Facultad de Ciencias

Paralelización Eficiente en GPUs mediante Tensor Cores y CUDA Cores

(Efficient Tensor-CUDA Core Parallelism in GPUs)

Trabajo de Fin de Grado

para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Pablo Saura Sánchez

Director: Borja Pérez Pavón

Junio - 2025

Índice

Resumen

En este trabajo se aborda el diseño y la implementación de kernels de convolución Gaussiana que combinan CUDA Cores y Tensor Cores en GPUs modernas, con especial atención a los desafíos de organización de datos y gestión de memoria. Desarrollamos una transformación manual del vecindario de cada píxel en una matriz intermedia, garantizando que los 32 threads de cada warp lean posiciones contiguas en memoria global. Se declarará un espacio fijo de memoria shared para que distintos fragmentos (entrada, filtro y acumulador) compartan el mismo bloque de `shared` sin interferencias.

La obligación de declarar un tamaño fijo de memoria compartida llevó a diseñar un esquema en el que, aunque el buffer se reserva una sola vez por bloque, cada warp calcula su propio offset de lectura y escritura, salvaguardando espacio para otros warps que cubran la latencia. Además, reescribimos comprobaciones de límites y bucles de carga con operaciones aritméticas y máscaras, lo que unificó el flujo de ejecución dentro de cada warp y limitó al mínimo las sincronizaciones necesarias.

En paralelo, investigamos distintas estrategias de paralelismo: kernels dedicados por completo a CUDA Cores, otros al 100 % a Tensor Cores y versiones híbridas que alternan dinámicamente entre ambos. Esta variedad de aproximaciones permitió identificar los cuellos de botella reales (tanto en cómputo como en memoria) y probar técnicas para optimizar la coalescencia en caches L1/L2. El resultado es un conjunto de prácticas para maximizar el rendimiento en operaciones matriciales de gran tamaño, que serán evaluadas en detalle en las secciones siguientes.

Abstract

In this work, we explore the design and implementation of Gaussian convolution kernels that combine CUDA Cores and Tensor Cores on modern GPUs, paying particular attention to data-layout challenges and memory management.We manually transformed each pixel's neighborhood into an intermediate matrix, ensuring that all 32 threads of a warp read contiguous global-memory addresses. We create a fixed space in shared memory to coexist different segments (input, filter and accumulator) in the same shared block without interference.

Because shared memory requires a fixed size declaration, we devised a scheme in which the buffer is reserved just once per block, while each warp computes its own read/write offset. Furthermore, to eliminate costly branch divergences, we replaced boundary checks and load loops with purely arithmetic operations and bitmasking, unifying the execution path within each warp and minimizing necessary synchronizations.

Simultaneously, we evaluated several parallelization strategies: kernels fully dedicated to CUDA Cores, others exclusively using Tensor Cores, and hybrid versions that dynamically switch between the two. This range of approaches let us identify the real bottlenecks—both compute- and memory-bound—and experiment with techniques to improve coalescing in L1/L2 caches. The end result is a collection of practices for squeezing out maximum performance in large-scale matrix operations, which we will examine in detail in the sections that follow.

1. Introducción

1.1. Una breve historia de las GPUs

El origen de las unidades de procesamiento gráfico (GPUs) se remonta a los primeros aceleradores 3D de finales de los años noventa, diseñados principalmente para descongestionar a la CPU de las tediosas rutinas de rasterizado y textura en los videojuegos [1]. Empresas como NVIDIA y ATI (hoy AMD) compitieron por integrar cada vez más etapas del pipeline gráfico en hardware dedicado, lo que llevó a la aparición de arquitecturas programables con lenguajes como OpenGL y Direct3D. Con la llegada de la GPU GeForce 3 en 2001, se introdujeron los primeros shaders programables, sentando las bases de una programación de propósito general en la GPU (GPGPU) [1]. A partir de ese momento, investigadores y desarrolladores empezaron a aprovechar estos procesadores masivamente paralelos para tareas ajenas al renderizado, abriendo el camino a aplicaciones científicas, financieras y de machine learning.

