View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositorio Institucional da Universidade de Santiago de Compostela

MASTER EN COMPUTACIÓN DE ALTAS PRESTACIONES

PROYECTO FIN DE MÁSTER

Desarrollo sobre GPU de técnicas para la detección de objetivos en imágenes hiperespectrales mediante la utilización de redes neuronales.

Autor : Juan López Gómez

Directores: Dora Blanco Heras

Francisco Argüello Pedreira

Santiago de Compostela, 1 de Septiembre de 2011

Resumen

En este trabajo se presentan dos algoritmos de detección de objetivos en imágenes hiperespectrales específicamente desarrollados para su implementación sobre GPU, ambos basados en la aplicación de ANNs *(Artificial Neural Networks)*.

El primer algoritmo, denominado algoritmo de detección de objetivos a nivel de píxel, basa su búsqueda en la exploración píxel a píxel de la imagen hiperespectral, detectando si en cada uno de ellos se encuentra el objetivo buscado, o una parte del mismo. El segundo algoritmo, denominado algoritmo de detección de objetivos multi-resolución, basa su búsqueda en la exploración jerárquica de áreas de tamaño decreciente de imagen (volúmenes hiperespectrales), detectando y acotando el objetivo independientemente de la escala a la que éste se encuentre.

En la implementación sobre GPU de las ANNs utilizadas en ambos algoritmos se analizan dos aproximaciones diferentes de paralelización: paralelismo a nivel neuronal, y paralelismo a nivel de enlace sináptico. Además, se tienen en cuenta un gran número de estrategias de optimización específicas para GPU, con el fin de explotar adecuadamente la enorme capacidad de cómputo de las tarjetas, y de ocultar la latencia en los accesos a memoria.

En la fase de resultados los algoritmos son testeados mediante la búsqueda de objetivos sobre dos tipos diferentes de imágenes hiperespectrales, una aplicada al reconocimiento de materiales, y otra aplicada a funciones de búsqueda y rescate. Los tiempos de ejecución obtenidos muestran la efectividad de los algoritmos de detección desarrollados, así como la conveniencia de su implementación sobre GPU.

Este trabajo es fruto de la colaboración entre el Grupo Integrado de Ingeniería (GII) de la UDC y el grupo de Arquitectura de Computadores de la USC y financiado por la red GHPC-2.

ÍNDICE

ÍNDICE DE FIGURAS 4

ÍNDICE DE TABLAS 5

1. Introducción 6

1.1. Imágenes hiperespectrales: definición y aplicaciones. 6

1.2. Procesado de imágenes hiperespectrales. 7

1.3. Detección de objetivos mediante ANNs: objetivo del trabajo. 8

2. Algoritmos para la detección de objetivos. 10

2.1. Introducción y consideraciones generales. 10

2.2. Algoritmo de detección de objetivos a nivel de píxel. 11

2.3. Algoritmo de detección de objetivos multi-resolución. 13

3. Visión general de la arquitectura CUDA. 16

4. Implementación paralela. 20

4.1. Introducción y consideraciones generales. 20

4.2. Algoritmo de detección de objetivos a nivel de píxel. 21

4.2.1. Paralelismo a nivel neuronal. 22

4.2.2. Paralelismo a nivel sináptico. 23

4.3. Algoritmo de detección de objetivos multi-resolución. 26

4.3.1. Kernel de reducción inicial para bloques 8×8 píxeles. 27

4.3.2. Kernel de entrada ANN para cada ventana. 29

4.3.3. Kernel ANN-1 y ANN-2. 30

5. Análisis y resultados. 31

5.1. Procedimiento y equipo utilizado. 31

5.2. Resultados. 33

5.2.1. Algoritmo de detección de objetivos a nivel de píxel. 33
5.2.2. Algoritmo de detección de objetivos multi-resolución. 34
5.3. Análisis de rendimiento y discusión de resultados. 36
5.3.1. Transferencia CPU-GPU de las imágenes. 37
5.3.2. Dependencia con el tamaño de bloque. 37
5.3.3. Dependencia con el tamaño de la caché L1. 38
5.3.4. Aprovechamiento de los recursos de la GPU. 39
5.3.5. Ancho de banda y rendimiento en GFLOPS. 41

BIBLIOGRAFÍA **46**

ÍNDICE DE FIGURAS

1.1. Imagen hiperespectral. 6

2.1. Algoritmo de detección de objetivos a nivel de píxel. 11
2.2. ANN utilizada por el algoritmo de detección de objetivos a nivel de píxel. 12
2.3. Algoritmo de detección de objetivos multi-resolución 14
2.4. ANN-1 y ANN-2 para el algoritmo de detección de objetivos multi-resolución. 15

3.1. Arquitectura Fermi de Nvidia. 16
3.2. Fermi Streaming Multiprocessor (SM) 17
3.3. Jerarquía de memoria de las GPU de Nvidia con arquitectura Fermi. 18

4.1. Nivel de paralelismo neuronal con un bloque de 256 threads. 24
4.2. Nivel de paralelismo sináptico con un bloque de 256 threads. 26
4.3. Algoritmo de detección multi-resolución: Mapeo de threads 29

5.1. Imágenes hiperespectrales utilizadas. 32
5.2. Algoritmo a nivel de píxel: Tiempos de ejecución y speedup. 35
5.3. Algoritmo a nivel de píxel: Dependencia con el tamaño de bloque. 38
5.4. Algoritmo a nivel de píxel: Dependencia con el tamaño L1 39
5.5. Algoritmo a nivel de píxel: Tiempo de computación y de acceso a memoria. 42
5.6. Fracción de tiempo total consumido en accesos y computación. 43

ÍNDICE DE TABLAS

3.1. Espacios de memoria para la arquitectura CUDA. 18

5.1. Algoritmo de detección a nivel de píxel: Tiempos de ejecución. 34
5.2. Algoritmo de detección multi-resolución:Tiempos de ejecución y speedup 36
5.3. Tiempos de transferencia CPU-GPU de las imágenes. 37
5.4. Algoritmo de detección a nivel de píxel: Ocupancia de GPU y bloques por SM 40
5.5. Algoritmo de detección a nivel de píxel: Ancho de banda y GFLOPS. 43

Capítulo 1

Introducción

1.1. Imágenes hiperespectrales: definición y aplicaciones.

Una imagen hiperespectral es una imagen en la que para cada píxel se tiene un conjunto de m valores, los cuales se corresponden con las componentes espectrales para distintas longitudes de onda, tal y como se representa en la figura 1.1, donde cada uno de los m planos en el eje z representan a una lámina espectral de la imagen, es decir, la imagen para una determinada longitud de onda.

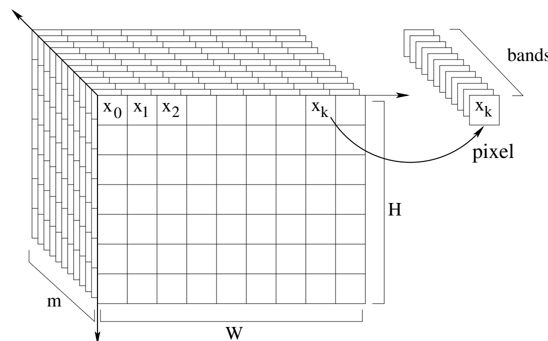


Figura 1.1: Imagen hiperespectral.

El término fue acuñado en el campo de la visualización mediante satélite [1]. Sin embargo, el creciente uso de sensores hiperespectrales, así como la aparición de cámaras hiperespectrales más asequibles, están haciendo que la investigación en este campo se extienda a más áreas y cobre un mayor interés, tanto en la búsqueda de nuevas aplicaciones, como en lo referente a los nuevos problemas que surgen debido a la creciente cantidad de datos hiperespectrales a tratar.

Las aplicaciones actuales de los sensores hiperespectrales van desde la convencional monitorización remota de detección y exploración del paisaje [2–5], hasta el análisis médico de laboratorio [6], pasando por el análisis de la calidad de ciertos alimentos [7], autenticación de obras de arte [8],

operaciones de búsqueda y rescate [9], y aplicaciones médicas [10] entre otras.

1.2. Procesado de imágenes hiperespectrales.

Uno de los aspectos más importantes para conseguir un mayor y mejor aprovechamiento de las imágenes hiperespectrales es poder operar y procesar en tiempo real las enormes cantidades de datos que producen los sensores. Este procesamiento en tiempo real de las imágenes hiperespectrales se ha llevado a cabo mediante implementaciones paralelas sobre clústers y redes de estaciones de trabajo, con el fin de repartir el elevado coste computacional entre diferentes unidades de procesado [11, 12]. Sin embargo, el principal problema de los clúster y de las estaciones de trabajo es que son sistemas generalmente caros, voluminosos y difíciles de adaptar a bordo de una unidad de captura hiperespectral. En consecuencia, se hace necesaria una implementación mediante componentes de bajo peso y de bajo consumo de energía y con tiempos de ejecución muy bajos, para lograr un análisis en tiempo real viable y satisfactorio.

Hoy en día las GPUs *(Graphics Processing Units)* son procesadores multi-núcleo de alto rendimiento, potentes y baratos, que pueden utilizarse para acelerar no sólo las clásicas aplicaciones gráficas, sino también un conjunto muy amplio y diverso de aplicaciones [13]. La razón de esto es que en los últimos años las GPU han evolucionado de ser aceleradores específicos para gráficos a ser procesadores vectoriales programables, con una capacidad de cómputo muy superior a la de las actuales CPUs multi-núcleo [14]. Recientemente, además, se ha desarrollado la arquitectura CUDA *(Compute Unified Device Architecture)* para las GPUs de Nvidia. La ventaja de CUDA es que ofrece la posibilidad de realizar implementaciones de aplicaciones de propósito general sobre las GPUs, mediante el uso de un lenguaje de programación que extiende C/C++, gracias a lo cual se hace posible el aprovechamiento del enorme nivel de paralelismo que las GPU ofrecen [15]. Sin embargo, las implementaciones directas de los algoritmos paralelos sobre GPUs ofrecen rendimientos muy irregulares, ya que éstos dependen fuertemente de aspectos como el patrón de acceso a memoria, o el balanceo entre comunicación y computación principalmente, lo que hace que se vuelva imprescindible llevar a cabo una adaptación de cada algoritmo a las características específicas de las GPUs, realizando las optimizaciones que sean necesarias sobre el patrón de acceso a memoria y mejorando la ratio computación/comunicación [13].

Afortunadamente, los algoritmos de procesamiento de imagen son, en general, bueno candidatos para la implementación sobre un modelo de programación SIMD *(Single Instruction, Multiple Data)* como el de las GPUs. Esto es así porque todos estos algoritmos requieren, habitualmente, un número grande de computaciones repetitivas sobre conjuntos disjuntos de datos, porque los accesos

presentan una elevada localidad, y porque el número de computaciones por dato transferido es alto (alta intensidad aritmética).

1.3. Detección de objetivos mediante ANNs: objetivo del trabajo.

Un problema relevante para el procesamiento hiperespectral es la detección automática de objetivos [16]. En los últimos años se han desarrollado varios algoritmos que realizan la detección automática de objetivos en imágenes hiperespectrales para diferentes resoluciones, yendo desde la detección de objetivos que ocupan varios píxeles, hasta la detección de varios objetivos dentro de un mismo píxel [17]. Se han desarrollado diferentes implementaciones sobre clústers y FPGAs para estos algoritmos de detección, y se han comparado sus rendimientos con los equivalentes obtenidos para implementaciones sobre GPUs [11, 12] . Los resultados demuestran claramente los beneficios que supone una implementación eficiente de estos algoritmos sobre la GPU [18].

El principal objetivo de este trabajo es continuar con esta línea de trabajo, explorando implementaciones eficientes de los algoritmos de detección de objetivos para diferentes resoluciones, mediante procesamiento sobre GPUs. La idea del trabajo es programar sobre GPU algoritmos de detección desarrollados a través de la colaboración con investigadores del grupo GII de la UDC, diseñados específicamente para que se adapten al hardware y al procesamiento en paralelo característicos de la GPU.

Los algoritmos de detección que se implementan en este trabajo están basados en la utilización de ANNs *(Artificial Neural Networks)*. Las ANNs han sido ampliamente utilizadas en trabajos previos de detección de objetivos en imágenes hiperespectrales [19]. Esto es debido, principalmente, a que son potentes herramientas para tareas que implican reconocimiento de patrones, dado su carácter no-lineal, y dado el hecho de no necesitar realizar ningún tipo de asunción sobre la distribución de los objetivos en la imagen [20]. Las ANNs utilizadas en este trabajo han sido diseñadas por el grupo GII de la UDC, quienes también han llevado a cabo la fase de entrenamiento inicial. La complejidad de esta fase de entrenamiento de las ANNs (tiempo, selección de la arquitectura de la red, selección de muestras, etc.) depende del tipo de problema que se pretende modelar, y resulta clave a la hora de obtener una buena eficiencia del modelo.

Los algoritmos desarrollados utilizan para la detección de los objetivos conjuntos de ANNs que se aplican sobre la imagen hiperespectral de manera independiente, unos a nivel de píxel (vector de componentes espectrales), y otros a nivel de ventana (cubos hiperespectrales de información), con lo que el beneficio principal es que no hay dependencias de escritura entre los datos en el

cálculo de las diferentes ANNs, lo cual se adapta perfectamente al paradigma SIMD explotado por las GPUs, sin embargo, la ratio computación/comunicación es baja, lo cual exige cuidado en el diseño e implementación de los algoritmos. En este aspecto existen en la bibliografía un gran número de publicaciones sobre las posibilidades de implementación de las ANNs sobre GPU. Una de las aproximaciones empleadas en algunos trabajos, por ejemplo, se basa en la utilización de las funciones de la librería de álgebra matricial (sumas, productos, trasposiciones, etc.) para implementar las ANNs de manera eficiente sobre GPU [21, 22]. Por otra parte, hay además un gran número de optimizaciones que se pueden realizar en la implementación de las ANNs sobre GPU [23]. Estas optimizaciones abarcan desde la modificación de los datos y de los patrones de acceso para obtener coalescencia en los accesos a memoria global, hasta la utilización de memoria compartida, memoria de constantes y de texturas, pasando por técnicas para la maximización de la ocupancia de la GPU, o reducción de ciclos de computación mediante el uso de funciones matemáticas rápidas, minimización de divergencia entre threads, manejo efectivo de conexiones entrada-salida de las neuronas, etc. También son exploradas en detalle las distintas estrategias de paralelismo que ofrecen las ANNs, considerando el paralelismo a nivel de neurona, a nivel sináptico, o a nivel neuronal-sináptico [24]. Los algoritmos desarrollados en este trabajo han sido diseñados para explotar estas diferentes estrategias de paralelización, y en ellos se ha tenido en cuenta un gran número de las optimizaciones habituales en GPGPU.

