elements = 90000 speed 1,43842 errors 0
Time on global_memory= 179,197 msec, Time on sharedl_memory= 119,183 msec , dim = 300 x 900 x 600 block_size = 16 elements = 180000 speed 1,50354 errors 0
Time on global_memory= 274,865 msec, Time on sharedl_memory= 178,273 msec , dim = 300 x 900 x 900 block_size = 16 elements = 270000 speed 1,54182 errors 0
Time on global_memory= 370,675 msec, Time on sharedl_memory= 233,891 msec , dim = 300 x 900 x 1200 block_size = 16 elements = 360000 speed 1,58482 errors 0
Time on global_memory= 131,188 msec, Time on sharedl_memory= 78,9431 msec , dim = 300 x 1200 x 300 block_size = 16 elements = 90000 speed 1,6618 errors 0
Time on global_memory= 270,666 msec, Time on sharedl_memory= 157,126 msec , dim = 300 x 1200 x 600 block_size = 16 elements = 180000 speed 1,72261 errors 0
Time on global_memory= 408,196 msec, Time on sharedl_memory= 234,313 msec , dim = 300 x 1200 x 900 block_size = 16 elements = 270000 speed 1,7421 errors 0
Time on global_memory= 541,681 msec, Time on sharedl_memory= 306,526 msec , dim = 300 x 1200 x 1200 block_size = 16 elements = 360000 speed 1,76716 errors 0
Time on global_memory= 52,9196 msec, Time on sharedl_memory= 41,9526 msec , dim = 600 x 300 x 300 block_size = 16 elements = 180000 speed 1,26141 errors 0
Time on global_memory= 103,375 msec, Time on sharedl_memory= 81,4483 msec , dim = 600 x 300 x 600 block_size = 16 elements = 360000 speed 1,26921 errors 0
Time on global_memory= 155,199 msec, Time on sharedl_memory= 121,414 msec , dim = 600 x 300 x 900 block_size = 16 elements = 540000 speed 1,27827 errors 0
Time on global_memory= 204,86 msec, Time on sharedl_memory= 158,733 msec ,

dim = 600 x 300 x 1200 block_size = 16 elements = 720000 speed 1,29059 errors 0
Time on global_memory= 111,927 msec, Time on sharedl_memory= 80,3474 msec ,
dim = 600 x 600 x 300 block_size = 16 elements = 180000 speed 1,39304 errors 0
Time on global_memory= 226,487 msec, Time on sharedl_memory= 159,04 msec , dim = 600 x 600 x 600 block_size = 16 elements = 360000 speed 1,42409 errors 0
Time on global_memory= 336,462 msec, Time on sharedl_memory= 238,78 msec ,
dim = 600 x 600 x 900 block_size = 16 elements = 540000 speed 1,40908 errors 0
Time on global_memory= 453,615 msec, Time on sharedl_memory= 311,262 msec ,
dim = 600 x 600 x 1200 block_size = 16 elements = 720000 speed 1,45734 errors 0
Time on global_memory= 183,26 msec, Time on sharedl_memory= 120,008 msec ,
dim = 600 x 900 x 300 block_size = 16 elements = 180000 speed 1,52707 errors 0
Time on global_memory= 367,459 msec, Time on sharedl_memory= 236,95 msec ,
dim = 600 x 900 x 600 block_size = 16 elements = 360000 speed 1,55079 errors 0
Time on global_memory= 557,579 msec, Time on sharedl_memory= 354,643 msec ,
dim = 600 x 900 x 900 block_size = 16 elements = 540000 speed 1,57223 errors 0
Time on global_memory= 735,303 msec, Time on sharedl_memory= 461,867 msec , dim = 600 x 900 x 1200 block_size = 16 elements = 720000 speed 1,59202 errors 0
Time on global_memory= 267,485 msec, Time on sharedl_memory= 156,79 msec , dim = 600 x 1200 x 300 block_size = 16 elements = 180000 speed 1,70601 errors 0
Time on global_memory= 538,519 msec, Time on sharedl_memory= 309,607 msec ,
dim = 600 x 1200 x 600 block_size = 16 elements = 360000 speed 1,73936 errors 0
Time on global_memory= 803,299 msec, Time on sharedl_memory= 463,671 msec ,
dim = 600 x 1200 x 900 block_size = 16 elements = 540000 speed 1,73248 errors 0
Time on global_memory= 1074,77 msec, Time on sharedl_memory= 606,039 msec ,
dim = 600 x 1200 x 1200 block_size = 16 elements = 720000 speed 1,77344 errors 0
Time on global_memory= 79,6799 msec, Time on sharedl_memory= 61,6835 msec ,
dim = 900 x 300 x 300 block_size = 16 elements = 270000 speed 1,29175 errors 0
Time on global_memory= 156,112 msec, Time on sharedl_memory= 123,045 msec ,
dim = 900 x 300 x 600 block_size = 16 elements = 540000 speed 1,26874 errors 0
Time on global_memory= 233,206 msec, Time on sharedl_memory= 181,305 msec ,
dim = 900 x 300 x 900 block_size = 16 elements = 810000 speed 1,28626 errors 0
Time on global_memory= 308,201 msec, Time on sharedl_memory= 237,355 msec ,
dim = 900 x 300 x 1200 block_size = 16 elements = 1080000 speed 1,29848 errors 0
Time on global_memory= 168,587 msec, Time on sharedl_memory= 119,529 msec ,
dim = 900 x 600 x 300 block_size = 16 elements = 270000 speed 1,41043 errors 0
Time on global_memory= 337,389 msec, Time on sharedl_memory= 236,268 msec ,
dim = 900 x 600 x 600 block_size = 16 elements = 540000 speed 1,428 errors 0
Time on global_memory= 509,348 msec, Time on sharedl_memory= 355,588 msec ,