A medida que las GPUs fueron madurando, su diseño evolucionó para incorporar miles de núcleos sencillos capaces de ejecutar en paralelo las mismas instrucciones sobre distintos datos. La arquitectura CUDA de NVIDIA, presentada en 2007, marcó un antes y un después al ofrecer un modelo de programación unificado y accesible desde C/C++, con abstracciones como threads, bloques y grids [1]. Esto permitió a los programadores explorar el potencial masivo de cómputo paralelo sin tener que exprimir trucos de shader. Con cada generación, la densidad de transistores aumentó, y la memoria de alta velocidad (GDDR, luego HBM) creció en ancho de banda.

Solo en los últimos años las GPUs han incorporado unidades especializadas destinadas al cálculo matricial con precisión mixta: los Tensor Cores. Introducidos con la arquitectura Volta en 2017, estos núcleos internos están diseñados para acelerar multiplicaciones de matrices y acumulaciones (MMA), operaciones centrales en las redes neuronales profundas [2, 3]. Desde entonces, cada nueva arquitectura (Turing, Ampere, Hopper) ha refinado y ampliado el número y capacidad de los Tensor Cores, consolidando a las GPUs como la principal fuente de cómputo en aplicaciones IA entre otras [4].

1.2. Por qué importa el cómputo paralelo

En la era del big data y la inteligencia artificial, los problemas a resolver implican conjuntos de datos y modelos cada vez más grandes. Intentar procesar billones de elementos uno a uno en un único núcleo resulta inviable: los tiempos de ejecución se dispararían y el consumo energético sería prohibitivo [1]. La computación paralela ofrece la solución al descomponer un problema en tareas que puedan ejecutarse simultáneamente, aprovechando hardware con cientos o miles de unidades de cómputo. Este enfoque no solo reduce drásticamente los tiempos de ejecución, sino que también permite un uso más eficiente de la energía, al repartir la carga de trabajo entre varios núcleos que operan a menor frecuencia.

Las GPUs llevan esta filosofía al extremo: mientras mientras una CPU prioriza el rendimiento de cada thread individual mediante técnicas avanzadas como la

ejecución fuera de orden, una GPU apuesta por el multi-threading masivo, sacrifica complejidad en cada núcleo a cambio de tener miles de threads trabajando en paralelo [1]. Este hecho es especialmente relevante en simulaciones físicas, procesamiento de imágenes o cálculos de redes neuronales, donde las mismas operaciones deben repetirse millones o miles de millones de veces. Al distribuir esas operaciones entre threads, se consigue un speedup lineal (o cercano a lineal) con cada unidad de cómputo adicional, siempre que la tarea sea lo suficientemente granular y un acceso a memoria óptimo.

Sin embargo, escribir código paralelo conlleva varios problemas: hay que gestionar correctamente la sincronización entre threads, evitar condiciones de carrera y maximizar la reutilización de datos en cache y memoria compartida. La arquitectura CUDA facilita estas tareas, pero el programador debe diseñar cuidadosamente el algoritmo para minimizar conflictos y lograr un alto nivel de ocupación. La computación paralela es el pilar sobre el que se sostienen las aplicaciones de alto rendimiento de hoy, y las GPUs son el medio más accesible y flexible para explotarla a escala masiva si saben aprovecharse correctamente.

1.3. La importancia de las GPUs hoy en día

Hoy en día, las GPUs son muy importantes más allá del mundo de los videojuegos. En la investigación científica, aceleran simulaciones de dinámica molecular, modelado climático y análisis de grandes volúmenes de datos astronómicos. En ingeniería, se emplean para el diseño asistido por ordenador (CAD), la simulación de fluidos y la optimización de estructuras. En finanzas, permiten evaluar carteras de inversión en tiempo real y simular riesgos mediante métodos de Monte Carlo a gran escala.

El procesamiento de imágenes y vídeo también se ha visto revolucionado: el decodificación de vídeo en streaming, la reconstrucción 3D en tiempo real y la corrección de color a nivel profesional dependen de la capacidad de la GPU para manejar píxeles en paralelo. Por su parte, la industria del automóvil recurre a las GPUs para la conducción autónoma, procesando en milisegundos las imágenes de múltiples cámaras y sensores LIDAR.

Las GPU han supuesto un gran avance para la inferencia en redes neuronales [5]. Modelos de lenguaje natural, arquitecturas de visión por computador y sistemas de recomendación actualmente son entrenados en clústeres de GPU que suman decenas o centenares de tarjetas. En estos escenarios, la gran capacidad de paralelización en multiplicaciones matriciales y convoluciones ha sido el factor clave para alcanzar los niveles de precisión y velocidad que hoy vemos en servicios de traducción automática, asistentes virtuales y aplicaciones de diagnóstico médico asistido por IA.