Capítulo 2

Algoritmos para la detección de objetivos.

2.1. Introducción y consideraciones generales.

Esta sección describe dos algoritmos automáticos de detección de objetivos en imágenes hiperespectrales, los cuales han sido desarrollados para su implementación en GPU. Estos algoritmos se han denominado, *algoritmo de detección de objetivos a nivel de píxel* y *algoritmo de detección de objetivos multi-resolución*, y ambos aprovechan el uso de distintas ANNs (*Artificial Neural Networks*) como herramientas de identificación.

Hay que tener en cuenta que dependiendo de la resolución de la imagen, un objetivo puede estar representado desde por un único píxel, hasta por un conjunto de ellos. En este aspecto, cada uno de los algoritmos sigue procedimientos distintos para la detección de los objetivos. Por un lado, el algoritmo de detección de objetivos a nivel de píxel basa su búsqueda en la exploración píxel por píxel de la imagen, detectando si cada píxel contiene un objetivo, o una parte de él. Mientras que, por otro lado, el algoritmo de detección de objetivos multi-resolución basa su búsqueda en la exploración de ventanas de la imagen (volúmenes hiperespectrales si consideramos que tenemos una imagen para cada una de las longitudes de onda consideradas) cada vez más pequeñas, de tal forma que la escala es ajustada al tamaño de cada objetivo, acotando el área que éste ocupa.

Es importante señalar que en ambos casos las ANNs utilizadas deben ser entrenadas en un proceso previo. Este entrenamiento, que es específico para una clase de objetivos a buscar, es esencial para fijar los parámetros de las redes de tal forma que la respuesta de éstas tenga relación con la probabilidad de que las entradas analizadas (píxeles de la imagen, o ventanas de ésta) contengan realmente los objetivos buscados.

2.2. Algoritmo de detección de objetivos a nivel de píxel.

Este algoritmo realiza una inspección de cada uno de los píxeles de la imagen de manera separada, generando, para cada uno de ellos, un *índice de detección* relacionado con la probabilidad de que el píxel considerado contenga el objetivo, o una parte de éste. Su funcionamiento se basa en la utilización de una RBFN *(Radial Basis Function Network)* [25], la cual se diseña para recibir como entrada las m componentes espectrales de un píxel, y para devolver como salida el índice de detección correspondiente a dicho píxel. Una vez calculados estos índices de detección para todos los píxeles de la imagen es necesario realizar un postprocesado de la información obtenida con el fin de conocer el tamaño y la forma de los objetivos detectados. En la figura 2.1 se puede ver un esquema del flujo de ejecución que sigue el algoritmo.

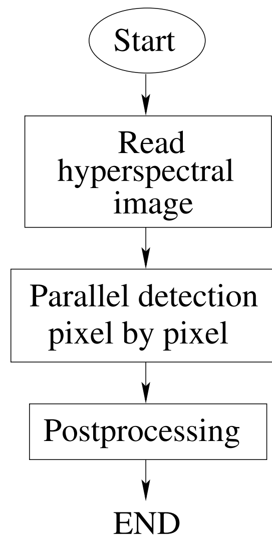


Figura 2.1: Algoritmo de detección de objetivos a nivel de píxel.

Las RBFNs utilizadas en el algoritmo de detección de objetivos a nivel de píxel están, como decíamos, diseñadas para admitir como entrada un píxel, es decir, un vector con las m componentes espectrales $(x_1, x_2, \cdots, x_m)$. Éstas tienen a su vez una capa intermedia, denominada cada oculta, con, en general, k neuronas, cada una de las cuales tienen asociados un conjunto de valores constantes, que son fijados en la fase de entrenamiento de la red. Para cada neurona i de la capa oculta, se tienen m centros $u_{1i}, u_{2i}, \ldots, u_{mi}$, y una desviación σ_i. Dados todos estos valores, la salida de la neurona i de la capa oculta, denominada h_i, se obtiene mediante la ecuación 2.1:

$$h_i = e^{-\frac{\sum_{j=1}^{m} (x_i - u_{ji})^2}{2\sigma_i^2}}, \quad \text{con} \quad i = 1, 2, \ldots, k. \tag{2.1}$$

Por último, la RBFN utilizada en este algoritmo consta de una única neurona en la capa de salida, que será la que nos dé el índice de detección relativo al píxel de entrada. Esta última neurona tiene asociados, de manera semejante a las neuronas de la capa oculta, un conjunto de valores constantes $w_1, w_2, \ldots, w_k$ denominados pesos, y un th llamado umbral, que también son fijados en la fase de entrenamiento, y que están involucrados en el cálculo del valor de salida de esta neurona, según la ecuación 2.2.

$$y = \sum_{i=1}^{k} w_i h_i + th \tag{2.2}$$

En la figura 2.2 puede verse un esquema de la arquitectura descrita de la RBFN, donde $x_1, x_2, \cdots, x_m$ denotan las m componentes espectrales del píxel bajo estudio.

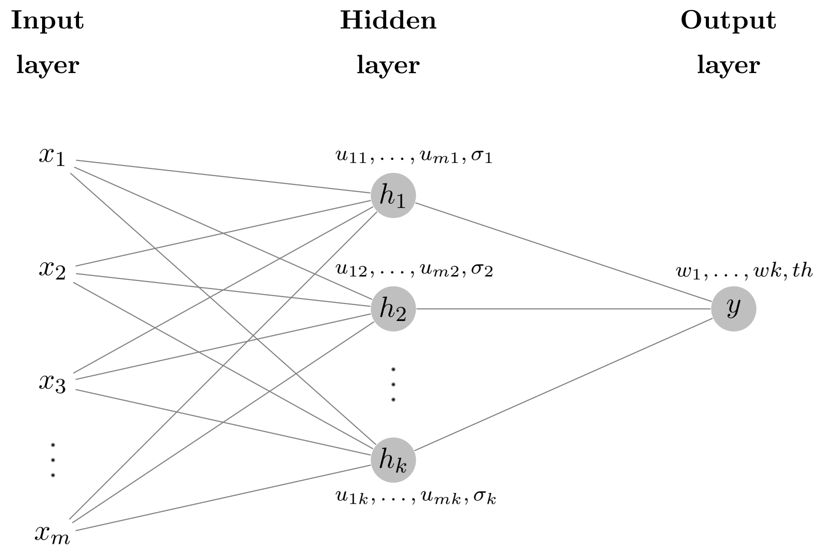


Figura 2.2: ANN utilizada por el algoritmo de detección de objetivos a nivel de píxel.

A la hora de computar la RBFN descrita solamente será necesario realizar las computaciones para las componentes x_i espectrales del píxel considerado ya que los centros, desviaciones estándar, pesos y valor umbral ya habrán sido determinados en la fase de entrenamiento y serán comunes para todos los niveles.

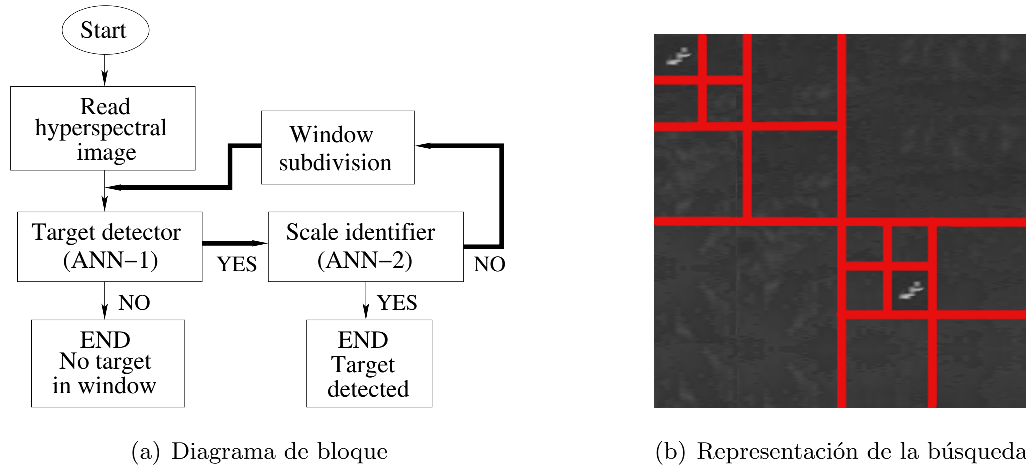
Es importante hacer notar que ambas operaciones 2.1 y 2.2 son operaciones de reducción, y que la aplicación de la RBFN sobre un píxel de la imagen no presenta interdependencias con la aplicación de la misma sobre otro píxel distinto, hecho que será esencial cuando abordemos la implementación paralela de este algoritmo.

2.3. Algoritmo de detección de objetivos multi-resolución.

Este algoritmo se centra en la búsqueda de objetivos independientemente de su escala, con lo que el postprocesado utilizado en el algoritmo anterior no es necesario aquí. Este algoritmo tiene un diseño jerárquico basado en la utilización de dos ANN del tipo FFBP *(Feed Forward Back Propagation)* [26]. La primera de ellas, a partir de ahora ANN-1, es la encargada de discernir si hay alguna cierta probabilidad de que algún objetivo esté contenido dentro de un área determinada de imagen. Y la segunda red, ANN-2, es la encargada de discernir si ese área se debe acotar mejor o si por el contrario, la escala de detección es ya la correcta.

El procedimiento seguido por el algoritmo consiste pues en ir acotando en ventanas (cubos hiperespectrales de información) de tamaño cada vez menor la presencia de los diferentes objetivos, comenzando con una única ventana del tamaño de la imagen original. Para esto, cada ventana es analizada primeramente por la red ANN-1. En el caso de que la respuesta de esta red sea negativa, esto significará que el objetivo no está dentro del área delimitada por la ventana, con lo que la ventana será desechada. Mientras que, si la respuesta de ANN-1 es positiva, entonces esto significará que dicha ventana contiene un objetivo (o varios) y pasará a ser analizada por la segunda red, la cual se encargará de discernir si la escala del objeto detectado es la adecuada. De esta forma, si al pasar la ventana como entrada de ANN-2 la red da una respuesta negativa, entonces significará que la escala de detección todavía no es la correcta, y por tanto la ventana se dividirá en cuatro sub-ventanas de menor tamaño, que entrarán a ser analizadas de nuevo por ANN-1, repitiéndose el proceso. Por contra, si ANN-2 da una respuesta positiva, esto querrá decir que el objetivo ha sido detectado en la escala correcta, y, por tanto, la ventana dejará de dividirse para pasar a ser almacenada como un objetivo detectado. Una vez finalizado el análisis de todas las ventanas el algoritmo se detiene, mostrando la localización y el tamaño de las ventanas con los objetivos detectados. En la figura 2.3(a) puede verse un diagrama de bloque del algoritmo que muestra este funcionamiento, y en la figura 2.3(b) puede verse un ejemplo de detección para una imagen con dos objetivos indicando las ventanas en que se han producido las búsquedas.

Para poder aplicar estas ANNs a una ventana es necesario calcular una desviación para cada una de las bandas espectrales de los píxeles que componen dicha ventana. Esta desviación para cada banda se calcula mediante la ecuación 2.3, en donde W y H son el número de píxeles de ancho y alto de la ventana respectivamente, i representa cada una de las bandas espectrales, k al índice de cada uno de los valores de la ventana para la banda i y $\overline{x_i}$ a la media de los valores en la ventana



(a) Diagrama de bloque (b) Representación de la búsqueda

Figura 2.3: Algoritmo de detección de objetivos multi-resolución.

para la banda i.

$$\sigma_i = \sqrt{\frac{\sum_{k=1}^{W.H}(x_{ki} - \overline{x_i})^2}{W.H}} \quad \text{con} \quad i = 1, 2, \ldots, m. \tag{2.3}$$

Además, es necesario también normalizar este vector de desviaciones $\sigma_1, \sigma_2, \ldots, \sigma_m$, al rango $[-1, 1]$, calculando $\sigma_1', \sigma_2', \ldots, \sigma_m'$ mediante:

$$\sigma_i' = 2 \cdot \frac{(\sigma_i - \sigma_{i_{min}})}{\sigma_{i_{max}} - \sigma_{i_{min}}} - 1 \quad \text{con} \quad i = 1, 2, \ldots, m. \tag{2.4}$$

en donde $\sigma_{i_{min}}$ y $\sigma_{i_{max}}$ son valores fijados en la fase de entrenamiento de la red.

De esta forma, la entrada de ambas ANN-1 y ANN-2, es este vector normalizado $\sigma_1', \sigma_2', \ldots, \sigma_m'$ calculado para cada una de las ventanas.

Como se puede observar en la figura 2.4, ambas ANN-1 y ANN-2 tienen un diseño parecido a la RBFN utilizada en el algoritmo anterior en cuanto al número de capas, y a tener una única neurona en la capa de salida. Sin embargo las ecuaciones para el cálculo de las respuestas de cada neurona son diferentes a las de la RBFN. Aquí, la salida de las neuronas de la capa oculta se computa a través de:

$$h_i = \sum_{j=1}^{m} \sigma_j' \cdot u_{ji} + b_i \quad \text{con} \quad i = 1, 2, \ldots, k \tag{2.5}$$

siendo u_{ij} los pesos, y b_i el bias, asociado a la neurona i. Ahora bien, en este tipo de redes, la salida de cada neurona se ve modificada además por una función de transferencia sigmoide. El valor de la salida tras pasar por esta función se representa por k_i y se obtiene mediante:

$$k_i = \frac{2}{1 + e^{-2 \cdot h_i}} - 1 \quad \text{con} \quad i = 1, 2, \ldots, k \tag{2.6}$$

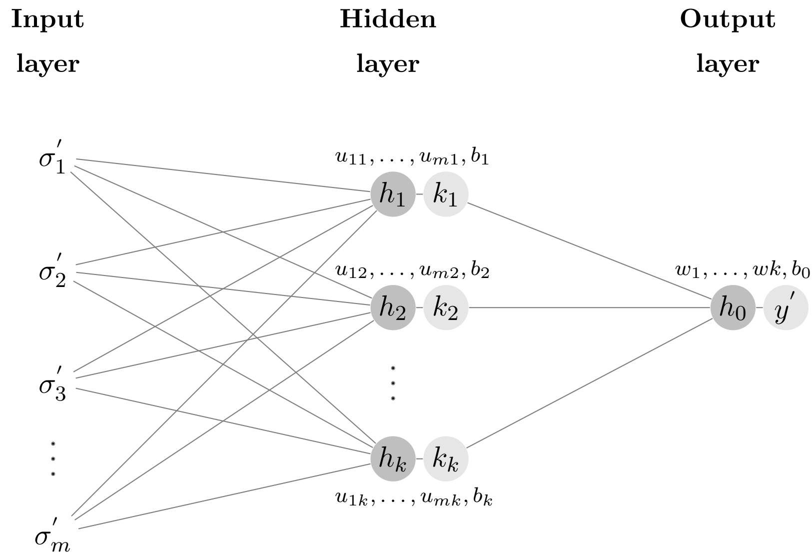


Figura 2.4: ANN-1 y ANN-2 para el algoritmo de detección de objetivos multi-resolución.