dim = 900 x 600 x 900 block_size = 16 elements = 810000 speed 1,43241 errors 0
Time on global_memory= 677,183 msec, Time on sharedl_memory= 467,926 msec ,
dim = 900 x 600 x 1200 block_size = 16 elements = 1080000 speed 1,4472 errors 0
Time on global_memory= 275,412 msec, Time on sharedl_memory= 179,275 msec ,
dim = 900 x 900 x 300 block_size = 16 elements = 270000 speed 1,53626 errors 0
Time on global_memory= 553,613 msec, Time on sharedl_memory= 352,508 msec ,
dim = 900 x 900 x 600 block_size = 16 elements = 540000 speed 1,5705 errors 0
Time on global_memory= 825,659 msec, Time on sharedl_memory= 529,087 msec , dim = 900 x 900 x 900 block_size = 16 elements = 810000 speed 1,56053 errors 0
Time on global_memory= 1099,42 msec, Time on sharedl_memory= 693,553 msec ,
dim = 900 x 900 x 1200 block_size = 16 elements = 1080000 speed 1,5852 errors 0
Time on global_memory= 399,122 msec, Time on sharedl_memory= 232,406 msec ,
dim = 900 x 1200 x 300 block_size = 16 elements = 270000 speed 1,71734 errors 0
Time on global_memory= 795,545 msec, Time on sharedl_memory= 463,763 msec ,

dim = 900 x 1200 x 600 block_size = 16 elements = 540000 speed 1,71541 errors 0
Time on global_memory= 1191,63 msec, Time on sharedl_memory= 693,71 msec , dim = 900 x 1200 x 900 block_size = 16 elements = 810000 speed 1,71776 errors 0
Time on global_memory= 1618,88 msec, Time on sharedl_memory= 907,527 msec , dim = 900 x 1200 x 1200 block_size = 16 elements = 1080000 speed 1,78383 errors 0
Time on global_memory= 104,783 msec, Time on sharedl_memory= 80,5461 msec , dim = 1200 x 300 x 300 block_size = 16 elements = 360000 speed 1,30091 errors 0
Time on global_memory= 206,651 msec, Time on sharedl_memory= 159,466 msec , dim = 1200 x 300 x 600 block_size = 16 elements = 720000 speed 1,29589 errors 0
Time on global_memory= 310,074 msec, Time on sharedl_memory= 239,995 msec , dim = 1200 x 300 x 900 block_size = 16 elements = 1080000 speed 1,292 errors 0
Time on global_memory= 408,345 msec, Time on sharedl_memory= 313,783 msec , dim = 1200 x 300 x 1200 block_size = 16 elements = 1440000 speed 1,30136 errors 0
Time on global_memory= 226,536 msec, Time on sharedl_memory= 156,954 msec , dim = 1200 x 600 x 300 block_size = 16 elements = 360000 speed 1,44333 errors 0
Time on global_memory= 453,947 msec, Time on sharedl_memory= 311,725 msec , dim = 1200 x 600 x 600 block_size = 16 elements = 720000 speed 1,45624 errors 0
Time on global_memory= 682,926 msec, Time on sharedl_memory= 466,369 msec , dim = 1200 x 600 x 900 block_size = 16 elements = 1080000 speed 1,46435 errors 0
Time on global_memory= 908,685 msec, Time on sharedl_memory= 612,876 msec , dim = 1200 x 600 x 1200 block_size = 16 elements = 1440000 speed 1,48266 errors 0
Time on global_memory= 372,219 msec, Time on sharedl_memory= 234,734 msec , dim = 1200 x 900 x 300 block_size = 16 elements = 360000 speed 1,58571 errors 0
Time on global_memory= 739,954 msec, Time on sharedl_memory= 464,293 msec , dim = 1200 x 900 x 600 block_size = 16 elements = 720000 speed 1,59372 errors 0
Time on global_memory= 1119,63 msec, Time on sharedl_memory= 695,897 msec , dim = 1200 x 900 x 900 block_size = 16 elements = 1080000 speed 1,60891 errors 0
Time on global_memory= 1484,45 msec, Time on sharedl_memory= 913,653 msec , dim = 1200 x 900 x 1200 block_size = 16 elements = 1440000 speed 1,62475 errors 0
Time on global_memory= 535,418 msec, Time on sharedl_memory= 307,477 msec , dim = 1200 x 1200 x 300 block_size = 16 elements = 360000 speed 1,74133 errors 0
Time on global_memory= 1085,57 msec, Time on sharedl_memory= 608,002 msec , dim = 1200 x 1200 x 600 block_size = 16 elements = 720000 speed 1,78548 errors 0
Time on global_memory= 1621,08 msec, Time on sharedl_memory= 910,524 msec , dim = 1200 x 1200 x 900 block_size = 16 elements = 1080000 speed 1,78038 errors 0
Time on global_memory= 2174,54 msec, Time on sharedl_memory= 1198,42 msec , dim = 1200 x 1200 x 1200 block_size = 16 elements = 1440000 speed 1,8145 errors 0

Time on global_memory= 31,9047 msec, Time on sharedl_memory= 27,0356 msec , dim = 300 x 300 x 300 block_size = 24 elements = 90000 speed 1,1801 errors 0
Time on global_memory= 62,8536 msec, Time on sharedl_memory= 51,2512 msec , dim = 300 x 300 x 600 block_size = 24 elements = 180000 speed 1,22638 errors 0
Time on global_memory= 94,259 msec, Time on sharedl_memory= 77,4728 msec , dim = 300 x 300 x 900 block_size = 24 elements = 270000 speed 1,21667 errors 0
Time on global_memory= 109,253 msec, Time on sharedl_memory= 89,6104 msec , dim = 300 x 300 x 1200 block_size = 24 elements = 360000 speed 1,2192 errors 0