1.3.1. Redes neuronales de gran escala y cargas de trabajo intensivas

Hoy en día los arquitectos de modelos buscan capas cada vez más anchas y profundas, lo que implica multiplicaciones de matrices gigantescas y pasos de convolución de alta dimensión. Entrenar un transformer con miles de millones de parámetros puede consumir semanas de cómputo en cientos de GPUs [5], mientras que la inferencia en tiempo real (por ejemplo, para asistentes de voz o análisis de vídeo en

streaming) exige latencias por debajo de los pocos milisegundos. Esta combinación de computaciones masivas y restricciones de tiempo real ha exigido a la industria rediseñar tanto hardware como algoritmos para exprimir la paralelización al máximo.

1.3.2. Límites de los pipelines tradicionales de GPU

A pesar de su gran capacidad, las GPU basadas únicamente en CUDA Cores comienzan a mostrar sus límites cuando la carga de trabajo incluye decenas de miles de operaciones de multiplicación y suma por unidad. El pipeline SIMD/ SIMT tradicional se enfrenta a cuellos de botella en la memoria (tanto en caches L1/L2 como en DRAM) y a conflictos en memoria compartida cuando la cantidad de threads supera cierto umbral. Además, el coste de mover los datos desde la memoria global a la compartida y la frecuencia de sincronizaciones interwarp pueden reducir buena parte de la ganancia esperada por paralelizar el cómputo.

1.4. La llegada de los Tensor Cores

Para afrontar estos nuevos retos nació la idea de añadir unidades especializadas en multiplicación de matrices de tamaño fijo directamente dentro de cada Streaming Multiprocessor. Con Volta, en 2017, NVIDIA introdujo los primeros Tensor Cores, diseñados para ejecutar en un solo paso operaciones de *Mixed-Precision Matrix Multiply-Accumulate (MMA)*, capaces de combinar rápido cómputo en media precisión con acumulaciones en precisión completa.

1.4.1. Qué los hace diferentes

A diferencia de un CUDA Core, que ejecuta instrucciones escalares o vectoriales clásicas, un Tensor Core procesa bloques predefinidos (por ejemplo, 16×16 o 32×8) de datos en punto flotante o entero y realiza el producto de matrices y la suma acumulada en un solo ciclo de reloj. Este enfoque reduce drásticamente el número de instrucciones necesarias y permite tasas de throughput que superan en varios órdenes de magnitud a las de los pipelines tradicionales, siempre que los datos estén alineados y organizados según sus requerimientos.

1.4.2. Operaciones de matrices en precisión mixta

La clave está en un compromiso entre velocidad y precisión: los Tensor Cores trabajan con formatos reducidos (FP16, bfloat16, TF32, INT8, etc.) para maximizar el paralelismo, pero conservan la exactitud de la acumulación en FP32, evitando que el error numérico aumente a límites graves. Esta combinación, conocida como *mixed-precision*, es ideal para redes neuronales: por un lado, acelera el entrenamiento; por otro, mantiene la calidad del modelo final.

1.5. Objetivos y estructura del TFG

El propósito central de este TFG ha sido explorar hasta qué punto resulta rentable combinar la computación tradicional en CUDA Cores con el paralelismo masivo

de los Tensor Cores de las GPUs modernas, concretamente en el contexto de aplicar filtros gaussianos sobre imágenes de alta resolución. Para ello, en primer lugar se diseñaron tres familias de kernels: unos que vuelcan toda la carga de trabajo en los CUDA Cores, otros que delegan al cien por ciento en los Tensor Cores y un tercer grupo híbrido que alterna dinámicamente entre ambos tipos de unidad. El objetivo fue comparar su rendimiento sobre filtros de dimensiones crecientes (desde 3×3 hasta 243×243) y variar asimismo el número de filtros simultáneos, evaluando así cómo escala cada estrategia.

A partir de ese punto de partida, se detallaron una serie de objetivos específicos que guiaron el desarrollo a lo largo de varias etapas. En la fase de preparación, se implementó la transformación de la convolución en un producto matricial, ajustando la disposición de datos en memoria global y compartida para maximizar la localidad espacial y minimizar lecturas redundantes. A continuación, se avanzó en la optimización de cada kernel: se aplicaron técnicas de coalescencia de accesos orientado a sostener elevadas tasas de acierto en L1 y L2. Paralelamente, se definió un riguroso plan de benchmarks que incluyó mediciones con marcas de tiempo (‘time.h‘) y un profiling fino con NVIDIA Nsight, de modo que en cada iteración se pudieran identificar y atacar los cuellos de botella de cómputo y memoria.