La neurona de la capa de salida se calcula utilizando ecuaciones similares a las anteriores:

$$h_0 = \sum_{j=1}^{k} k_j \cdot w_j + b_0 \tag{2.7}$$

$$y^{'} = \frac{2}{1 + e^{-2 \cdot h_0}} - 1 \tag{2.8}$$

siendo w_j los pesos, y b_0 el bias, asociados a esta última neurona.

Finalmente este valor de salida es necesario denormalizarlo, mediante:

$$y = (y_{n_{max}} - y_{n_{min}}) \frac{y^{'} + 1}{2} - 0{,}5 \tag{2.9}$$

Como se puede observar, en este algoritmo de detección de objetivos multi-resolución, resulta desconocido a priori el número de veces que se computan las ANNs para una imagen específica, en contra de lo que sucede con el algoritmo de detección de objetivos a nivel de píxel, en el que se ejecutan siempre tantas instancias de ANNs como píxeles tenga la imagen. Este algoritmo multi-resolución ahorra computaciones al precio de incrementar la complejidad de su flujo y, por tanto, la dificultad de alcanzar una implementación eficiente en GPU.

Capítulo 3

Visión general de la arquitectura CUDA.

En Noviembre de 2006 Nvidia introdujo CUDA (*Computer Unified Device Arquitecture*), una arquitectura para computación paralela de propósito general, que permite aprovechar la potencia de cómputo paralelo de las GPUs de Nvidia para resolver problemas computacionalmente costosos de una forma más eficiente que sobre CPU [15]. Desde su creación, Nvidia ha ido revisando y mejorando este modelo de arquitectura a lo largo de los últimos años, llegando hasta la actual arquitectura Fermi, cuyo esquema puede verse en la figura 3.1

Figura 3.1: Arquitectura Fermi de Nvidia.

La arquitectura CUDA está organizada en un conjunto de multiprocesadores o SMs (*Streaming Multiprocessors*), cada uno de los cuales contiene varios núcleos o SPs (*Streaming Processors*), siendo este número de núcleos por cada SM dependiente del modelo de tarjeta utilizado [15]. Cada uno de estos SMs es capaz de manejar cientos de *threads* para ejecutar porciones de código de

forma paralela, basadas en un modelo SIMD (*Single Instruction, Multiple Data*). Típicamente estas porciones de código, denominadas *kernels*, son ejecutadas de forma paralela sobre la GPU por miles de threads [27]. En la figura 3.1 puede verse una representación simplificada de los elementos que componen cada uno de los SM en la arquitectura Fermi.

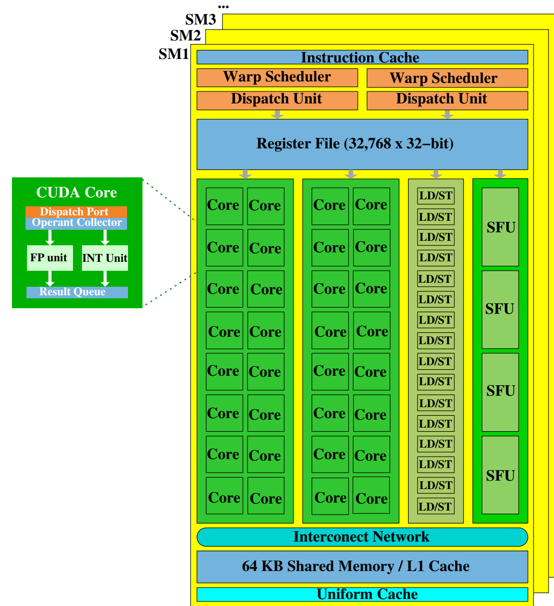


Figura 3.2: Fermi Streaming Multiprocessor (SM)

Desde el punto de vista del programador y compilador, los threads son manejados y agrupados en bloques de threads, y éstos a su vez organizados en *grids*. Los bloques de threads son conjuntos de threads que pueden colaborar entre ellos mediante el uso de barreras de sincronización y de una memoria compartida que tiene un tiempo de vida igual al tiempo en que el bloque de threads esté activo. Los *grids* son conjuntos de bloques de threads que ejecutan el mismo *kernel*, y que no poseen estos mecanismos de comunicación, es decir, distintos bloques no pueden sincronizarse entre ellos ni compartir datos en memoria compartida. Esta restricción resulta esencial a la hora de diseñar algoritmos que se adapten bien a la GPU, pues hay que tener en cuenta que dentro de una misma llamada de un kernel, un thread no podrá requerir datos que sean generados fuera de su propio bloque. Por otra parte, desde el punto de vista del hardware, es importante también tener presente que los threads son gestionados y ejecutados en grupos de 32 threads, denominados *warps*, los cuales constituyen el tamaño mínimo de procesado SIMD (*Single Instruction, Multiple Data*). En la arquitectura Fermi, además, cada SM viene equipado con dos *warp schedulers* lo que les permite ser capaces de ejecutar de manera concurrente dos warps diferentes [28]

En lo referente a la memoria de la arquitectura CUDA, físicamente está dividida en una

Memoria	Loc.	Cacheable	Acceso	Alcance	Tiempo vida
Registros	On-chip	N/A	R/W	Un Thread	Thread
Local	Off-chip	No	R/W	Un Thread	Thread
Compartida	On-chip	N/A	R/W	Threads de un bloque	Bloque
Global	Off-chip	Sí (Fermi)	R/W	Todos threads + Host	Aplicación
Texturas	Off-chip	Sí	R	Todos threads + Host	Aplicación
Constantes	Off-chip	Sí	R	Todos threads + Host	Aplicación

Tabla 3.1: Espacios de memoria para la arquitectura CUDA.

memoria DRAM común a todos los SMs, *off-chip*, y una memoria incluida dentro de cada SM, *on-chip*. Sin embargo, desde el punto de vista de la computación, la memoria está organizada en varios espacios de memoria diferentes con características de acceso y tiempo de vida distintos. En la tabla 3.1 pueden verse los distintos espacios de memoria existentes así como sus principales características, y en la figura 3.3 puede verse una representación de la jerarquía de niveles.

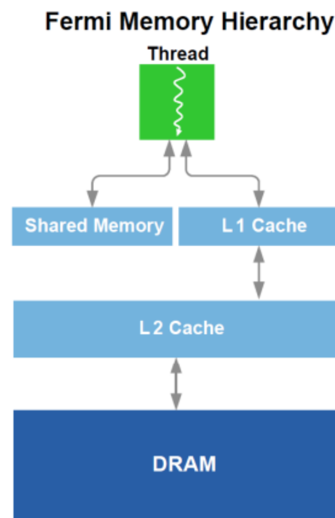


Figura 3.3: Jerarquía de memoria de las GPU de Nvidia con arquitectura Fermi.

Uno de los aspectos clave a la hora de reducir la latencia en los accesos a memoria global es la *coalescencia*, que es la capacidad que tiene la GPU de gestionar en una única transacción de memoria las peticiones a memoria global de un *half-warp* bajo ciertas restricciones en el patrón de acceso [29].

Por otro lado el hecho de que la memoria compartida esté integrada *on-chip* dentro de cada SM hace que tenga una latencia mucho menor que los espacios de memoria local y global. Además para

conseguir un ancho de banda alto la memoria compartida está dividida en módulos de memoria de igual tamaño denominados *bancos*, a los cuales se accede de forma simultánea. De esta forma cualquier petición de lectura o escritura realizada por un mismo *warp* sobre n direcciones que caigan en n bancos distintos son gestionadas de manera simultánea, mientras que si k peticiones recaen sobre el mismo banco entonces habrá conflicto en los accesos, y éstas k peticiones serán gestionadas de forma secuencial [15].

Los memorias de constantes y de texturas son memorias cacheables diseñadas para obtener un mayor rendimiento para determinados patrones de acceso. La memoria de constantes está optimizada para cuando los threads de un *half-warp* leen la misma dirección. La memoria de texturas está optimizada para explotar la localidad espacial 2D, y resulta más útil en patrones de acceso irregulares que la memoria global [15].

Es importante decir además que el sistema de memoria en la arquitectura Fermi cuenta con un completo sistema caché implementado a través de una memoria L1 por cada SM, y una memoria L2 de 768 KB unificada para todos ellos [28]. Además, el tamaño de la memoria caché es configurable junto con el de la memoria compartida de cada SM, pudiendo optar por dos tipos de configuraciones, 16/48 KB ó 48/16 KB (tamaño caché/tamaño memoria compartida), según se quiera optar por tener un mayor tamaño de caché L1 o un mayor tamaño de memoria compartida, en cada SM.

Capítulo 4
Implementación paralela.

4.1. Introducción y consideraciones generales.

En este capítulo profundizaremos en las diversas estrategias de paralelización sobre GPU del algoritmo de detección de objetivos a nivel de píxel, y del algoritmo de detección de objetivos multi-resolución.

Primeramente, hay que señalar que debido a la gran cantidad de datos que conforman una imagen hiperespectral ocurrirá que, en general, la memoria de la GPU no será capaz de almacenarlos todos simultáneamente, ya sea en memoria global, o en memoria compartida. Es por esto que se plantean dos posibles estrategias de fraccionamiento de la imagen [18], para poder moverla por trozos a memoria global, y para repartirla entre los bloques de threads en memoria compartida:

- Fraccionamiento por dominio espacial: La imagen se divide en cubos de píxeles, cada uno de ellos con todas sus bandas espectrales.
- Fraccionamiento por dominio espectral: La imagen se divide en láminas constituidas por una o varias bandas contiguas de todos los píxeles de la imagen.

En nuestro caso, el procesamiento de la hiperimagen es llevado a cabo mediante ANNs que, tal y como vimos en el capítulo 2, requieren como entrada todas las componentes espectrales de una área (píxel o ventana) de la imagen hiperespectral. De esta forma, el fraccionamiento que nos resultará más ventajoso, y que explotará mejor la organización de la memoria, es el fraccionamiento por dominio espacial. Así los píxeles o conjuntos de estos siempre serán movidos en memoria con la totalidad de sus bandas espectrales.

En los algoritmos desarrollados cuando un thread accede a una componente espectral de un píxel se requerirá acceder también a las otras componentes espectrales de ese mismo píxel, ya sea este

acceso realizado por otros threads del mismo warp o por ese mismo algunos ciclos más tarde. Por este motivo en este trabajo se decide almacenar la imagen hiperespectral con las bandas espectrales de un mismo pixel contiguas en memoria, aprovechando mejor así patrones de acceso coalescentes a memoria global, y el uso de la memoria caché de las nuevas tarjetas con arquitectura Fermi.

Por otra parte, los principales aspectos a tener en cuenta a lo hora de implementar las ANNs en GPU son básicamente la distribución de las computaciones entre los bloques de threads y el aprovechamiento del ancho de banda del dispositivo. En general las ANNs son aplicaciones que hacen mucho uso de memoria debido al elevado número de datos y constantes que utilizan, con lo que para este tipo de aplicaciones resulta esencial conseguir un buen mapeo de computaciones a threads, maximizando el reuso de datos en las operaciones realizadas en memoria compartida, y ocultando así, en lo posible, la latencia de los accesos.

Otra consideración general seguida en todos los *kernels* implementados es el uso de la memoria de constantes para almacenar los parámetros de las distintas ANNs. Esto se ha decidido así debido a que estos parámetros son constantes a lo largo de toda la ejecución de la aplicación, y que además, frecuentemente, un conjunto grande de threads debe acceder al mismo dato en el mismo ciclo, acceso que está optimizado para este tipo de memoria. La memoria de texturas también podría ser utilizada debido a la localidad espacial 2D presente en los accesos a los diferentes píxeles de la imagen, sin embargo en este tipo particular de problema se consigue un mejor ancho de banda utilizando la memoria global cacheable de las actuales Fermi [30], dado que la mayoría de los accesos gozarán de coalescencia. Finalmente, es importante destacar que para la implementación de las ANNs se ha utilizado precisión simple debido a que, al tener las ANNs utilizadas solamente dos capas, los resultados finales no se ven afectados por la acumulación de errores de truncamiento.

4.2. Algoritmo de detección de objetivos a nivel de píxel.

Como ya se vio en el capítulo 2 este algoritmo realiza una detección de objetivos mediante una búsqueda píxel a píxel en la imagen. Por ello, en este tipo de detección, cada píxel de la imagen hiperespectral (vector de datos que consiste en todas las componentes espectrales de ese píxel) es aplicado como entrada de la ANN, lo que implica que se tenga que realizar el cómputo de un gran número de instancias ANN sobre la GPU (tantas como píxeles tenga la imagen). Sin embargo, vimos en la sección 2.2 que los pasos para la detección del objetivo, o una parte de éste, sobre un píxel son totalmente independientes de la detección sobre otro píxel, con lo que esta búsqueda puede realizarse totalmente en paralelo.

Además existe otro grado de paralelismo dentro de las operaciones que realizan las RBFN (ecuaciones 2.1 y 2.2), ya que éstas son, básicamente, operaciones de reducción. Estas operaciones de reducción pueden ser implementadas de una forma optimizada usando la estrategia de reducción paralela descrita en los tutoriales de Nvidia [31], en la que se hace uso de técnicas como direccionamiento secuencial, desenrollamiento de bucles, etc. y mediante técnicas adicionales como *register packing* [32] en el que cada thread realiza una mayor carga computacional aprovechando mejor el uso de registros e incrementando por tanto la eficiencia. Una alternativa a estas técnicas es organizar las operaciones que se deben realizar en una ANN como producto de matrices, y utilizar entonces las librerías disponibles en CUDA de álgebra matricial [33]. En este trabajo se ha probado esta última opción, aunque se ha terminado optando por la primera de ellas por ofrecer mayor potencial de optimización.