Time on global_memory= 55,672 msec, Time on sharedl_memory= 46,0122 msec , dim = 300 x 600 x 300 block_size = 24 elements = 90000 speed 1,20994 errors 0
Time on global_memory= 109,486 msec, Time on sharedl_memory= 82,6046 msec , dim = 300 x 600 x 600 block_size = 24 elements = 180000 speed 1,32542 errors 0
Time on global_memory= 155,974 msec, Time on sharedl_memory= 123,062 msec , dim = 300 x 600 x 900 block_size = 24 elements = 270000 speed 1,26744 errors 0
Time on global_memory= 206,843 msec, Time on sharedl_memory=

161,691 msec , dim = 300 x 600 x 1200 block_size = 24 elements = 360000 speed 1,27924 errors 0
Time on global_memory= 79,5931 msec, Time on sharedl_memory= 64,91 msec , dim = 300 x 900 x 300 block_size = 24 elements = 90000 speed 1,22621 errors 0
Time on global_memory= 156,076 msec, Time on sharedl_memory= 123,086 msec , dim = 300 x 900 x 600 block_size = 24 elements = 180000 speed 1,26802 errors 0
Time on global_memory= 232,706 msec, Time on sharedl_memory= 186,326 msec , dim = 300 x 900 x 900 block_size = 24 elements = 270000 speed 1,24892 errors 0
Time on global_memory= 308,894 msec, Time on sharedl_memory= 245,388 msec , dim = 300 x 900 x 1200 block_size = 24 elements = 360000 speed 1,2588 errors 0
Time on global_memory= 104,492 msec, Time on sharedl_memory= 84,9056 msec , dim = 300 x 1200 x 300 block_size = 24 elements = 90000 speed 1,23069 errors 0
Time on global_memory= 207,001 msec, Time on sharedl_memory= 161,21 msec , dim = 300 x 1200 x 600 block_size = 24 elements = 180000 speed 1,28404 errors 0
Time on global_memory= 307,194 msec, Time on sharedl_memory= 245,474 msec , dim = 300 x 1200 x 900 block_size = 24 elements = 270000 speed 1,25143 errors 0
Time on global_memory= 409,228 msec, Time on sharedl_memory= 321,293 msec , dim = 300 x 1200 x 1200 block_size = 24 elements = 360000 speed 1,27369 errors 0
Time on global_memory= 52,7779 msec, Time on sharedl_memory= 44,8489 msec , dim = 600 x 300 x 300 block_size = 24 elements = 180000 speed 1,17679 errors 0
Time on global_memory= 103,387 msec, Time on sharedl_memory= 83,3084 msec , dim = 600 x 300 x 600 block_size = 24 elements = 360000 speed 1,24102 errors 0
Time on global_memory= 153,493 msec, Time on sharedl_memory= 126,33 msec , dim = 600 x 300 x 900 block_size = 24 elements = 540000 speed 1,21502 errors 0
Time on global_memory= 203,515 msec, Time on sharedl_memory= 166,203 msec , dim = 600 x 300 x 1200 block_size = 24 elements = 720000 speed 1,22449 errors 0
Time on global_memory= 103,466 msec, Time on sharedl_memory= 82,4151 msec , dim = 600 x 600 x 300 block_size = 24 elements = 180000 speed 1,25543 errors 0
Time on global_memory= 201,365 msec, Time on sharedl_memory= 157,392 msec , dim = 600 x 600 x 600 block_size = 24 elements = 360000 speed 1,27938 errors 0
Time on global_memory= 301,264 msec, Time on sharedl_memory= 237,165 msec , dim = 600 x 600 x 900 block_size = 24 elements = 540000 speed 1,27027 errors 0
Time on global_memory= 402,094 msec, Time on sharedl_memory= 313,682 msec , dim = 600 x 600 x 1200 block_size = 24 elements = 720000 speed 1,28185 errors 0
Time on global_memory= 153,01 msec, Time on sharedl_memory= 123,925 msec , dim = 600 x 900 x 300 block_size = 24 elements = 180000 speed 1,2347 errors 0
Time on global_memory= 301,362 msec, Time on sharedl_memory= 237,037 msec , dim = 600 x 900 x 600 block_size = 24 elements = 360000 speed 1,27137 errors 0
Time on global_memory= 453,465 msec, Time on sharedl_memory= 358,859 msec , dim = 600 x 900 x 900 block_size = 24 elements = 540000 speed 1,26363 errors 0
Time on global_memory= 603,005 msec, Time on sharedl_memory= 471,098 msec , dim = 600 x 900 x 1200 block_size = 24 elements = 720000 speed 1,28 errors 0
Time on global_memory= 203,447 msec, Time on sharedl_memory= 163,011 msec , dim = 600 x 1200 x 300 block_size = 24 elements = 180000 speed 1,24805 errors 0
Time on global_memory= 403,415 msec, Time on sharedl_memory= 310,435 msec , dim = 600 x 1200 x 600 block_size = 24 elements = 360000 speed 1,29951 errors 0
Time on global_memory= 601,904 msec, Time on sharedl_memory= 469,572 msec , dim = 600 x 1200 x 900 block_size = 24 elements = 540000 speed 1,28181 errors 0
Time on global_memory= 795,493 msec, Time on sharedl_memory= 614,529 msec , dim = 600 x 1200 x 1200 block_size = 24 elements = 720000 speed 1,29448 errors 0
Time on global_memory= 78,3916 msec, Time on sharedl_memory= 66,4088 msec , dim = 900 x 300 x 300 block_size = 24 elements = 270000 speed 1,18044 errors 0
Time on global_memory= 154,697 msec, Time on sharedl_memory= 126,389 msec ,

dim = 900 x 300 x 600 block_size = 24 elements = 540000 speed 1,22397 errors 0