En la fase final, los estudios de caso con varias imágenes permitieron contrastar los resultados entre los kernels dedicados y los híbridos, ilustrando las ventajas competenciales de los Tensor Cores cuando los volúmenes de datos y el número de filtros crecen. Con todo ello, este trabajo aporta no solo un conjunto de implementaciones comparativas, sino también un conjunto de buenas prácticas (desde la organización de buffers en memoria compartida hasta la estrategia de subdivisión de trabajo en fragmentos compatibles con WMMA) que servirán de referencia a futuros desarrollos de código paralelo en GPUs.

1.6. Estructura del documento

El capítulo 2 presenta los conceptos clave sobre CUDA, la arquitectura de las GPUs modernas y el funcionamiento de los Tensor Cores. En el capítulo 3 se describe el diseño y desarrollo de los distintos kernels implementados, junto con las técnicas de optimización aplicadas. El capítulo 4 recoge los resultados experimentales y el análisis del rendimiento obtenido en distintos escenarios. Finalmente, el capítulo 5 expone las conclusiones extraídas del trabajo y plantea posibles líneas de mejora y extensión.

2. Background

A continuación, se presentarán las bases teóricas y tecnológicas fundamentales que sustentan el desarrollo de este proyecto. En primer lugar se explicará los fundamentos de la programación en CUDA

2.1. Fundamentos del modelo de programación CUDA

CUDA (Compute Unified Device Architecture) es la plataforma de programación creada por NVIDIA que nos permite escribir código paralelo directamente para la GPU utilizando una extensión de C/C++. A través de CUDA, podemos definir funciones que se ejecutarán en la GPU, llamadas kernels. Una de sus principales características es el modelo de ejecución jerárquico, que permite distribuir la carga de trabajo entre miles de threads [6, 7]. Esta jerarquía no representa directamente cómo se ejecuta el código a nivel físico en la GPU, sino que es una abstracción pensada para que el programador pueda estructurar de forma sencilla problemas complejos y aprovechar el paralelismo masivo del hardware.

2.1.1. Threads, Blocks y Grids

Threads

Los *threads* son la unidad más básica de ejecución. Cada thread ejecuta el mismo kernel, pero de forma independiente, lo que permite que cada uno trabaje sobre un conjunto distinto de datos [8]. Aunque todos los threads corren el mismo código, lo hacen con identificadores únicos, lo que les permite saber qué parte del problema deben resolver.

En CUDA, cada thread puede identificarse mediante las variables `threadIdx.x`, `threadIdx.y` y `threadIdx.z`, que indican su posición dentro del bloque al que pertenece [6]. Estas coordenadas permiten organizar threads en una, dos o tres dimensiones, lo cual resulta especialmente útil para representar estructuras de datos multidimensionales como matrices o volúmenes.

Por ejemplo, si un thread tiene `threadIdx.x = 5`, significa que es el sexto thread (empezando desde 0) dentro de su dimensión X. Esta información suele utilizarse para calcular índices y acceder a los datos correspondientes dentro de un array.

Blocks

Para poder lanzar una gran cantidad de threads de forma estructurada, CUDA los agrupa en bloques, llamados *blocks* [6]. Cada bloque contiene un conjunto de threads que comparten ciertas características (que se explicarán más adelante). A nivel organizativo, los bloques permiten dividir el trabajo en pequeñas tareas más manejables y escalables.

Un bloque también puede tener hasta tres dimensiones, definidas por `blockDim.x`, `blockDim.y` y `blockDim.z`. Dentro de un bloque, cada thread tiene una posición relativa (su `threadIdx`) y puede saber cuántos threads hay en total dentro del mismo bloque mediante `blockDim`.

Es importante tener en cuenta que los bloques tienen una limitación en cuanto al número máximo de threads que pueden contener. En la mayoría de GPUs modernas, el límite está en **1024 threads por bloque**. Esto significa que si se necesita más paralelismo, hay que distribuir los threads en varios bloques.

Un ejemplo muy común es definir bloques unidimensionales de 128 o 256 threads y lanzar varios bloques para cubrir todo el espacio de datos. De este modo, se mantiene un equilibrio entre rendimiento y flexibilidad.