La manera de proyectar sobre la GPU el cálculo de un número tan elevado de ANNs con el grado de paralelismo descrito debe estudiarse con atención. Las alternativas de mapeo de las ANNs sobre los bloques de threads son básicamente dos: paralelismo a nivel neuronal y paralelismo a nivel sináptico [24].

4.2.1. Paralelismo a nivel neuronal.

Esta aproximación recibe su nombre del hecho de que en ella cada uno de los threads de un bloque computa por si solo la salida de una neurona de una ANN. Así, las diferentes sinapsis de cada neurona son computadas de manera secuencial por cada thread. Este *kernel* utiliza para computar una instancia de ANN tantos threads como neuronas haya en la capa oculta.

El *kernel* asigna a cada uno de los threads de un bloque el cómputo de la salida de una neurona de la capa oculta, esto es, el cálculo de la expresión 2.1. Y luego, encarga a uno de ellos el cómputo de la neurona de la capa de salida, esto es, el cálculo de la expresión 2.2.

Para ello, primeramente cada uno de los threads va leyendo secuencialmente cada una de las componentes espectrales x_i del píxel correspondiente desde memoria global, junto con los pesos u_{ji} asociados a la neurona que está computando desde memoria de constantes, y los va operando y acumulando en memoria compartida, calculando primeramente el numerador de la expresión 2.1, y luego la expresión completa (trayendo desde memoria de constantes también el valor σ_i). Una vez hecho esto los threads son sincronizados, y uno de ellos se encarga entonces de computar la neurona de la capa de salida. Para ello, éste lee de memoria compartida los resultados de las neuronas de la capa oculta ya calculados, junto con los parámetros w_i y th de la neurona desde memoria de

constantes, y los opera según 2.2. Hecho esto, escribe el valor resultante en la correspondiente posición de memoria global. Este procedimiento puede observarse mejor en el pseudocódigo 4.1

Además, para conseguir un mejor rendimiento, el *kernel* está preparado para computar diferentes instancias de ANN dentro de un mismo bloque, según se elija el número de threads del mismo. De esta forma, como cada thread se asigna a una neurona de la capa oculta, si tenemos por ejemplo una RBFN con 64 entradas, 4 neuronas en la capa oculta, y 1 neurona en la capa de salida, como ocurre en nuestro caso, entonces cada bloque será capaz de computar en paralelo $B/4$ instancias de ANN, donde B es el tamaño del bloque. Así si $B = 256$, entonces 64 ANNs serán computadas por cada bloque, haciendo un reparto de los threads tal y como se representa en la figura 4.1.

```
SHARED[Th(x,y)] = 0; {//'Th(x,y)' es el identificador único de thread dentro del bloque}
for all i en x_i^y do {//x_i^y es la componente espectral "i"de una ANN "y"}
  SHARED[Th(x,y)] += (x_i^y − u_xi)^2;
end for
SHARED[Th(x,y)] /= σ_x;
SHARED[Th(x,y)] = exp(SHARED[Th(x,y)]);
SHARED[Th(x,y)] *= w_x;
SyncThreads();
if x == 0 then {Un único thread por cada ANN se encarga de la neurona de salida}
  for j = 1 to blockIdx.x do
    SHARED[Th(0,y)] += SHARED[Th(j,y)];
  end for
  GLOBAL[y] = SHARED[Th(0,y)] + th;
end if
```

Pseudocódigo 4.1: **Paralelismo a nivel neuronal.** *Por simplicidad y por comprensión del kernel, este pseudocódigo hace referencia a las instrucciones ejecutadas por un único bloque de threads. En realidad, el kernel debe tener en cuenta también el identificador de bloque, para conseguir que cada thread acceda siempre a distintas* x_i^y *y escriba en posiciones diferentes de memoria compartida y global.*

4.2.2. Paralelismo a nivel sináptico.

Esta aproximación recibe su nombre del hecho de que en ella cada uno de los threads de un bloque computa la sinapsis de una neurona. Así, un conjunto de threads colaboran en el cómputo de la salida de cada neurona. Este *kernel* utiliza para computar una instancia ANN tantos threads como componentes espectrales de entrada tiene la ANN (64 en nuestro caso).

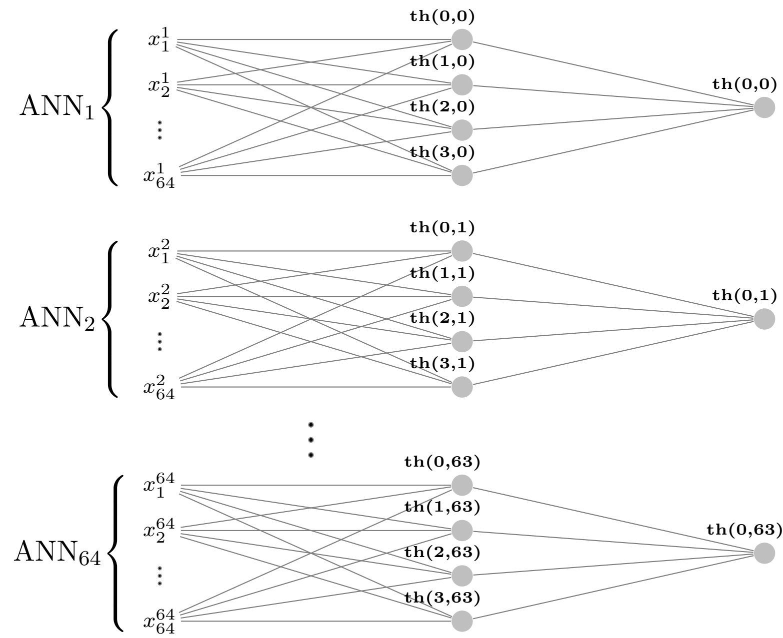


Figura 4.1: Nivel de paralelismo neuronal con un bloque de 256 threads para una ANN de 4 neuronas en la capa oculta. x_j^i denota la componente espectral j del píxel i. Encima de cada neurona pueden verse los threads encargados de calcular su salida, identificados mediante (blockIdx.x, blockIdx.y).

El *kernel* utiliza un conjunto de threads, tantos como entradas a la ANN haya, para que colaboren en el cálculo de la expresión 2.1 de cada una de las neuronas de la capa oculta. Estas neuronas de la capa oculta son computadas pues de forma secuencial, por un conjunto de threads que colaboran entre ellos. Una vez computadas todas las neuronas de la capa oculta, un único thread se encargará del cálculo de la expresión 2.2 de la neurona de la capa de salida.

Para ello, inicialmente, cada thread i lee de memoria global una única componente espectral x_i y la almacena en memoria compartida. Luego cada uno de ellos se ocupa de computar, y almacenar en un lugar distinto de la memoria compartida, el resultado de hacer $(x_i - u_{ji})^2$, utilizando el x_i almacenado en memoria compartida, y trayendo desde memoria de constantes los centros u_{ji} de la neurona j que están calculando. Después todos los threads cooperan mediante un algoritmo de reducción idéntico al utilizado en los tutoriales de Nvidia [31], computando el sumatorio $\sum(x_i - u_{ji})^2$, y dejando el resultado en otra posición de memoria compartida. Hecho esto, uno de los threads termina de realizar las operaciones de la expresión 2.1 para calcular h_i, trayendo de memoria de constantes la desviación σ_j y efectuando la exponencial negativa del cociente.

Una vez se tienen todos las salidas h_i de las neuronas de la capa oculta, uno de los threads se

encarga de calcular con ellas la expresión 2.2, trayendo de memoria de constantes los pesos w_i y el umbral th, y finalmente, almacena el resultado en memoria global. Este procedimiento puede observarse mejor en el pseudocódigo 4.2.

Además, para conseguir un mejor rendimiento, este *kernel* también está preparado para computar diferentes instancias de ANN dentro de un mismo bloque. De esta forma, por ejemplo, para una RBFN con 64 entradas, 4 neuronas en la capa oculta, y 1 neurona en la capa de salida, cada bloque es capaz de computar en paralelo $B/64$ ANNs, donde B es el tamaño del bloque. Así, si $B = 256$, entonces 4 instancias de ANN serán computadas por cada bloque, haciendo un reparto de los threads tal y como se representa en la figura 4.2.

1: SHARED[Th$_{(x,y)}$] = x_i^y; {*//x_i^y es la componente espectral "i"de una ANN "y", Th$_{(x,y)}$ es el identificador único de thread dentro del bloque*}
2: **for all** j en Neuronas capa oculta **do**
3: SHARED'[Th$_{(x,y)}$] = (SHARED[Th$_{(x,y)}$] $- u_{jx})^2$
4: SyncThreads();
5: 'Alg. Reducción' ⇒ SHARED'[Th$_{(0,y)}$] = $\sum_x$(SHARED'[Th$_{(x,y)}$])
6: **if** $x == 0$ **then**
7: SHARED'[Th$_{(0,y)}$] /= σ_x
8: SHARED"[Th$_{(0,y)} + j$] = exp(SHARED'[Th$_{(0,y)}$])
9: **end if**
10: **end for**
11: **if** $x == 0$ **then** {*Un único thread por cada ANN se encarga de la neurona de salida*}
12: SHARED"[Th$_{(0,y)}$] *= w_0
13: **for** $j = 1$ to blockIdx.x **do**
14: SHARED"[Th$_{(0,y)}$] += SHARED"[Th$_{(0,y)} + j$] $* w_j$
15: **end for**
16: GLOBAL[y] = SHARED"[Th$_{(0,y)}$] + th;
17: **end if**

Pseudocódigo 4.2: **Paralelismo a nivel sináptico.** *Por simplicidad y por comprensión del kernel, se omite el algoritmo de reducción, así como únicamente se hace referencia a las instrucciones ejecutadas por un único bloque de threads, ya que en realidad, el kernel debe tener en cuenta también el identificador de bloque para conseguir que cada thread acceda siempre a distintas x_i^y y escriba en posiciones diferentes de memoria compartida y global.*

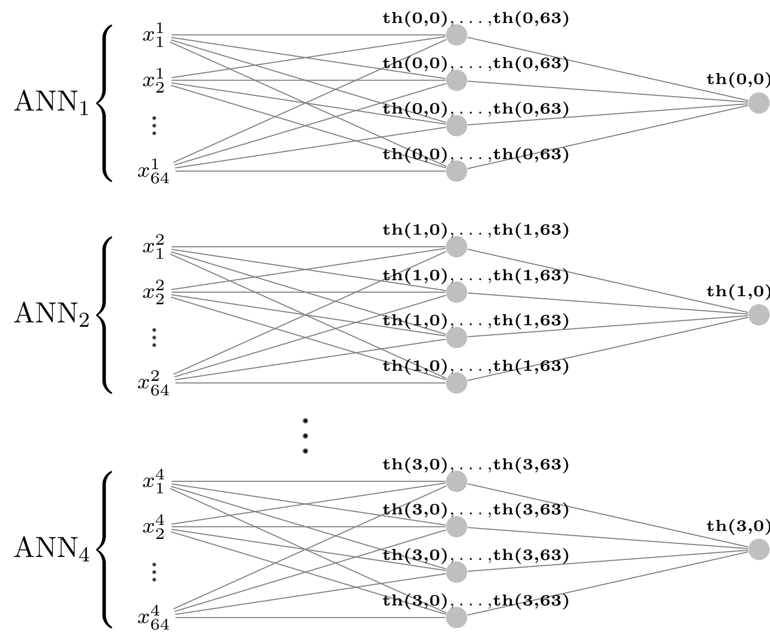


Figura 4.2: Nivel de paralelismo sináptico con un bloque de 256 threads para una ANN de 4 neuronas en la capa oculta. Encima de cada neurona pueden verse los threads que colaboran para calcular su salida, identificados mediante (blockIdx.x, blockIdx.y).

4.3. Algoritmo de detección de objetivos multi-resolución.

Como ya se vio en el capítulo 2 este algoritmo realiza una detección de objetivos mediante una división iterativa de ventanas de la imagen. Tal como se explicó en la sección 2.3 para una ventana dada, el camino a tomar por el algoritmo depende del resultado de dos redes ANN, las cuales se encargan de decidir si esta ventana es desechada, guardada, o si se sigue dividiendo en sub-ventanas que deberán, a su vez, analizarse.

Entrando más en detalle, el algoritmo lo que hace es comenzar analizando con una de las redes, ANN-1, una única ventana, que se corresponde con la imagen completa. Como resultado de este análisis esta ventana será desechada o aceptada como contenedora de algún objetivo, según la respuesta de la red sea negativa o positiva, respectivamente. En el caso de que la respuesta sea positiva, la ventana pasará a ser entonces analizada por la segunda red, ANN-2, la cual decidirá si se ha detectado el objetivo en la escala correcta, o si por el contrario se debe buscar en tamaños de ventana más pequeños. Si esto último sucede, entonces la ventana inicial es dividida en cuatro sub-ventanas, las cuales serán procesadas de forma análoga. El proceso iterativo se detendrá cuando todas las ventanas sean analizadas, o cuando se alcance un umbral mínimo de búsqueda, que en este trabajo se ha fijado para ventanas de 8×8 píxeles y que viene determinado

por la fase de entrenamiento de las redes de detección ANN-1 y ANN-2 y que deberá ser menor si las imágenes están tomadas a mayor altura. Merece la pena observar que este procedimiento jerárquico de búsqueda es fuertemente secuencial en cuanto a que las ventanas de un nivel de división dado no pueden analizarse hasta haberse analizado las ventanas del nivel superior en tamaño, hecho que limitará el grado de paralelismo a la búsqueda paralela en ventanas de un mismo nivel.

Una de las peculiaridades de este algoritmo reside en el hecho de que la CPU debe controlar en cada iteración el número de ventanas que se tienen que analizar. Esto resulta indispensable, ya que es ella la que controla el flujo de la aplicación, así como su parada, y también la que determina el número de bloques con los que deben invocarse los distintos *kernels*. Esto hace necesario que las respuestas de las ANNs, computadas sobre GPU, deban ser escritas en memoria global y de ahí llevadas a la memoria de la CPU, para que ésta sea capaz de saber qué ventanas se desechan, cuales conformarán la siguiente iteración, o cuales se deberán almacenar como objetivos detectados.