Time on global_memory= 231,561 msec, Time on sharedl_memory= 190,742 msec ,
dim = 900 x 300 x 900 block_size = 24 elements = 810000 speed 1,214 errors 0
Time on global_memory= 306,005 msec, Time on sharedl_memory= 252,102 msec ,
dim = 900 x 300 x 1200 block_size = 24 elements = 1080000 speed 1,21382 errors 0
Time on global_memory= 154,728 msec, Time on sharedl_memory= 124,674 msec ,
dim = 900 x 600 x 300 block_size = 24 elements = 270000 speed 1,24107 errors 0
Time on global_memory= 303,535 msec, Time on sharedl_memory= 236,619 msec ,
dim = 900 x 600 x 600 block_size = 24 elements = 540000 speed 1,2828 errors 0
Time on global_memory= 455,713 msec, Time on sharedl_memory= 358,514 msec ,
dim = 900 x 600 x 900 block_size = 24 elements = 810000 speed 1,27112 errors 0
Time on global_memory= 604,219 msec, Time on sharedl_memory= 472,025 msec ,
dim = 900 x 600 x 1200 block_size = 24 elements = 1080000 speed 1,28006 errors 0
Time on global_memory= 231,125 msec, Time on sharedl_memory= 187,669 msec ,
dim = 900 x 900 x 300 block_size = 24 elements = 270000 speed 1,23155 errors 0
Time on global_memory= 454,798 msec, Time on sharedl_memory= 357,608 msec ,
dim = 900 x 900 x 600 block_size = 24 elements = 540000 speed 1,27178 errors 0
Time on global_memory= 685,156 msec, Time on sharedl_memory= 541,814 msec , dim = 900 x 900 x 900 block_size = 24 elements = 810000 speed 1,26456 errors 0
Time on global_memory= 908,066 msec, Time on sharedl_memory= 710,415 msec ,
dim = 900 x 900 x 1200 block_size = 24 elements = 1080000 speed 1,27822 errors 0
Time on global_memory= 309,695 msec, Time on sharedl_memory= 245,294 msec ,
dim = 900 x 1200 x 300 block_size = 24 elements = 270000 speed 1,26255 errors 0
Time on global_memory= 603,827 msec, Time on sharedl_memory= 468,109 msec ,
dim = 900 x 1200 x 600 block_size = 24 elements = 540000 speed 1,28993 errors 0
Time on global_memory= 903,315 msec, Time on sharedl_memory= 711,579 msec ,
dim = 900 x 1200 x 900 block_size = 24 elements = 810000 speed 1,26945 errors 0
Time on global_memory= 1199 msec, Time on sharedl_memory= 932,772 msec , dim =
900 x 1200 x 1200 block_size = 24 elements = 1080000 speed 1,28542 errors 0
Time on global_memory= 103,577 msec, Time on sharedl_memory= 87,5421 msec ,
dim = 1200 x 300 x 300 block_size = 24 elements = 360000 speed 1,18317 errors 0
Time on global_memory= 203,172 msec, Time on sharedl_memory= 165,571 msec ,
dim = 1200 x 300 x 600 block_size = 24 elements = 720000 speed 1,2271 errors 0
Time on global_memory= 306,714 msec, Time on sharedl_memory= 250,362 msec ,
dim = 1200 x 300 x 900 block_size = 24 elements = 1080000 speed 1,22509 errors 0
Time on global_memory= 405,562 msec, Time on sharedl_memory= 330,049 msec ,
dim = 1200 x 300 x 1200 block_size = 24 elements = 1440000 speed 1,2288 errors 0
Time on global_memory= 204,048 msec, Time on sharedl_memory= 162,98 msec ,
dim = 1200 x 600 x 300 block_size = 24 elements = 360000 speed 1,25199 errors 0
Time on global_memory= 400,217 msec, Time on sharedl_memory= 311,917 msec ,
dim = 1200 x 600 x 600 block_size = 24 elements = 720000 speed 1,28309 errors 0
Time on global_memory= 598,997 msec, Time on sharedl_memory= 471,698 msec ,
dim = 1200 x 600 x 900 block_size = 24 elements = 1080000 speed 1,26987 errors 0
Time on global_memory= 799,324 msec, Time on sharedl_memory= 620,95 msec ,
dim = 1200 x 600 x 1200 block_size = 24 elements = 1440000 speed 1,28726 errors 0
Time on global_memory= 307,517 msec, Time on sharedl_memory= 245,463 msec ,
dim = 1200 x 900 x 300 block_size = 24 elements = 360000 speed 1,2528 errors 0
Time on global_memory= 602,334 msec, Time on sharedl_memory= 470,06 msec ,
dim = 1200 x 900 x 600 block_size = 24 elements = 720000 speed 1,2814 errors 0
Time on global_memory= 902,79 msec, Time on sharedl_memory= 713,62 msec , dim
= 1200 x 900 x 900 block_size = 24 elements = 1080000 speed 1,26509 errors 0
Time on global_memory= 1199,05 msec, Time on sharedl_memory= 935,901 msec ,
dim = 1200 x 900 x 1200 block_size = 24 elements = 1440000 speed 1,28117 errors 0
Time on global_memory= 409,606 msec, Time on sharedl_memory= 322,201 msec ,

dim = 1200 x 1200 x 300 block_size = 24 elements = 360000 speed 1,27128 errors 0
Time on global_memory= 793,907 msec, Time on sharedl_memory= 614,982 msec ,
dim = 1200 x 1200 x 600 block_size = 24 elements = 720000 speed 1,29094 errors 0
Time on global_memory= 1192,05 msec, Time on sharedl_memory= 935,171 msec ,
dim = 1200 x 1200 x 900 block_size = 24 elements = 1080000 speed 1,27468 errors 0
Time on global_memory= 1586,91 msec, Time on sharedl_memory= 1225,81 msec , dim = 1200 x 1200 x 1200 block_size = 24 elements = 1440000 speed 1,29458 errors 0