Grids

El nivel superior de esta jerarquía es el *grid*, que no es más que un conjunto de bloques [8]. Cuando se lanza un kernel desde el host, se define el número de threads por bloque y bloques que se desean lanzar, y CUDA se encarga de distribuirlos entre los multiprocesadores de la GPU.

Al igual que los bloques, las grids también pueden tener una, dos o tres dimensiones. Esto permite representar estructuras complejas de datos con una distribución lógica clara. Cada bloque dentro de la grid se identifica con `blockIdx.x`, `blockIdx.y` y `blockIdx.z`, y puede saber cuántos bloques hay en total mediante `gridDim`.

Combinando los identificadores del thread y del bloque, se puede calcular el índice global del thread en el grid. Este índice se usa frecuentemente para acceder a estructuras de datos lineales como arrays. Por ejemplo:

```
int globalIndex = blockIdx.x * blockDim.x + threadIdx.x;
```

El programador tiene control total sobre cómo se organizan los threads y bloques. Es decir, no es la GPU quien decide la estructura de ejecución, sino que es el propio desarrollador quien debe especificar explícitamente cuántos threads tendrá cada bloque y cuántos bloques compondrán el grid.

Para ello, CUDA ofrece el tipo `dim3`, una estructura que permite definir dimensiones unidimensionales, bidimensionales o tridimensionales tanto para los bloques como para el grid. Esto se hace en la llamada al kernel, utilizando la sintaxis con triple chevron («`<...>`»).

Por ejemplo, si se quiere lanzar un kernel llamado *myKernel* usando argumentos a y b, donde cada bloque tenga 16×16 threads y el grid esté compuesta por 32×32 bloques, la llamada sería la siguiente:

```
dim3 blockSize(16, 16);  // 256 threads por bloque
dim3 gridSize(32, 32);   // 1024 bloques en total

myKernel<<<gridSize, blockSize>>>(a,b);
```

2.1.2. Warps y ejecución SIMD (SIMT)

Aunque en la programación CUDA se conceptualizan threads individuales, la unidad mínima de ejecución en una GPU no corresponde a threads aislados, sino a entidades denominadas *warps* [9, 10]. Un **warp** consiste en un grupo fijo de 32 threads que se ejecutan de manera conjunta. Esta agrupación está diseñada para optimizar el procesamiento eficiente del hardware, siendo fundamental comprender su funcionamiento para maximizar el rendimiento de las aplicaciones.

Funcionamiento del Warp

Un warp está constituido exactamente por **32 threads**, en todas las arquitecturas CUDA actuales. Los threads dentro de un mismo warp ejecutan la misma instrucción de manera simultánea, compartiendo un contador de programa único. Este comportamiento permite que la GPU procese warps completos como unidades atómicas de ejecución, en lugar de manejar threads de forma independiente. Internamente, el hardware planifica y despacha estos grupos como entidades indivisibles.

Modelo de ejecución SIMT

CUDA emplea un paradigma denominado *SIMT* (Single Instruction, Multiple Threads), análogo al modelo SIMD (Single Instruction, Multiple Data) [9, 10]. En este esquema, cada thread de un warp ejecuta idénticas instrucciones sobre datos distintos. Aunque el flujo de código es uniforme para todos los threads del warp, cada uno opera sobre su propio conjunto de datos.

Divergencia de control y sus implicaciones

Cuando threads de un warp siguen rutas de ejecución diferentes, por ejemplo, al bifurcarse mediante instrucciones condicionales como `if`— se genera *divergencia*. En tales casos, la GPU serializa la ejecución: primero procesa los threads que cumplen la condición y posteriormente los restantes, mientras los threads inactivos permanecen en espera. Este modelo reduce el paralelismo efectivo del warp y puede conllevar una pérdida de rendimiento. Es por ello que las instrucciones que producen divergencias de ejecución entre los threads de un mismo warp [11, 12]

Al ejecutar un kernel, CUDA organiza automáticamente los threads en warps de 32 elementos [13]. Si el número total de threads no es múltiplo de 32, el último warp se completa con threads inactivos, los cuales consumen recursos sin contribuir al cómputo [12].