En el algoritmo de detección de objetivos multi-resolución se utilizan los siguientes *kernels*:

1. Kernel de reducción inicial para bloques 8×8 píxeles.
2. Kernel de entrada ANN para cada ventana.
3. Kernel ANN-1 y ANN-2 que computa las redes neuronales a partir de sus entradas.

4.3.1. Kernel de reducción inicial para bloques 8×8 píxeles.

Como vimos en el capítulo 2 la expresión 2.3 nos da la entrada no normalizada de las ANNs. Observándola detenidamente vemos que esta expresión es costosa computacionalmente, ya que supone calcular, para cada ventana, la desviación de cada banda espectral respecto al valor medio, sobre todos los píxeles que componen cada ventana. Para evitar este cuello de botella se diseña en el algoritmo una etapa previa en la que se calculan unos parámetros que se pueden reutilizar sistemáticamente en el cálculo de esta expresión para todas las ventanas.

En esta etapa previa, la imagen hiperespectral es dividida en bloques de un tamaño fijado de 8×8 píxeles, los cuales son tratados por un *kernel* que se encarga de calcular y almacenar en memoria global dos parámetros para cada una de las bandas espectrales de cada uno de ellos. Estos parámetros son la suma de los 8×8 valores por banda de cada bloque ($\sum_k x_{ki}$), y la suma de los cuadrados de los mismos ($\sum_k x_{ki}^2$). Esto se hace debido a que la ecuación 2.3 puede ser desarrollada

de la forma indicada por 4.1:

$$\sum_{k=1}^{N}(x_{ki} - \overline{x_i})^2 = \sum_{k=1}^{N} x_{ki}^2 - \frac{1}{N}\left(\sum_{k=1}^{N} x_{ki}\right)^2. \quad (4.1)$$

Y así, para calcular los $\sigma_1, \sigma_2, \ldots, \sigma_m$ de una ventana dada, simplemente bastará considerar los bloques 8×8 que conforman dicha ventana, y utilizar sus parámetros almacenados para el cálculo. Por ejemplo, si una ventana está compuesta por cuatro bloques de 8×8 píxeles, podremos calcular la entrada de las ANN teniendo en cuenta que:

$$\sum_{k=1}^{N} x_{ki} = \sum_{k=1}^{N/4} x_{ki} + \cdots + \sum_{k=3N/4+1}^{N} x_{ki} \quad (4.2)$$

$$\sum_{k=1}^{N} x_{ki}^2 = \sum_{k=1}^{N/4} x_{ki}^2 + \cdots + \sum_{k=3N/4+1}^{N} x_{ki}^2 \quad (4.3)$$

Básicamente lo que tiene que hacer el *kernel* de reducción inicial es calcular para cada bloque de 8×8 píxeles y cada banda espectral la suma de todos los valores así como la suma de los cuadrados de dichos valores. De esta forma, se pueden realizar las sumas de cada una de las bandas de forma totalmente paralela, ya que no hay que dependencias entre ellas.

Dada la importancia de tener coalescencia en la lectura de datos desde memoria global, y dado que la imagen hiperespectral está almacenada con las bandas espectrales de un mismo píxel de manera contigüa en memoria, el *kernel* se diseña pensando en que los threads de un mismo warp accedan a bandas contigüas de un píxel concreto. De esta manera, threads contigüos accederán a datos contigüos, y la coalescencia estará asegurada incluso en dispositivos con capacidad de cálculo 1.0 [15]. Además los datos así leídos de memoria global, deberán irse sumando y almacenando haciendo uso de la memoria compartida para aumentar la eficiencia.

Una primera aproximación realizada en este trabajo para este *kernel* fue utilizar tantos threads como bandas espectrales (64), encargando a cada uno de ellos el cálculo del sumatorio y el sumatorio de los cuadrados, de los 8×8 valores de su banda asignada. Sin embargo, aunque esta aproximación es sencilla de implementar, no resulta ser la más efectiva, ya que cada thread realiza demasiadas operaciones de manera secuencial. Por tanto, la opción más adecuada es hacer que varios threads colaboren en el cálculo de los sumatorios para una misma banda. De esta forma, en el *kernel* desarrollado todos los threads van accediendo de forma coalescente a los datos de memoria global, y los van acumulando (las sumas, y sumas de cuadrados) en memoria compartida. Terminado esto, un único thread por cada banda se encarga de realizar las sumas finales de los resultados parciales almacenados en memoria compartida y de escribirlos en memoria global en la posición adecuada.

Más en concreto, para una imagen $1024 \times 1024 \times 64$ el *kernel* será llamado con 16384 bloques de 512 threads en el grid, cada uno de los cuales estará encargado de un cubo hiperespectral $8 \times 8 \times 64$. El tamaño del bloque de threads utilizado en la reducción es de 512, distribuidos de forma bidimensional en 64 threads para el eje x, y 8 threads en el eje y. Estos threads son mapeados al cubo $8 \times 8 \times 64$ de forma que threads con igual identificador x accedan siempre a la misma banda x de los diferentes píxeles. De esta forma, 8 threads por banda deben colaborar en la reducción final de sus 8×8 valores a través de memoria compartida. El encargado de hacer la suma final de los resultados parciales es un único thread (en particular el que tiene identificador $y = 0$). Un diagrama de este reparto puede verse en la figura 4.3a).

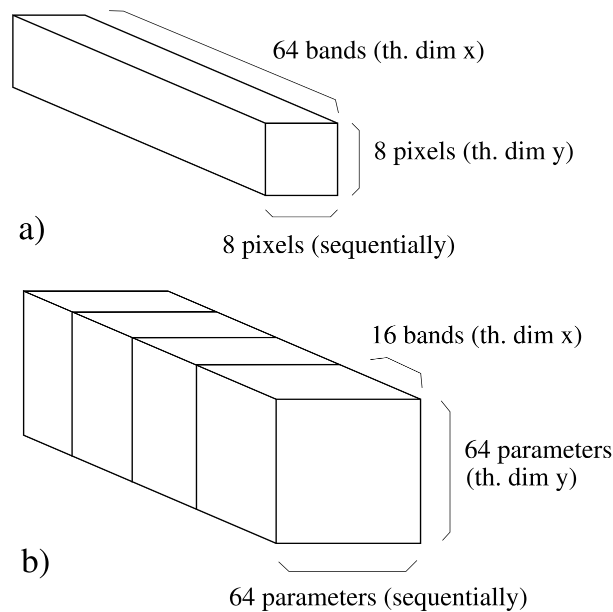


Figura 4.3: *a*) Mapeo de un bloque de 512 threads sobre un bloque de 8×8 píxeles realizado por el kernel de reducción inicial. *b*) Mapeo de un bloque de 1024 threads sobre un cubo de $64 \times 64 \times 64$ valores realizado por el kernel de entrada de ANNs.

4.3.2. Kernel de entrada ANN para cada ventana.

Este *kernel* es el encargado de calcular para cada ventana las entradas de las ANNs mediante el uso de los parámentros almacenados por el *kernel* anterior. Para ello, cada una de las ventanas es descompuesta en los bloques de 8×8 píxeles precalculados, y sobre los parámetros de éstos se realiza una operación de reducción, semejante a las ejemplificadas en las expresiones 4.2 y 4.3.

Estas operaciones de reducción se realizan de una forma parecida a la llevada a cabo por el *kernel* anterior, pues al igual que en aquél, aquí el cálculo del valor σ_j de la banda j no influye sobre el cálculo del valor σ_i de la banda i, con lo que cada banda puede computarse de forma

paralela, y también en bloques de threads distintos, ya que éstos no necesitarán colaboración entre ellos. Sin embargo, la principal diferencia estriba en que aquí no tenemos que operar sobre bloques de un tamaño fijo y pequeño de 8×8 valores por banda, sino sobre bloques de un tamaño mayor y variable. Serán variables en tanto en cuanto las ventanas consideradas se irán reduciendo en tamaño con cada iteración del algoritmo. Y serán mayores en tanto en cuanto se estarán considerando, por ejemplo, ventanas del orden de 512×512 píxeles, las cuales se calcularán a partir de 64×64 valores correspondientes a ventanas 8×8 calculadas en el kernel anterior.

Por estos motivos, el mapeo de los threads y de los bloques de threads en este *kernel* se realiza de una manera distinta al anterior, tal y como puede verse en la figura 4.3b) para el caso de considerar bloques de 1024 threads que cooperan para el cálculo de 16 bandas de la ventana considerada, que en este ejemplo es de 512×512 píxeles. Aquí un bloque de threads no computará las reducciones de todas las bandas, sino sólo las de un grupo contigüo de éstas, con lo que la entrada de una misma ANN será computada utilizando varios bloques de trheads distintos.

4.3.3. Kernel ANN-1 y ANN-2.

El cómputo de ambas ANN se implementa de una forma totalmente análoga a la detallada en el caso de la RBFN del algoritmo de detección a nivel de píxel, sin más que variar las fórmulas utilizadas allí por las descritas en 2.5, 2.6 y 2.7. Y al igual que para aquel algoritmo, aquí también se desarrollan las aproximaciones de paralelismo neuronal, y de paralelismo sináptico.

Capítulo 5
Análisis y resultados.

5.1. Procedimiento y equipo utilizado.

Las implementaciones propuestas han sido evaluadas sobre una Nvidia GPU de arquitectura Fermi, la GTX 580, la cual cuenta con 512 cores a 1.5GHz, agrupados en 16 SMs, con 32 cores por cada uno de ellos, alcanzando un rendimiento pico en simple precisión de 1581 GFLOPS. En cada instancia de tiempo, cada uno de estos SM es capaz de tener activos hasta 1536 threads simultáneamente (48 warps), permitiendo, gracias a los dos *warp schedulers* equipadas en las nuevas Fermi, que en cada ciclo se ejecuten de manera concurrente dos warps diferentes. Cada SM es capaz de tener activos 8 bloques de threads haciendo uso de un máximo de 32768 registros.

La GTX 580 está equipada sobre un interfaz PCI Express 2,0 $\times$ 16 que alcanza un pico de 20GB/s de ancho de banda en comunicación CPU-GPU. La GTX 580 tiene una memoria *off-chip* de 1.5GB, GDDR5, a 2004MHz, con un ancho de banda de 192.4GB/s, y una memoria *on-chip* de 64KB por cada SM. Los 64KB de memoria *on-chip* pueden ser repartidos como 48/16KB o 16/48KB entre memoria compartida y caché de nivel L1. Al ser de arquitectura Fermi la tarjeta cuenta con un sistema de memoria caché, con un nivel L1 por cada SM de tamaño configurable y un nivel unificado L2 de 768KB.

Para contrastar resultados los mismos algoritmos fueron adaptados e implementados de forma secuencial para su ejecución sobre CPU, optimizando la localidad mediante reorganizaciones de datos y reordenación en los accesos, y ocultando latencias en los accesos mediante solapamiento de fallos y desenrrollamiento de bucles, así como aprovechando los flags de optimización del compilador. En este trabajo dichas ejecuciones fueron realizadas con un Intel Quad Core Xeon E5440 a 2.83GHz, que cuenta con dos cachés L2 (una por cada par de cores) de 6MB, con 64B de tamaño de línea y de asociatividad 24, y con cachés L1 de instrucciones y datos por cada core de

(a) Imagen para reconocimiento de materiales.

(b) Imagen para búsqueda y rescate.

Figura 5.1: La imagen 5.1(a) es una fotografía realizada con una cámara fotografiando en el espectro visible del montaje utilizado para la toma de la imagen hiperespectral para el reconocimiento de materiales, mientras que la imagen 5.1(b) se corresponde a la información de una de las bandas de la imagen hiperespectral utilizada para el problema de búsqueda y rescate, editada para señalar los objetivos a detectar.

32KB, 64B de tamaño de línea y asociatividad 8.

En la toma de resultados se han utilizado dos tipos de imágenes hiperespectrales diferentes. Todas las imágenes han sido obtenidas por miembros del Grupo Integrado de Ingeniería de la UDC en laboratorio y utilizando el hiperespectrómetro desarrollado por dicho grupo. El primer tipo de imagen, representa un problema de reconocimiento de materiales, en concreto hojas de árboles. De este tipo de imágenes se han obtenido varias para entrenar los sistemas de detección. Para obtener resultados la imagen de hojas utilizada, en adelante *hojas.bin*, es una imagen $1024 \times 256 \times 64$, es decir, de tamaño 1024×256 y 64 bandas espectrales, que contiene 17 hojas. El segundo tipo de imágenes representan un problema de búsqueda y rescate, y son imágenes $1024 \times 1024 \times 64$ que simulan el mar, sobre el que se encuentran dispersados varios objetos como boyas o barcos, además de los objetivos a detectar. La imagen de este tipo utilizada para obtener los resultados es una imagen con 8 náufragos, aunque el entrenamiento se hizo con varias imágenes de igual tamaño y distinto número de náufragos. En la figura 5.1 pueden verse las dos imágenes utilizadas para la sección de resultados.

Por otra parte, todas las medidas de tiempo mostradas en este trabajo fueron obtenidas como promedio de 100 ejecuciones independientes, con el sistema completo dedicado de forma exclusiva, y haciendo uso de la función *gettimeofday()*. En la medida de tiempos de ejecución de los diferentes

kernels se ha tenido en cuenta la sincronización de los threads necesaria antes y después de cada llamada a la función de toma de tiempos. Además, las medidas del tiempo consumido por cada algoritmo realizadas se inician una vez la imagen hiperespectral se encuentra almacenada en la memoria global de la GPU, y terminan una vez los resultados de la detección son obtenidos y almacenados en CPU.

Respecto a la precisión es importante señalar que todos los algoritmos implementados detectan todas las instancias de los objetivos en cada una de las imágenes, y que las implementaciones sobre GPU arrojan exactamente los mismos resultados que las correspondientes sobre CPU.

5.2. Resultados.

En esta sección se detallan los tiempos de ejecución obtenidos mediante la implementación en GPU y mediante la implementación en CPU del algoritmo de detección de objetivos a nivel de píxel y del algoritmo de detección de objetivos multi-resolución, utilizando las imágenes *hojas.bin* de tamaño $1024\times256\times64$ y *naufragos.bin* de tamaño $1024\times1024\times64$. Los resultados más relevantes indicados en esta sección han sido aceptados para su publicación en el congreso [34].