Time on global_memory= 39,6263 msec, Time on sharedl_memory= 26,2492 msec , dim = 300 x 300 x 300 block_size = 29 elements = 90000 speed 1,50962 errors 0
Time on global_memory= 75,9567 msec, Time on sharedl_memory= 50,6906 msec ,
dim = 300 x 300 x 600 block_size = 29 elements = 180000 speed 1,49844 errors 0
Time on global_memory= 113,651 msec, Time on sharedl_memory= 74,0342 msec ,
dim = 300 x 300 x 900 block_size = 29 elements = 270000 speed 1,53511 errors 0
Time on global_memory= 149,656 msec, Time on sharedl_memory= 96,88 msec , dim = 300 x 300 x 1200 block_size = 29 elements = 360000 speed 1,54476 errors 0
Time on global_memory= 70,4601 msec, Time on sharedl_memory= 43,786 msec ,
dim = 300 x 600 x 300 block_size = 29 elements = 90000 speed 1,60919 errors 0
Time on global_memory= 131,931 msec, Time on sharedl_memory= 81,7078 msec ,
dim = 300 x 600 x 600 block_size = 29 elements = 180000 speed 1,61467 errors 0
Time on global_memory= 196,377 msec, Time on sharedl_memory= 118,304 msec ,
dim = 300 x 600 x 900 block_size = 29 elements = 270000 speed 1,65994 errors 0
Time on global_memory= 252,585 msec, Time on sharedl_memory= 155,504 msec , dim = 300 x 600 x 1200 block_size = 29 elements = 360000 speed 1,62431 errors 0
Time on global_memory= 98,4963 msec, Time on sharedl_memory= 62,7513 msec ,
dim = 300 x 900 x 300 block_size = 29 elements = 90000 speed 1,56963 errors 0
Time on global_memory= 188,956 msec, Time on sharedl_memory= 117,842 msec ,
dim = 300 x 900 x 600 block_size = 29 elements = 180000 speed 1,60346 errors 0
Time on global_memory= 289,108 msec, Time on sharedl_memory= 178,694 msec ,
dim = 300 x 900 x 900 block_size = 29 elements = 270000 speed 1,61789 errors 0
Time on global_memory= 381,753 msec, Time on sharedl_memory= 233,332 msec ,
dim = 300 x 900 x 1200 block_size = 29 elements = 360000 speed 1,6361 errors 0
Time on global_memory= 132,4 msec, Time on sharedl_memory= 81,9033 msec , dim = 300 x 1200 x 300 block_size = 29 elements = 90000 speed 1,61654 errors 0
Time on global_memory= 254,021 msec, Time on sharedl_memory= 154,914 msec ,
dim = 300 x 1200 x 600 block_size = 29 elements = 180000 speed 1,63976 errors 0
Time on global_memory= 384,759 msec, Time on sharedl_memory= 235,083 msec ,
dim = 300 x 1200 x 900 block_size = 29 elements = 270000 speed 1,6367 errors 0
Time on global_memory= 514,209 msec, Time on sharedl_memory= 305,484 msec ,
dim = 300 x 1200 x 1200 block_size = 29 elements = 360000 speed 1,68326 errors 0
Time on global_memory= 63,1619 msec, Time on sharedl_memory= 42,9102 msec ,
dim = 600 x 300 x 300 block_size = 29 elements = 180000 speed 1,47196 errors 0
Time on global_memory= 121,721 msec, Time on sharedl_memory= 79,862 msec , dim = 600 x 300 x 600 block_size = 29 elements = 360000 speed 1,52414 errors 0
Time on global_memory= 184,092 msec, Time on sharedl_memory= 121,563 msec ,
dim = 600 x 300 x 900 block_size = 29 elements = 540000 speed 1,51437 errors 0
Time on global_memory= 242,617 msec, Time on sharedl_memory= 159,132 msec ,
dim = 600 x 300 x 1200 block_size = 29 elements = 720000 speed 1,52463 errors 0
Time on global_memory= 126,715 msec, Time on sharedl_memory= 79,3799 msec ,
dim = 600 x 600 x 300 block_size = 29 elements = 180000 speed 1,59631 errors 0
Time on global_memory= 244,028 msec, Time on sharedl_memory=

149,272 msec , dim = 600 x 600 x 600 block_size = 29 elements = 360000 speed 1,63479 errors 0
Time on global_memory= 368,579 msec, Time on sharedl_memory= 225,625 msec , dim = 600 x 600 x 900 block_size = 29 elements = 540000 speed 1,63359 errors 0
Time on global_memory= 491,04 msec, Time on sharedl_memory= 295,508 msec , dim = 600 x 600 x 1200 block_size = 29 elements = 720000 speed 1,66168 errors 0
Time on global_memory= 189,905 msec, Time on sharedl_memory= 119,973 msec , dim = 600 x 900 x 300 block_size = 29 elements = 180000 speed 1,58289 errors 0
Time on global_memory= 366,973 msec, Time on sharedl_memory= 224,935 msec , dim = 600 x 900 x 600 block_size = 29 elements = 360000 speed 1,63146 errors 0
Time on global_memory= 563,736 msec, Time on sharedl_memory= 341,151 msec , dim = 600 x 900 x 900 block_size = 29 elements = 540000 speed 1,65246 errors 0
Time on global_memory= 746,945 msec, Time on sharedl_memory= 446,416 msec , dim = 600 x 900 x 1200 block_size = 29 elements = 720000 speed 1,6732 errors 0
Time on global_memory= 256,892 msec, Time on sharedl_memory= 154,927 msec , dim = 600 x 1200 x 300 block_size = 29 elements = 180000 speed 1,65815 errors 0