2.1.3. Streaming Multiprocessors y planificación del Hardware

Los *Streaming Multiprocessors* (SMs) son las unidades fundamentales de procesamiento. Cada SM actúa con una lógica de procesamiento independiente capaz de gestionar cientos de threads simultáneamente.

Arquitectura de los Streaming Multiprocessors

Cada SM integra componentes especializados que trabajan en conjunto para gestionar el paralelismo [13]. En su núcleo se encuentran los CUDA Cores, unidades aritméticas capaces de ejecutar operaciones diversas [12]. Cada CUDA Core está especializados para un tipo de operaciones distintos, ya sea para cálculo con enteros, FP32, FP64, loads, stores y operaciones matemáticas complejas (SFU) Cada SM cuenta con varios registros asignados exclusivamente a cada thread, permitiendo almacenar variables temporales sin acceder a memorias más lentas, reduciendo el tiempo de espera.

Planificación de Warps

Como puede verse en la figura 1 Cada SM incluye múltiples *warp schedulers* (generalmente cuatro en GPUs recientes) que trabajan en paralelo para optimizar el uso de recursos [12, 14]. El proceso de planificación sigue tres etapas principales. Primero, los schedulers seleccionan los warps que están listos para ejecutarse, es decir, aquellos cuyas unidades funcionales y datos necesarios están disponibles. Luego, lanzan una instrucción a los núcleos CUDA, haciendo que todos los threads del warp la ejecuten paralelamente aprovechando las unidades funcionales disponibles. Si un warp se bloquea (por ejemplo, al requerir datos de la memoria global), el scheduler no espera a que se resuelva la operación [12]. En lugar de ello, cambia inmediatamente a otro warp activo, tratando que los núcleos CUDA no permanezcan inactivos [12].

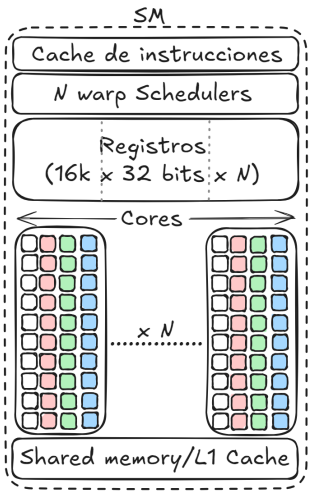

Figura 1: Representación arquitectural de un SM

2.1.4. Balanceo de carga y Ocupación

Balanceo de carga

El *balanceo de carga* indica cómo se reparte el trabajo entre los diferentes Streaming Multiprocessors [13]. Idealmente, cada SM debería recibir una cantidad similar de trabajo para evitar que existan sobrecargas o periodos de inactividad. Cuando se lanza un kernel, la GPU asigna bloques a los SMs disponibles [13]. Si el número total de bloques no es múltiplo del número de SMs, algunos multiprocesadores terminarán procesando más bloques que otros. Por ejemplo, si una GPU tiene 30 SMs y se lanzan 32 bloques, dos SMs procesarán dos bloques cada uno, mientras que los demás solo uno [13]. Para mitigar este problema, es recomendable lanzar una cantidad de bloques significativamente mayor que el número de SMs. De esta forma, la GPU puede distribuir los bloques de manera dinámica, asignando nuevos bloques a los SMs que terminen primero su trabajo [13].

Ocupación

La *ocupación* mide cuántos threads están activos en un SM en relación con su capacidad máxima [12]. Un SM con alta ocupación tiene muchos warps listos para ejecutar, lo que permite a los warp schedulers alternar entre ellos rápidamente cuando algunos se bloquean (por ejemplo, al acceder a memoria) [12]. Esto es crítico para el ocultamiento de latencia, ya que evita que los núcleos CUDA permanezcan inactivos. Sin embargo, esta puede estar limitada por tres factores principales [12]:

- Registros por thread: Cada thread utiliza registros para almacenar variables temporales. Si un kernel requiere de muchos registros, el SM no podrá alojar tantos threads [12].
- Memoria compartida por bloque: Si un bloque reserva mucha memoria compartida, el número de bloques paralelos en cada SM disminuirá [12].

- Límite de threads por SM: Cada arquitectura de GPU impone un máximo de threads simultáneos por SM [12].

Priorizar la ocupación por encima de estos factores es un error. Por ejemplo, usar bloques muy grandes (1024 threads) podría reducir el número de bloques por SM, limitando la flexibilidad del planificador. Es por ello que valores entre 128 y 256 son los más recomendados [12].