5.2.1. Algoritmo de detección de objetivos a nivel de píxel.

Para este algoritmo la carga computacional consiste básicamente en aplicar la RBFN a cada píxel (vector de información hiperespectral) de la imagen. Tal y como vimos en el capítulo 4, se realizan dos tipos de aproximaciones paralelas: paralelismo neuronal y paralelismo sináptico.

Los tiempos de ejecución para ambas implementaciones sobre GPU, así como el speedup obtenido respecto a la ejecución secuencial del algoritmo en CPU pueden verse en la figura 5.2 para los dos tipos de imágenes utilizadas (imagen de hojas de $1024 \times 256 \times 64$ e imagen de náufragos de $1024 \times 1024 \times 64$). En ellas se han obtenido diferentes medidas variando el número de píxeles por *chunk*, esto es, el máximo número de píxeles que pueden ser procesados simultáneamente. Como es lógico, el incremento de este parámetro (eje X) no afecta al tiempo obtenido en CPU, que permanece constante ya que en CPU las computaciones serán siempre secuenciales, mientras que, para las implementaciones sobre GPU, el incremento del parámetro hace que la cantidad de trabajo que puede ser realizada en paralelo se incremente, con lo que en consecuencia, el tiempo de ejecución disminuye. Estas medidas fueron tomadas utilizando un tamaño de bloque de 256 threads en todos los casos.

(a) *hojas.bin* (1024 × 256 × 64)

Chunk	Neuronal (s)	Synaptic (s)	CPU (s)
2^4	0.2824	0.1593	0.3655
2^6	0.0739	0.0399	0.3655
2^8	0.0185	0.0168	0.3655
2^{10}	0.0047	0.0155	0.3655
2^{12}	0.0027	0.0150	0.3655
2^{14}	0.0024	0.0148	0.3655
2^{16}	0.0024	0.0149	0.3655
2^{18}	0.0024	-	0.3655

(b) *naufragos.bin* (1024 × 1024 × 64)

Chunk	Neuronal (s)	Synaptic (s)	CPU (s)
2^4	1.1297	0.6372	1.3018
2^6	0.2957	0.1594	1.3018
2^8	0.0740	0.0673	1.3018
2^{10}	0.0186	0.0619	1.3018
2^{12}	0.0108	0.0598	1.3018
2^{14}	0.0096	0.0592	1.3018
2^{16}	0.0094	0.0592	1.3018
2^{18}	0.0093	-	1.3018
2^{20}	0.0094	-	1.3018

Tabla 5.1: Tiempos de ejecución para el algoritmo de detección de objetivos a nivel de píxel para CPU y para las diferentes implementaciones GPU (paralelismo a nivel neuronal y sináptico), utilizando un tamaño de bloque de 256 threads.

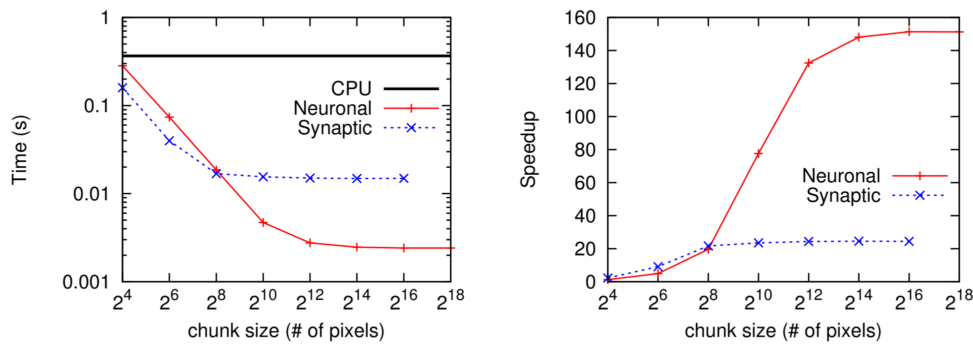
En ambas figuras se puede observar que la implementación basada en el paralelismo neuronal es la más eficiente, alcanzando un speedup máximo de 151.3x para 2^{18} píxeles por *chunk* en el caso de la imagen *hojas.bin*, y un 137.9x para 2^{20} píxeles por *chunk* en el caso de *naufragos.bin*, frente al paralelismo sináptico que alcanza un 24.5x para 2^{16} en *hojas.bin* y un 22.0x para 2^{16} en *naufragos.bin*. En la tabla 5.1 pueden verse de forma detallada los tiempos de ejecución obtenidos.

Las RBFNs utilizadas para ambas imágenes en este algoritmo tienen una entrada de 64 componentes, con 4 neuronas en la capa oculta y una neurona en la capa de salida. Para el caso del paralelismo neuronal cada thread computa la contribución de estas 64 bandas espectrales para una neurona de la capa oculta, con lo que son necesarios 4 threads para computar una instancia ANN. Para el caso del paralelismo sináptico cada thread computa una contribución sináptica, con lo que 64 threads son necesarios para computar una instancia ANN. Así, resulta claro que cada thread realiza más cálculos en el caso del paralelismo neuronal, lo cual explota mejor la arquitectura de la GPU, obteniendo mejores speedups.

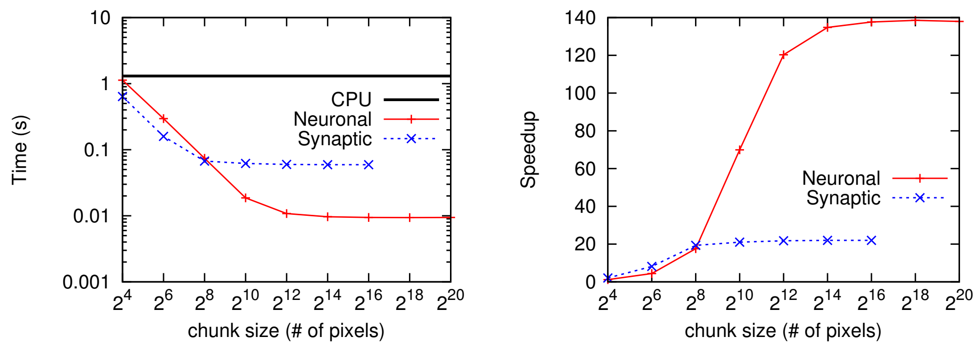
5.2.2. Algoritmo de detección de objetivos multi-resolución.

El algoritmo de detección de objetivos multi-resolución busca los objetivos independientemente de su escala, acotándolos de forma iterativa en ventanas cada vez menores de la imagen. En la tabla 5.2 pueden verse los tiempos de ejecución y speedups obtenidos para la imagen *naufragos.bin*.

Tal y como vimos en el capítulo 4, la implementación en GPU lleva a cabo una reducción



(a) Imagen hiperespectral 1024 × 256 × 64 (*hojas.bin*).



(b) Imagen hiperespectral 1024 × 1024 × 64 (*naufragos.bin*).

Figura 5.2: Tiempos de ejecución y speedup para los algoritmos de detección a nivel de píxel con 256 threads por bloque para las imágenes *hojas.bin* y *naufragos.bin*.

inicial de la imagen en bloques de 8 × 8 píxeles, almacenando en memoria compartida varios parámetros por cada banda espectral que serán luego aprovechados en el cálculo de la entrada de las ANNs. Los resultados muestran que esta etapa resulta muy eficiente, debido al gran número de operaciones de reducción a realizar, y debido al acertado reparto realizado de los threads para estas operaciones, en las que cada thread está encargado de reducir varios elementos, y no sólo 2 de ellos. El speedup alcanzado en esta etapa es de 65.4x.

Después de esta etapa comienza el proceso iterativo tal y como se detalla en el capítulo 4. Las ANNs utilizadas aquí admiten una entrada de 64 componentes espectrales, tienen 5 neuronas en la capa oculta, y una única neurona en la capa de salida, y son implementadas en GPU utilizando también las dos aproximaciones de paralelismo neuronal y sináptico, de una forma idéntica a la utilizada en el algoritmo de detección a nivel de píxel. En cada una de las iteraciones, las ANNs son aplicadas a cada una de las ventanas seleccionadas como posibles objetivos en la iteración anterior, pero dado que este número de ventanas es en general pequeño (no superan para la imagen considerada las 36 ventanas por iteración) las dos aproximaciones de paralelismo

(a) Paralelismo neuronal

	CPU (s)	GPU (s)	Speedup
Reducción inicial en bloques	0.1113	0.0017	65.4x
Detección ANN	0.0231	0.0010	22.8x
TOTAL	0.1344	0.0027	48.9x

(b) Paralelismo sináptico

	CPU (s)	GPU (s)	Speedup
Reducción inicial en bloques	0.1113	0.0017	65.4x
Detección ANN	0.0231	0.0009	25.6x
TOTAL	0.1344	0.0026	51.7x

Tabla 5.2: Tiempos de ejecución y speedup para las implementaciones CPU y GPU del algoritmo de detección de objetivos multi-resolución, para la imagen *naufragos.bin*.

de las ANNs, neuronal y sináptica, ofrecen speedups similares (pues trabajan en la zona con *chunk size* $< 2^6$). El mejor speedup obtenido en esta etapa lo ofrece el paralelismo sináptico, con un 25.6x, frente al 22.8x del paralelismo neuronal. La fila etiquetada como 'TOTAL' en la tabla 5.2 representa la suma de los tiempos de las etapas de reducción inicial en bloques y de computación ANN. En ella podemos apreciar que el mejor speedup obtenido por este algoritmo es de 51.7x para el caso de paralelismo sináptico, frente a los 48.9x del paralelismo neuronal.

Hay que hacer notar que aunque para este algoritmo se obtenga un speedup menor que para el algoritmo de detección a nivel de píxel (donde se alcanzan los 151.3x), los tiempos de detección para éste son, sin embargo, varios órdenes de magnitud más pequeños. Así, el algoritmo de detección de objetivos multi-resolución realiza, por ejemplo para la imagen *naufragos.bin*, la detección de todos los objetivos en un tiempo de 0.0026 segundos, mientras que el algoritmo de detección a nivel de píxel consume 0.0152 segundos. Por otra parte, el tiempo de computación empleado por este algoritmo de detección de objetivos multi-resolución dependerá, como es lógico, del número de objetivos a detectar en la imagen.

5.3. Análisis de rendimiento y discusión de resultados.

En esta sección se analiza el rendimiento obtenido mediante las implementaciones en GPU de los algoritmos de detección de objetivos a través de medidas realizadas sobre ancho de banda, Gflops, y ocupancia de la tarjeta. Además se muestran los tiempos de ejecución obtenidos para los diferentes algoritmos variando factores como el tamaño de bloque o la configuración del tamaño de memoria caché/memoria compartida.

Aunque no hemos encontrado ningún autor que previamente haya testeado algoritmos de búsqueda de objetivos a nivel de píxel basados en la detección con ANNs, podemos comparar estos resultados con los obtenidos para los algoritmos de detección estudiados en [18], donde los mejores resultados de speedup son de 70.1x para una imagen hiperespectral de 140MB. En nuestro caso el mejor resultado es de 151.3x para una imagen de 64MB (*hojas.bin*), y de 137.9x para una imagen de 256MB (*naufragos.bin*).

5.3.1. Transferencia CPU-GPU de las imágenes.

Como explicamos en la sección 5.1, en este trabajo las medidas del tiempo consumido por cada algoritmo se inician una vez la imagen hiperespectral se encuentra almacenada en la memoria global de la GPU, y terminan una vez los resultados de la detección son obtenidos y almacenados en CPU. Sin embargo, las medidas del tiempo necesario para transferir las imágenes utilizadas desde CPU a memoria global de GPU pueden verse en la tabla 5.3. El trasvase de estos datos se realiza una única vez al inicio de la aplicación, y mediante transferencias *page-locked data* que incrementan el ancho de banda efectivo [29]. Como se puede ver, el tiempo de transferencia supera en varios órdenes de magnitud a cualquiera de los tiempos de computación consumidos por los algoritmos de detección. Sin embargo, si este tiempo de transferencia es sumado al tiempo de computación, todavía se obtiene una pequeña aceleración respecto a CPU.

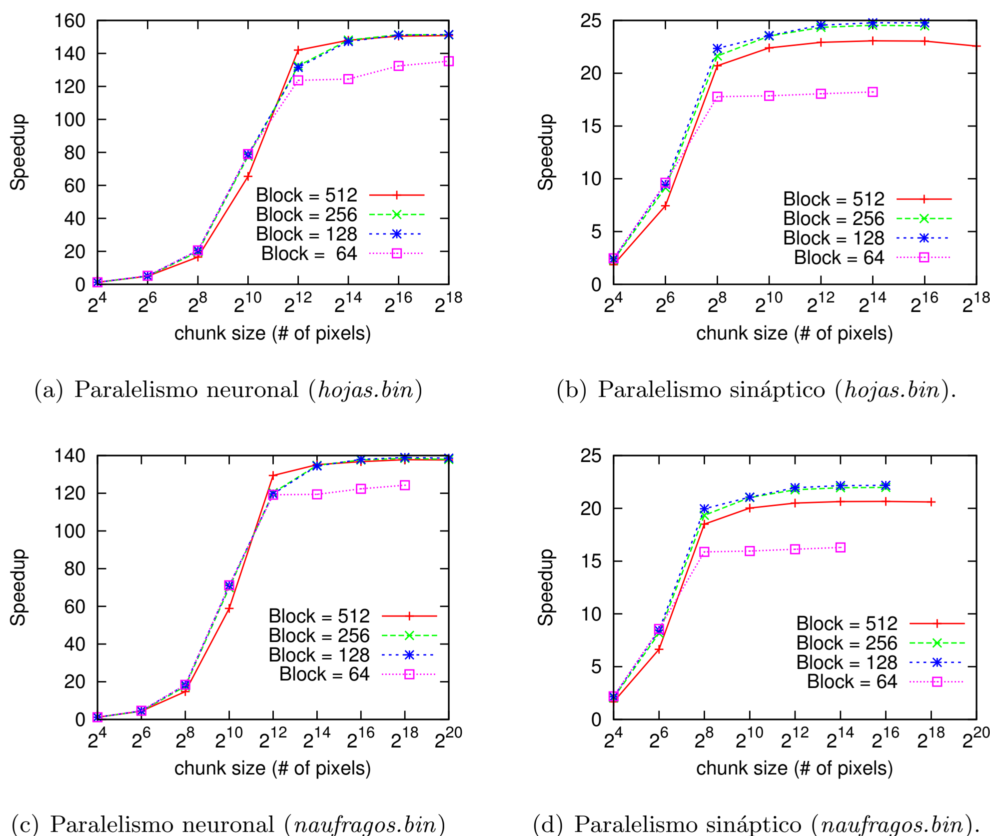
Imagen hiperespectral	Tamaño (MB)	Transferencia (s)	Ancho de banda(GB/s)
hojas.bin ($1024 \times 256 \times 64$)	64	0.0218	2.87
naufragos.bin ($1024 \times 1024 \times 64$)	256	0.0871	2.87

Tabla 5.3: Tiempos de transferencia CPU-GPU de las imágenes utilizadas, y ancho de banda alcanzado.