Time on global_memory= 494,134 msec, Time on sharedl_memory= 292,788 msec , dim = 600 x 1200 x 600 block_size = 29 elements = 360000 speed 1,68769 errors 0
Time on global_memory= 746,282 msec, Time on sharedl_memory= 444,37 msec , dim = 600 x 1200 x 900 block_size = 29 elements = 540000 speed 1,67942 errors 0
Time on global_memory= 1001,74 msec, Time on sharedl_memory= 582,209 msec , dim = 600 x 1200 x 1200 block_size = 29 elements = 720000 speed 1,72059 errors 0
Time on global_memory= 95,1448 msec, Time on sharedl_memory= 63,9411 msec , dim = 900 x 300 x 300 block_size = 29 elements = 270000 speed 1,48801 errors 0
Time on global_memory= 183,032 msec, Time on sharedl_memory= 120,442 msec , dim = 900 x 300 x 600 block_size = 29 elements = 540000 speed 1,51967 errors 0
Time on global_memory= 278,09 msec, Time on sharedl_memory= 182,612 msec , dim = 900 x 300 x 900 block_size = 29 elements = 810000 speed 1,52284 errors 0
Time on global_memory= 366,776 msec, Time on sharedl_memory= 239,854 msec , dim = 900 x 300 x 1200 block_size = 29 elements = 1080000 speed 1,52916 errors 0
Time on global_memory= 190,802 msec, Time on sharedl_memory= 118,669 msec , dim = 900 x 600 x 300 block_size = 29 elements = 270000 speed 1,60785 errors 0
Time on global_memory= 368,176 msec, Time on sharedl_memory= 225,333 msec , dim = 900 x 600 x 600 block_size = 29 elements = 540000 speed 1,63392 errors 0
Time on global_memory= 559,707 msec, Time on sharedl_memory= 341,927 msec , dim = 900 x 600 x 900 block_size = 29 elements = 810000 speed 1,63692 errors 0
Time on global_memory= 737,899 msec, Time on sharedl_memory= 448,439 msec , dim = 900 x 600 x 1200 block_size = 29 elements = 1080000 speed 1,64549 errors 0
Time on global_memory= 290,619 msec, Time on sharedl_memory= 178,685 msec , dim = 900 x 900 x 300 block_size = 29 elements = 270000 speed 1,62644 errors 0
Time on global_memory= 558,049 msec, Time on sharedl_memory= 338,847 msec , dim = 900 x 900 x 600 block_size = 29 elements = 540000 speed 1,6469 errors 0
Time on global_memory= 846,206 msec, Time on sharedl_memory= 515,178 msec , dim = 900 x 900 x 900 block_size = 29 elements = 810000 speed 1,64255 errors 0
Time on global_memory= 1121,62 msec, Time on sharedl_memory= 675,614 msec , dim = 900 x 900 x 1200 block_size = 29 elements = 1080000 speed 1,66015 errors 0
Time on global_memory= 384,885 msec, Time on sharedl_memory= 235,691 msec , dim = 900 x 1200 x 300 block_size = 29 elements = 270000 speed 1,63301 errors 0
Time on global_memory= 745,667 msec, Time on sharedl_memory= 444,381 msec , dim = 900 x 1200 x 600 block_size = 29 elements = 540000 speed 1,67799 errors 0
Time on global_memory= 1128,42 msec, Time on sharedl_memory= 675,437 msec , dim = 900 x 1200 x 900 block_size = 29 elements = 810000 speed 1,67066 errors 0
Time on global_memory= 1502,88 msec, Time on sharedl_memory= 884,207 msec ,

dim = 900 x 1200 x 1200 block_size = 29 elements = 1080000 speed 1,69969 errors 0
Time on global_memory= 124,98 msec, Time on sharedl_memory= 83,6624 msec , dim = 1200 x 300 x 300 block_size = 29 elements = 360000 speed 1,49386 errors 0
Time on global_memory= 242,589 msec, Time on sharedl_memory= 158,219 msec , dim = 1200 x 300 x 600 block_size = 29 elements = 720000 speed 1,53325 errors 0
Time on global_memory= 366,422 msec, Time on sharedl_memory= 241,287 msec , dim = 1200 x 300 x 900 block_size = 29 elements = 1080000 speed 1,51862 errors 0
Time on global_memory= 486,896 msec, Time on sharedl_memory= 314,394 msec , dim = 1200 x 300 x 1200 block_size = 29 elements = 1440000 speed 1,54868 errors 0
Time on global_memory= 251,16 msec, Time on sharedl_memory= 155,993 msec , dim = 1200 x 600 x 300 block_size = 29 elements = 360000 speed 1,61008 errors 0
Time on global_memory= 486,877 msec, Time on sharedl_memory= 296,117 msec , dim = 1200 x 600 x 600 block_size = 29 elements = 720000 speed 1,64421 errors 0
Time on global_memory= 737,524 msec, Time on sharedl_memory= 448,41 msec , dim = 1200 x 600 x 900 block_size = 29 elements = 1080000 speed 1,64476 errors 0
Time on global_memory= 976,443 msec, Time on sharedl_memory= 589,929 msec , dim = 1200 x 600 x 1200 block_size = 29 elements = 1440000 speed 1,65519 errors 0
Time on global_memory= 377,776 msec, Time on sharedl_memory= 234,327 msec , dim = 1200 x 900 x 300 block_size = 29 elements = 360000 speed 1,61217 errors 0
Time on global_memory= 741,835 msec, Time on sharedl_memory= 445,283 msec , dim = 1200 x 900 x 600 block_size = 29 elements = 720000 speed 1,66598 errors 0
Time on global_memory= 1118,5 msec, Time on sharedl_memory= 677,981 msec , dim = 1200 x 900 x 900 block_size = 29 elements = 1080000 speed 1,64974 errors 0