5.3.2. Dependencia con el tamaño de bloque.

Los tiempos de ejecución vistos hasta ahora se corresponden con ejecuciones en las que se utiliza un tamaño de bloque de 256 threads para el cómputo de las ANNs en ambos algoritmos de detección (a nivel de píxel y multi-resolución) y para ambas aproximaciones paralelas (neuronal y sináptica). Este tamaño de bloque ha sido escogido como óptimo en base a las medidas de rendimiento realizadas. En la figura 5.3 pueden verse los resultados obtenidos variando el tamaño de bloque para el algoritmo de detección a nivel de píxel con paralelismo neuronal y sináptico, para las dos imágenes *hojas.bin* y *naufragos.bin*.

Como vemos, en el caso del paralelismo sináptico los mejores rendimientos se obtienen para los tamaños de 128 y 256 threads, que muestran resultados casi idénticos, mientras que el rendimiento decae considerablemente para los casos de 64 y 512 threads. Por otro lado, en el caso del paralelismo neuronal los mejores rendimientos también se obtienen para los casos de 128 y 256 threads, aunque en algún caso puntual el tamaño de 512 llegue a superar a éstos ligeramente.



(a) Paralelismo neuronal (*hojas.bin*)

(b) Paralelismo sináptico (*hojas.bin*).

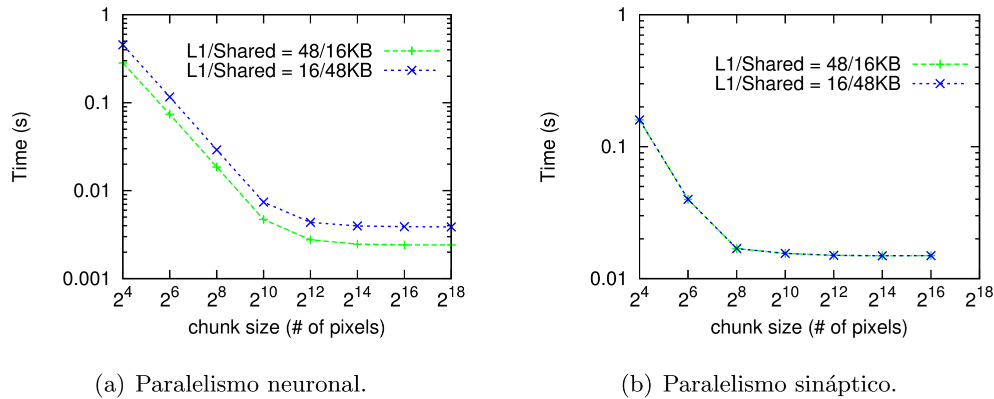
(c) Paralelismo neuronal (*naufragos.bin*)

(d) Paralelismo sináptico (*naufragos.bin*).

Figura 5.3: Dependencia con el tamaño del bloque de threads, para el algoritmo de detección a nivel de píxel, y para las dos aproximaciones de paralelismo, neuronal y sináptico.

5.3.3. Dependencia con el tamaño de la caché L1.

Como vimos en el capítulo 3 las nuevas GPUs de Nvidia, de arquitectura Fermi, cuentan con un completo sistema de memoria caché [28]. Este sistema está diseñado en dos niveles, un nivel L2 compartido por todos los SMs, y un nivel L1 propio de cada uno de ellos. Además, el tamaño de este nivel L1 es configurable en conjunción con el tamaño de la memoria compartida, pudiéndose elegir entre dos configuraciones, 48/16KB o 16/48KB (tamaño caché/tamaño memoria compartida), según se tenga preferencia por la caché, o por la memoria compartida. Debido a la importancia entre elegir una configuración u otra para el rendimiento de cada *kernel* se han realizado medidas para ver cómo afectan éstas a los tiempos de ejecución. En la figura 5.4



(a) Paralelismo neuronal.

(b) Paralelismo sináptico.

Figura 5.4: Dependencia con el tamaño de la caché L1, para el algoritmo de detección a nivel de píxel, y para las dos aproximaciones de paralelismo, neuronal y sináptico.

pueden verse como ejemplo los resultados para el algoritmo de detección a nivel de píxel, en las aproximaciones neuronal y sináptica, para la imagen *hojas.bin*.

Como vemos, la aproximación de paralelismo neuronal se ve fuertemente afectada por la elección de una u otra configuración, obteniéndose por ejemplo, para un tamaño de *chunk* de 2^{18}, un tiempo de 0.0038s para la configuración que da preferencia a la memoria compartida, y un tiempo de 0.0024s para la configuración que da preferencia a la caché L1, mejorándose por tanto el tiempo en un 63 %. La aproximación de paralelismo sináptico no muestra dependencias en el rendimiento entre la elección de una u otra configuración ya que, como veremos más adelante, la mayor parte de los accesos a memoria que realiza no son a memoria global (la que está cacheada) sino a memoria compartida. En base a estos resultados, las medidas del trabajo fueron tomadas para una configuración con caché L1 de 48 KB.

5.3.4. Aprovechamiento de los recursos de la GPU.

En este apartado estudiaremos el grado de aprovechamiento de los recursos que los *kernel* que computan ANNs está haciendo de la GPU. Para medir estos resultados se ha utilizado la herramienta *CUDA profiler*, que es capaz de ofrecer información sobre varios eventos hardware ocurridos durante la ejecución de los *kernels*. En particular, se han tomado datos de ocupancia y del número de bloques ejecutados en cada SM, para ejecuciones del algoritmo de detección a nivel de píxel, aproximaciones neuronal y sináptica, utilizando un tamaño de bloque de 256 threads. Los resultados obtenidos pueden observarse en la tabla 5.4.

La ocupancia es un parámetro que nos da información sobre el grado de aprovechamiento de

(a) Paralelismo neuronal

Chunk	Occupancy	Blocks per SM
2^4	0.042	1
2^6	0.167	1
2^8	0.667	1
2^{10}	1.000	1
2^{12}	1.000	4
2^{14}	1.000	16
2^{16}	1.000	64
2^{18}	1.000	256
2^{20}	1.000	1024

(b) Paralelismo sináptico.

Chunk	Occupancy	Blocks per SM
2^4	0.667	1
2^6	1.000	1
2^8	1.000	4
2^{10}	1.000	16
2^{12}	1.000	64
2^{14}	1.000	256
2^{16}	1.000	1024

Tabla 5.4: Ocupancia de GPU y bloques totales ejecutados por SM, para los kernels de detección a nivel de píxel, con un tamaño de bloque de 256 threads, para la imagen *naufragos.bin*.

los recursos de la GPU, de tal forma que una ocupancia de 1 indica que todos los SM de la tarjeta están siendo utilizados. Como vemos, para el paralelismo neuronal obtenemos una ocupancia máxima a partir de un *chunk* de 2^{10}. Dado que estamos utilizando un tamaño de bloque de 256 threads, en el paralelismo neuronal cada bloque de threads estará computando 64 ANNs (tal y como vimos en el capítulo 4), con lo que para computar $2^{10} = 1024$ instancias ANN en paralelo, el *kernel* será llamado con 16 bloques de 256 threads. Si atendemos ahora nuevamente a la tabla 5.4, vemos que estos bloques se están mapeando uno por cada SM, con lo que al contar la GTX 580 con 16 SMs obtendremos ya una ocupancia 1.

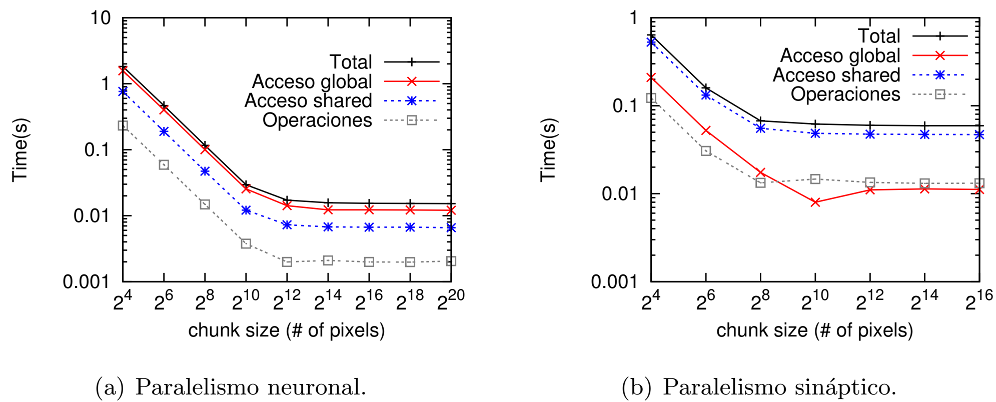
Sin embargo, sabemos que cada SM es capaz de mantener activos 1536 threads (48 warps), simultaneando su ejecución en grupos de 2 warps [28]. Ahora bien, para el paralelismo neuronal, para un chunk de 2^{10}, aunque estemos utilizando todos los SM de la tarjeta, cada uno de ellos tan solo está ejecutando 256 threads (8 warps), con lo que la GPU todavía no está aprovechada al máximo en cuanto a su capacidad de cómputo. El tamaño de *chunk* para el cual se comienzan a aprovechar todos los recursos de la GPU, para el paralelismo neuronal, resulta estar a partir de 2^{12}, pues para este caso cada SM ejecuta 4 bloques de 256 threads, es decir, 1024 threads, valor que ya se aproxima al límite de 1536 threads que pueden estar activos simultáneamente en cada SM (el valor límite se alcanzaría en realidad para un *chunk* entre 2^{13} y 2^{14}). Este hecho podría estar detrás del estancamiento que sufre la curva de paralelismo neuronal a partir del tamaño de *chunk* de 2^{12}, observado en la figura 5.2, a causa de la limitación del paralelismo impuesta por el número de recursos hardware. Un razonamiento análogo se sigue para el paralelismo sináptico, teniendo en cuenta que para este caso cada instancia ANN necesita 64 threads para ser computada, con lo que un bloque de 256 threads tan solo será capaz de computar 4 instancias.

5.3.5. Ancho de banda y rendimiento en GFLOPS.

En este apartado calcularemos el ancho de banda y el rendimiento de cómputo en GFLOPS alcanzado por los algoritmos de detección a nivel de píxel con paralelismo neuronal y sináptico. Para poder realizar este cálculo, se han realizado medidas aproximadas del tiempo que cada *kernel* consume en accesos a memoria global, accesos a memoria compartida, y en cálculos de punto flotante, y se ha contabilizado la cantidad de bytes movidos desde memoria global, y desde memoria compartida, así como también se contabilizan el número de operaciones en punto flotante realizadas por cada *kernel*.

Para aproximar las medidas de tiempo de acceso a memoria global se han modificado en los *kernels* todos los accesos a memoria global (lectura y escritura) por accesos a registros y se han medido los tiempos de ejecución de estos *kernels* modificados de manera similar a como se hace en [35]. El tiempo aproximado de acceso a memoria global se calcula entonces como la diferencia entre el tiempo de ejecución del *kernel* sin modificar y del *kernel* así modificado. Para el cálculo del tiempo consumido en accesos a memoria compartida se procede de una forma totalmente análoga, sustituyendo los accesos a memoria compartida por registros y contrastando medidas. Para el cálculo del tiempo consumido exclusivamente en computación, se sustituyen ambos accesos, tanto a memoria global como compartida, por accesos a registros y se mide el tiempo de ejecución, que ya se toma como el empleado en cálculos de punto flotante. En la figura 5.5 pueden verse representados los tiempos así calculados. Por ejemplo, para un *chunk* de 2^{16}, el *kernel* de paralelismo neuronal consume 0,015s, los accesos a memoria global 0,012s, los accesos a memoria compartida 0,006s, y la computación 0,002s. Mientras que para el mismo *chunk*, el *kernel* de paralelismo sináptico consume un tiempo total de 0,0592s, de los cuales 0,0112s son de acceso a memoria global, 0,0470s de acceso a memoria compartida, y 0,0131s de computación. La relación porcentual entre estos datos puede verse en la figura 5.6. Así, vemos que el *kernel* de paralelismo neuronal utiliza la mayor parte del tiempo en accesos a memoria global, mientras que el *kernel* de paralelismo sináptico lo hace en accesos a memoria compartida. Se hace notar que la suma de los tiempos parciales recogidos supera al tiempo total, hecho que ocurre debido a sobrecargas que no se están teniendo en cuenta. Sin embargo, el procedimiento es útil al menos de forma cualitativa para tener una idea de dónde consume más tiempo cada *kernel*. En la tabla 5.5 podemos ver los anchos de banda, y el rendimiento de computación alcanzados.

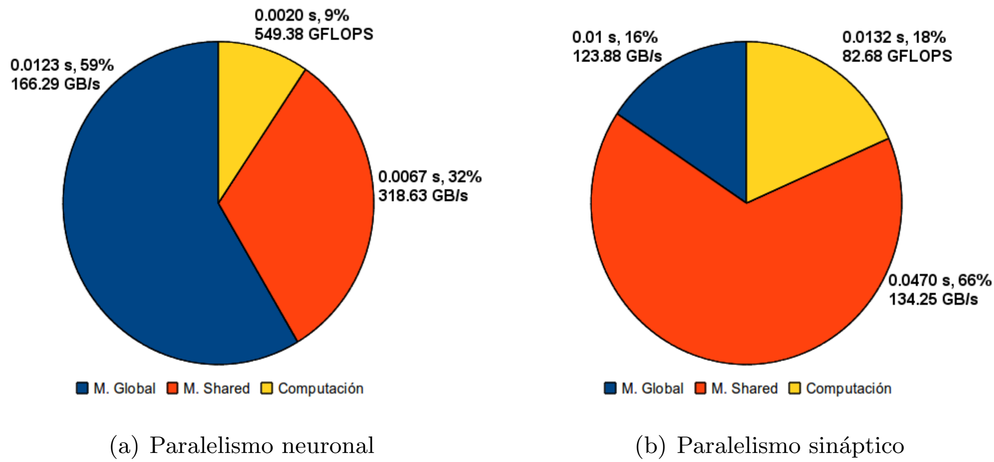
Estos valores deben ser tenidos en cuenta en conjunción con los valores pico de la GTX 580 vistos en la primera sección de este capítulo. Los mejores resultados de computación se obtienen para el paralelismo neuronal, alcanzando un porcentaje máximo de un 34.9 % de los 1581 GFLOPS



(a) Paralelismo neuronal. (b) Paralelismo sináptico.