Time on global_memory= 1483,5 msec, Time on sharedl_memory= 886,374 msec , dim = 1200 x 900 x 1200 block_size = 29 elements = 1440000 speed 1,67367 errors 0
Time on global_memory= 508,121 msec, Time on sharedl_memory= 307,689 msec , dim = 1200 x 1200 x 300 block_size = 29 elements = 360000 speed 1,65141 errors 0
Time on global_memory= 983,573 msec, Time on sharedl_memory= 583,138 msec , dim = 1200 x 1200 x 600 block_size = 29 elements = 720000 speed 1,68669 errors 0
Time on global_memory= 1492,14 msec, Time on sharedl_memory= 884,729 msec , dim = 1200 x 1200 x 900 block_size = 29 elements = 1080000 speed 1,68655 errors 0
Time on global_memory= 1997,35 msec, Time on sharedl_memory= 1160,66 msec , dim = 1200 x 1200 x 1200 block_size = 29 elements = 1440000 speed 1,72088 errors 0

Comparación de multiplicaciones entre la GPU y la CPU

Time on CPU= 158 msec, Time on GPU= 24,9288 msec , dim = 300 x 300 x 300 block_size = 29 elements = 90000 speed 6,33805 errors 0
Time on CPU= 281 msec, Time on GPU= 44,6428 msec , dim = 300 x 300 x 600 block_size = 29 elements = 180000 speed 6,29441 errors 0
Time on CPU= 593 msec, Time on GPU= 66,3966 msec , dim = 300 x 300 x 900 block_size = 29 elements = 270000 speed 8,93118 errors 0
Time on CPU= 708 msec, Time on GPU= 86,386 msec , dim = 300 x 300 x 1200 block_size = 29 elements = 360000 speed 8,19577 errors 0
Time on CPU= 328 msec, Time on GPU= 44,2492 msec , dim = 300 x 600 x 300 block_size = 29 elements = 90000 speed 7,41256 errors 0
Time on CPU= 711 msec, Time on GPU= 82,2995 msec , dim = 300 x 600 x 600 block_size = 29 elements = 180000 speed 8,63918 errors 0

Time on CPU= 1160 msec, Time on GPU= 123,454 msec , dim = 300 x 600 x 900 block_size = 29 elements = 270000 speed 9,39623 errors 0
Time on CPU= 1639 msec, Time on GPU= 146,805 msec , dim = 300 x 600 x 1200 block_size = 29 elements = 360000 speed 11,1645 errors 0
Time on CPU= 551 msec, Time on GPU= 62,9251 msec , dim = 300 x 900 x 300 block_size = 29 elements = 90000 speed 8,75644 errors 0
Time on CPU= 1153 msec, Time on GPU= 110,727 msec , dim = 300 x 900 x 600 block_size = 29 elements = 180000 speed 10,413 errors 0
Time on CPU= 1770 msec, Time on GPU= 166,927 msec , dim = 300 x 900 x 900 block_size = 29 elements = 270000 speed 10,6034 errors 0
Time on CPU= 2525 msec, Time on GPU= 217,815 msec , dim = 300 x 900 x 1200 block_size = 29 elements = 360000 speed 11,5924 errors 0
Time on CPU= 726 msec, Time on GPU= 78,0749 msec , dim = 300 x 1200 x 300 block_size = 29 elements = 90000 speed 9,29876 errors 0
Time on CPU= 1462 msec, Time on GPU= 144,787 msec , dim = 300 x 1200 x 600 block_size = 29 elements = 180000 speed 10,0976 errors 0
Time on CPU= 2404 msec, Time on GPU= 218,217 msec , dim = 300 x 1200 x 900 block_size = 29 elements = 270000 speed 11,0166 errors 0
Time on CPU= 3811 msec, Time on GPU= 285,046 msec , dim = 300 x 1200 x 1200 block_size = 29 elements = 360000 speed 13,3698 errors 0
Time on CPU= 294 msec, Time on GPU= 39,7912 msec , dim = 600 x 300 x 300 block_size = 29 elements = 180000 speed 7,38857 errors 0
Time on CPU= 554 msec, Time on GPU= 75,0758 msec , dim = 600 x 300 x 600 block_size = 29 elements = 360000 speed 7,3792 errors 0
Time on CPU= 1088 msec, Time on GPU= 111,432 msec , dim = 600 x 300 x 900 block_size = 29 elements = 540000 speed 9,7638 errors 0
Time on CPU= 1312 msec, Time on GPU= 145,721 msec , dim = 600 x 300 x 1200 block_size = 29 elements = 720000 speed 9,00353 errors 0
Time on CPU= 615 msec, Time on GPU= 74,7777 msec , dim = 600 x 600 x 300 block_size = 29 elements = 180000 speed 8,22438 errors 0
Time on CPU= 1379 msec, Time on GPU= 137,989 msec , dim = 600 x 600 x 600 block_size = 29 elements = 360000 speed 9,99356 errors 0
Time on CPU= 2198 msec, Time on GPU= 208,029 msec , dim = 600 x 600 x 900 block_size = 29 elements = 540000 speed 10,5659 errors 0
Time on CPU= 3026 msec, Time on GPU= 272,347 msec , dim = 600 x 600 x 1200 block_size = 29 elements = 720000 speed 11,1108 errors 0
Time on CPU= 1011 msec, Time on GPU= 110,931 msec , dim = 600 x 900 x 300 block_size = 29 elements = 180000 speed 9,11381 errors 0
Time on CPU= 2114 msec, Time on GPU= 207,844 msec , dim = 600 x 900 x 600 block_size = 29 elements = 360000 speed 10,1711 errors 0
Time on CPU= 3405 msec, Time on GPU= 314,402 msec , dim = 600 x 900 x 900 block_size = 29 elements = 540000 speed 10,8301 errors 0
Time on CPU= 4755 msec, Time on GPU= 438,631 msec , dim = 600 x 900 x 1200 block_size = 29 elements = 720000 speed 10,8406 errors 0
Time on CPU= 1365 msec, Time on GPU= 144,915 msec , dim = 600 x 1200 x 300 block_size = 29 elements = 180000 speed 9,41929 errors 0