Figura 5.5: Tiempo de computación y de acceso a memoria global y compartida en la detección a nivel de píxel, para un tamaño de bloque de 256 threads, y para la imagen *naufragos.bin*.

pico. Los mejores anchos de banda de memoria global se obtienen para el paralelismo sináptico, con un 90 % de los 192.4 GB/s. Los mejores anchos de banda de memoria compartida se obtienen para el paralelismo neuronal con una tasa de hasta 325.49 GB/s.



(a) Paralelismo neuronal (b) Paralelismo sináptico

Figura 5.6: Fracción de tiempo de ejecución total consumido por los accesos a memoria global, accesos a memoria compartida y operaciones en punto flotante para el algoritmo de detección a nivel de píxel, en las dos aproximaciones de paralelismo. Los datos se corresponden con un *chunk* de 2^{16} para la imagen *naufragos.bin*, utilizando un tamaño de bloque de threads de 256.

(a) Paralelismo neuronal

Chunk	Global (GB/s)	Shared (GB/s)	Rendimiento (GFLOPS)
2^4	1.30	2.80	4.69
2^6	5.01	11.28	18.48
2^8	20.34	45.30	73.60
2^{10}	80.31	176.43	289.37
2^{12}	144.28	294.44	549.21
2^{14}	166.21	315.77	519.10
2^{16}	166.29	318.63	549.38
2^{18}	166.94	319.98	551.99
2^{20}	169.06	325.49	532.69

(b) Paralelismo sináptico.

Chunk	Global (GB/s)	Shared (GB/s)	Rendimiento (GFLOPS)
2^4	6.61	11,95	8.89
2^6	26.42	47,78	35.51
2^8	79.37	114,00	81.97
2^{10}	173.25	129,86	74.07
2^{12}	125.12	133,08	80.82
2^{14}	122.39	134,06	82.94
2^{16}	123.88	134,25	82.68

Tabla 5.5: Ancho de banda y rendimiento de computación para el algoritmo de detección a nivel de píxel, para las estrategias de paralelización neuronal y sináptica, utilizando un tamaño de bloque de 256 threads para la imagen *naufragos.bin*.

Conclusiones y trabajo futuro.

Se han implementado sobre GPU dos algoritmos de detección de objetivos en imágenes hiperespectrales: algoritmo de detección a nivel de píxel, y algoritmo de detección multi-resolución. El algoritmo de detección a nivel de píxel realiza una búsqueda determinando de forma individual si cada píxel de la imagen contiene el objetivo, o una parte de éste, mientras que el algoritmo de detección multi-resolución realiza la búsqueda de manera iterativa en ventanas de tamaño decreciente, acotando el área delimitada por cada uno de los objetivos. En ambos algoritmos, la implementación sobre GPU ha conseguido aceleraciones favorables respecto a implementaciones optimizadas en CPU, alcanzándose por ejemplo un speedup máximo de 137.9x para el algoritmo de detección a nivel de píxel, y un 51.7x para el algoritmo de detección multi-resolución, para una imagen de $1024 \times 1024 \times 64$. Además para las imágenes utilizadas en este trabajo, el algoritmo de detección multi-resolución ha demostrado ser más eficiente en la detección que el algoritmo a nivel de píxel, localizando los objetivos en un tiempo mínimo de 0,0026s, frente a los 0,0094s consumidos por el algoritmo de detección a nivel de píxel, para la imagen $1024 \times 1024 \times 64$.

Por otra parte, ambos algoritmos hacen uso de ANNs para la localización de los objetivos. Estas ANNs han sido implementadas sobre la GPU utilizando dos aproximaciones distintas de paralelización: paralelismo neuronal y paralelismo sináptico. El paralelismo neuronal realiza un reparto de los threads de tal forma que un único thread se encarga de computar la salida de una única neurona de la capa oculta, mientras que para el paralelismo sináptico son varios los threads que colaboran en el cómputo de dicha salida, utilizándose uno por cada sinapsis existente entre la capa de entrada y la neurona de la capa oculta. Dado que el algoritmo de detección a nivel de píxel consiste únicamente en computar ANNs de forma independiente, los rendimientos de ambas aproximaciones paralelas pueden observarse a través de los resultados obtenidos para este algoritmo de detección. Así, las dos aproximaciones de paralelización de ANNs demuestran obtener rendimientos similares cuando el número de instancias ANN a computar simultáneamente es relativamente pequeño ($\lesssim 2^8$), mientras que para tamaños mayores el rendimiento obtenido por el paralelismo neuronal supera considerablemente al obtenido por el paralelismo sináptico.

La aceptable eficiencia de las implementaciones sobre GPU de los algoritmos de detección de objetivos son consecuencia de las diferentes estrategias de optimización llevadas a cabo para los distintos *kernels*. En todos ellos se ha cuidado, en la medida de lo posible, el acceso coalescente a memoria global, la minimización de divergencias de threads de un mismo warp, la ausencia de conflictos en el acceso a los bancos de memoria compartida, la utilización de memoria de constantes para accesos a los parámetros de las ANNs, la utilización de funciones matemáticas rápidas donde la precisión lo permitiese, así como la maximización de la ocupancia y del aprovechamiento de los recursos de la GPU, buscando el tamaño de bloque de threads óptimo, maximizando el reuso de la memoria compartida y registros utilizados por bloque, y mejorando la localidad de los accesos para aumentar el uso de las memorias caché, etc. Sobre todo, y dado que estas aplicaciones realizan muchos accesos a memoria, los factores clave han sido el mejor aprovechamiento de los accesos, mediante el incremento del número de operaciones que cada bloque realiza, ocultando así la latencia del movimiento de datos. Aún así el tiempo total de transferencia de datos a la GPU es muy superior al de computaciones lo que impide por el momento la detección de objetivos en tiempo real.

Para finalizar, cabe señalar como trabajo futuro el objetivo de detección en tiempo real que sólo se podría conseguir mediante solapamiento entre la computación de los algoritmos y el trasvase de los datos a memoria global. Por otra parte, otra línea de trabajo será ampliar la detección de objetivos independientemente de su escala, llevado a cabo por el algoritmo de detección multi-resolución, para realizar detección de objetivos independientemente de escala, orientación y posición, lo que permitirá detectar objetivos girados o posicionados entre 2 ventanas de la imagen. Finalmente estos algoritmos pueden ser aplicados con algunos cambios y después del entrenamiento adecuado de las redes neuronales a un amplio espectro de aplicaciones de imágenes hiperespectrales.

BIBLIOGRAFÍA

[1] Alexander F.H. Goetz, Gregg Vane, Jerry E. Solomon, and Barrett N. Rock. Imaging spectrometry for earth remote sensing. *Science*, 228(4704):1147–1153, 1985.

[2] F. A. Kruse, J. W. Boardman, and J. F. Huntington. Comparison of airborne hyperspectral data and eo-1 hyperion for mineral mapping. *IEEE Transactions on Geoscience and Remote Sensing*, 41(6):1388–1400, June 2003.

[3] Jens Oldeland, Wouter Dorigo, Lena Lieckfeld, Arko Lucieer, and Norbert Jürgens. Combining vegetation indices, constrained ordination and fuzzy classification for mapping semi-natural vegetation units from hyperspectral imagery. *Remote Sensing of Environment*, 114(6):1155 – 1166, 2010.

[4] Ville Heikkinen, Timo Tokola, Jussi Parkkinen, Ilkka Korpela, and Timo Jaaskelainen. Simulated multispectral imagery for tree species classification using support vector machines. *IEEE T. Geoscience and Remote Sensing*, 48(3-2):1355–1364, 2010.

[5] F. López-Pena, J. L. Crespo, and R. J. Duro. Unmixing low-ratio endmembers in hyperspectral images through gaussian synapse ANNs. *Instrumentation and Measurement*, 59(7):1834 – 1840, 2010.

[6] J. E. Freeman, S. Panasyuk, A. E. Rogers, S. Yang, and R. Lew. Advantages of intraoperative medical hyperspectral imaging (MHSI) for the evaluation of the breast cancer resection bed for residual tumor. *Journal of Clinical Oncology, 2005 ASCO Annual Meeting Proceedings, vol 23, No. 16S, Part I of II (June 1 Supplement), 709*, 2005.

[7] L. Liu, M.O. Ngadi, S.O. Prasher, and C. Gariépy. Categorization of pork quality using gabor filter-based hyperspectral imaging technology. *Journal of Food Engineering*, 99(3):284 – 293, 2010.

[8] Virginie Fresse, Dominique Houzet, and Christophe Gravier. GPU architecture evaluation for multispectral and hyperspectral image analysis. In ECSI, editor, *DASIP Design and Architectures of Signal and image processing*, page 7, Edinburgh, Royaume-Uni, October 2010.

[9] Yuliya Tarabalka, Trym Haavardsholm, Ingebjørg Kåsen, and Torbjørn Skauli. Real-time anomaly detection in hyperspectral images using multivariate normal mixture models and gpu processing. *Journal of Real-Time Image Processing*, 4:287–300, 2009.

[10] D. Cohen, M. Arnoldussen, G. Bearman, and W.S. Grundfest. The use of spectral imaging for the diagnosis of retinal disease. *IEEE*, 1999.

[11] Antonio Plaza, Javier Plaza, and Hugo Vegas. Improving the performance of hyperspectral image and signal processing algorithms using parallel, distributed and specialized hardware-based systems. *J. Signal Process. Syst.*, 61:293–315, December 2010.

[12] S. Sánchez, G. Martín, A. Plaza, and C.-I. Chang. GPU implementation of fully constrained linear spectral unmixing for remotely sensed hyperspectral data exploitation. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 7810 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, August 2010.

[13] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[14] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28:39–55, 2008.

[15] NVIDIA Corporation. NVIDIA CUDA C Programming Guide. `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf`, 2010. Version 3.2.

[16] D. Manolakis, D. Marden, and G. A. Shaw. Hyperspecral image processing for automatic target detection applications. *MIT Lincolm Laboratory Journal*, 14:79–116, 2003.

[17] Antonio Plaza, Javier Plaza, and Sergio Sánchez. Parallel implementation of endmember extraction algorithms using NVidia graphical processing units. In *IGARSS (5)*, pages 208–211, 2009.

[18] Abel Paz and Antonio Plaza. Clusters versus GPUs for parallel target and anomaly detection in hyperspectral images. *EURASIP J. Adv. Sig. Proc.*, 2010, 2010.

[19] E. Merényi, W.H. Farrand, J.V. Taranik, and T.B. Minor. Classification of hyperspectral imagery with neural networks: Comparison to conventional tools. In *Machine Learning Reports*, volume 4, 2011. `http://www.techfak.uni-bielefeld.de/~fschleif/mlr/mlr_04_2011.pdf`.

[20] Subramanian S., Gat N., Sheffield M., Barhen J., and Toomarian N. Methodology for hyperspectral image classification using novel neural network. In *Proc. of SPIE*, volume 3071, pages 128–137, 1997.

[21] D.L. Ly, V. Paprotski, and D. Yen. Neural networks on GPUs: Restricted boltzmann machines. *Department of Electrical and Computer Engineering, University of Toronto*, 2009.

[22] Honghoon Jang, Anjin Park, and Keechul Jung. Neural network implementation using CUDA and OpenMP. In *Proceedings of the 2008 Digital Image Computing: Techniques and Applications*, pages 155–161, Washington, DC, USA, 2008. IEEE Computer Society.

[23] M.A. Bhuiyan, Vivek K. Pallipuram, and Melissa C. Smith. Acceleration of spiking neural networks in emerging multi-core and gpu architectures. In *IPDPS Workshops 10*, pages 1–8, 2010.

[24] J.M. Nageswaran, N. Dutt, J.L. Krichmar, A.Nicolau, and A.V. Veidenbaum. A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural Networks*, 22(5-6):791 – 800, 2009.

[25] J. Park and I. W. Sandberg. Universal approximation using radial-basis-function networks. *Neural Comput.*, 3:246–257, June 1991.

[26] Daniel Svozil, Vladimír Kvasnicka, and Jirí Pospichal. Introduction to multi-layer feed-forward neural networks. *Chemometrics and Intelligent Laboratory Systems*, 39(1):43 – 62, 1997.

[27] D. Kirk, W.M.W. Hwu, and W. Hwu. *Programming massively parallel processors: a hands-on approach*. Applications of GPU Computing Series. Morgan Kaufmann Publishers, 2010.

[28] NVIDIA. Fermi Compute Architecture Whitepaper. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

[29] NVIDIA Corporation. NVIDIA CUDA C Best Practices Guide. `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf`, 2010. Version 3.2.

[30] Daren Lee, Ivo Dinov, Bin Dong, Boris Gutman, Igor Yanovsky, and Arthur W. Toga. CUDA optimization strategies for compute- and memory-bound neuroimaging algorithms. *Computer Methods and Programs in Biomedicine*, In Press, Corrected Proof:–, 2010.

[31] NVIDIA. NVIDIA CUDA technical training, vol. II: CUDA case studies. `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`, 2008.

[32] Andrew Davidson and John D. Owens. Register packing for cyclic reduction: a case study. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 4:1–4:6, New York, NY, USA, 2011. ACM.

[33] NVIDIA. NVIDIA CUDA CUBLAS library. `http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUBLAS_Library_2.0.pdf`, March 2008.

[34] D.B. Heras, F. Argüello, J. López Gómez, J.A. Becerra, and Richard J. Duro. Towards Real-time Hyperspectral Image Processing, a GP-GPU Implementation of Target Identification. In *The 6th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, Prague, Czech Republic, 15-17 September 2011.

[35] Y. Zhang, J. Cohen, and J.D. Owens. Fast tridiagonal solvers on the GPU. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 127–136, New York, NY, USA, 2010. ACM.