Time on CPU= 2775 msec, Time on GPU= 274,574 msec , dim = 600 x 1200 x 600 block_size = 29 elements = 360000 speed 10,1066 errors 0
Time on CPU= 4534 msec, Time on GPU= 410,724 msec , dim = 600 x 1200 x 900 block_size = 29 elements = 540000 speed 11,039 errors 0
Time on CPU= 7197 msec, Time on GPU= 538,148 msec , dim = 600 x 1200 x 1200 block_size = 29 elements = 720000 speed 13,3736 errors 0
Time on CPU= 431 msec, Time on GPU= 59,4522 msec , dim = 900 x 300 x 300 block_size = 29 elements = 270000 speed 7,24953 errors 0
Time on CPU= 844 msec, Time on GPU= 111,844 msec , dim = 900 x 300 x 600

block_size = 29 elements = 540000 speed 7,54624 errors 0
Time on CPU= 1641 msec, Time on GPU= 170,585 msec , dim = 900 x 300 x 900
block_size = 29 elements = 810000 speed 9,61981 errors 0
Time on CPU= 2008 msec, Time on GPU= 221,446 msec , dim = 900 x 300 x 1200
block_size = 29 elements = 1080000 speed 9,06768 errors 0
Time on CPU= 951 msec, Time on GPU= 110,593 msec , dim = 900 x 600 x 300
block_size = 29 elements = 270000 speed 8,59911 errors 0
Time on CPU= 2095 msec, Time on GPU= 208,501 msec , dim = 900 x 600 x 600
block_size = 29 elements = 540000 speed 10,0479 errors 0
Time on CPU= 3354 msec, Time on GPU= 314,933 msec , dim = 900 x 600 x 900
block_size = 29 elements = 810000 speed 10,6499 errors 0
Time on CPU= 4563 msec, Time on GPU= 413,054 msec , dim = 900 x 600 x 1200
block_size = 29 elements = 1080000 speed 11,047 errors 0
Time on CPU= 1545 msec, Time on GPU= 167,801 msec , dim = 900 x 900 x 300
block_size = 29 elements = 270000 speed 9,20735 errors 0
Time on CPU= 3245 msec, Time on GPU= 314,16 msec , dim = 900 x 900 x 600
block_size = 29 elements = 540000 speed 10,3291 errors 0
Time on CPU= 5127 msec, Time on GPU= 500,753 msec , dim = 900 x 900 x 900 block_size = 29 elements = 810000 speed 10,2386 errors 0
Time on CPU= 7289 msec, Time on GPU= 624,673 msec , dim = 900 x 900 x 1200
block_size = 29 elements = 1080000 speed 11,6685 errors 0
Time on CPU= 2086 msec, Time on GPU= 217,826 msec , dim = 900 x 1200 x 300
block_size = 29 elements = 270000 speed 9,57644 errors 0
Time on CPU= 4230 msec, Time on GPU= 410,579 msec , dim = 900 x 1200 x 600
block_size = 29 elements = 540000 speed 10,3025 errors 0
Time on CPU= 7450 msec, Time on GPU= 622,64 msec , dim = 900 x 1200 x 900
block_size = 29 elements = 810000 speed 11,9652 errors 0
Time on CPU= 11499 msec, Time on GPU= 816,923 msec , dim = 900 x 1200 x 1200
block_size = 29 elements = 1080000 speed 14,076 errors 0
Time on CPU= 558 msec, Time on GPU= 77,6001 msec , dim = 1200 x 300 x 300
block_size = 29 elements = 360000 speed 7,19071 errors 0
Time on CPU= 1096 msec, Time on GPU= 145,963 msec , dim = 1200 x 300 x 600
block_size = 29 elements = 720000 speed 7,50878 errors 0
Time on CPU= 2158 msec, Time on GPU= 221,113 msec , dim = 1200 x 300 x 900
block_size = 29 elements = 1080000 speed 9,75969 errors 0
Time on CPU= 2616 msec, Time on GPU= 308,787 msec , dim = 1200 x 300 x 1200
block_size = 29 elements = 1440000 speed 8,47186 errors 0
Time on CPU= 1250 msec, Time on GPU= 144,224 msec , dim = 1200 x 600 x 300
block_size = 29 elements = 360000 speed 8,66705 errors 0
Time on CPU= 2772 msec, Time on GPU= 272,215 msec , dim = 1200 x 600 x 600
block_size = 29 elements = 720000 speed 10,1831 errors 0
Time on CPU= 4379 msec, Time on GPU= 412,437 msec , dim = 1200 x 600 x 900
block_size = 29 elements = 1080000 speed 10,6174 errors 0
Time on CPU= 5960 msec, Time on GPU= 540,655 msec , dim = 1200 x 600 x 1200
block_size = 29 elements = 1440000 speed 11,0237 errors 0
Time on CPU= 2014 msec, Time on GPU= 217,95 msec , dim = 1200 x 900 x 300
block_size = 29 elements = 360000 speed 9,24064 errors 0
Time on CPU= 4185 msec, Time on GPU= 411,146 msec , dim = 1200 x 900 x 600
block_size = 29 elements = 720000 speed 10,1789 errors 0
Time on CPU= 6755 msec, Time on GPU= 623,504 msec , dim = 1200 x 900 x 900
block_size = 29 elements = 1080000 speed 10,8339 errors 0
Time on CPU= 9489 msec, Time on GPU= 816,79 msec , dim = 1200 x 900 x 1200
block_size = 29 elements = 1440000 speed 11,6174 errors 0
Time on CPU= 2755 msec, Time on GPU= 284,834 msec , dim = 1200 x 1200 x 300
block_size = 29 elements = 360000 speed 9,67229 errors 0

Time on CPU= 5599 msec, Time on GPU= 559,532 msec , dim = 1200 x 1200 x 600 block_size = 29 elements = 720000 speed 10,0066 errors 0
Time on CPU= 9096 msec, Time on GPU= 815,325 msec , dim = 1200 x 1200 x 900 block_size = 29 elements = 1080000 speed 11,1563 errors 0
Time on CPU= 14630 msec, Time on GPU= 1068,63 msec , dim = 1200 x 1200 x 1200 block_size = 29 elements = 1440000 speed 13,6904 errors 0