2.2. Jerarquía de memoria en CUDA

La GPU dispone de diferentes tipos de memoria, cada uno con sus propias características, latencias y ámbitos de visibilidad [13]. No todas se comportan igual ni se accede a ellas de la misma manera:

Registros

Los *registros* están más cercanos a los CUDA Cores que cualquier otra memoria. Cada thread dispone de sus propios registros, que son accesibles únicamente por él. Estos son accedidos directamente por las instrucciones de datos, provocando una latencia mínima [13]. Se utilizan de forma automática para almacenar variables locales, aunque el programador no tiene control directo sobre ellos.

Sin embargo, los registros son un recurso escaso. Cada SM tiene un número limitado, y si un kernel utiliza demasiados registros por thread, la GPU tendrá que reducir la cantidad de bloques planificados a cada SM para no exceder el límite total. Esto se conoce como *register pressure* y puede impactar negativamente en la ocupación del dispositivo [12]. Además, si un thread necesita más espacio que el disponible en registros, se utilizará memoria local, que es mucho más lenta.

Shared Memory

A nivel físico, la memoria compartida se aloja dentro de la cache L1 de cada SM. Esto significa que parte del espacio disponible para cache se reserva para la shared memory, y por tanto, cuanto más espacio se use para memoria compartida, menos queda disponible para el resto de la cache. Este reparto puede afectar al rendimiento si no se tiene en cuenta.

La *memoria compartida* no se usa automáticamente, sino que es el programador quien tiene que reservar explícitamente su tamaño y definir cómo van a acceder a ella los threads. Este tamaño debe poder resolverse en tiempo de compilación. Esto implica cierta planificación, sobre todo para evitar condiciones de carrera cuando varios threads intentan leer o escribir sobre la misma posición al mismo tiempo [15].

Una de las principales ventajas de esta memoria es que permite actuar como una especie de "cache hecha a mano". Podemos copiar en ella datos de la memoria global y reutilizarlos entre varios threads del mismo bloque, lo que reduce bastante el número de accesos a memoria global, que es mucho más lenta [15]. Esto es especialmente útil en algoritmos donde los threads colaboran entre sí, como por ejemplo en reducciones, escaneos o transposiciones.

Memoria global

La *memoria global* es la región de memoria principal de la GPU. Es accesible por cualquier thread, sin importar a qué bloque pertenezca. Ofrece una gran capacidad (varios GB), pero sus accesos tienen latencias muy elevadas, del orden de cientos de ciclos de reloj [13].

El uso de memoria global es inevitable para almacenar datos grandes, pero es fundamental que los accesos estén bien organizados. CUDA utiliza una técnica llamada *coalescing*, que agrupa accesos contiguos de varios threads en una única transacción de memoria [16]. Para ello, los threads deben acceder a direcciones consecutivas y alineadas. Si los accesos están desordenados o cada thread lee posiciones muy alejadas, se pierden las ventajas de coalescing y el número de transacciones aumenta, degradando el rendimiento.

Memoria constante

La *memoria constante* es una pequeña región (normalmente 64 KB) que puede ser leída por todos los threads. Está pensada para datos inmutables, como parámetros globales o tablas de consulta. Su principal ventaja es que dispone de una cache especializada que permite lecturas muy rápidas, siempre que todos los threads accedan a la misma posición. Cuando cada thread accede a una dirección distinta, se produce un fenómeno similar a la divergencia, y se pierde eficiencia [11].

Otra característica interesante es que esta memoria se carga desde el host mediante la función `cudaMemcpyToSymbol`, y no se puede modificar desde el propio kernel.

Memoria local

A pesar de su nombre, la *memoria local* no es compartida entre threads. Es una porción de la memoria global reservada de forma privada para cada thread por si se exceden los límites de los registros. Por ejemplo, si una variable local es un array grande o una estructura compleja, se guarda en memoria local por si no entra en los registros. Esta región tiene la misma latencia que la memoria global y su uso introduce accesos mucho más lentos que los registros [13].

Memoria de textura

CUDA ofrece mecanismos adicionales para acceder a datos 2D o 3D mediante la *memoria de textura*. Esta memoria permite aprovechar una cache especializada y aplicar filtrado, direccionamiento y otros efectos propios de los motores gráficos. Son especialmente útiles en aplicaciones donde los accesos no son secuenciales o presentan patrones espaciales complejos, como en procesado de imágenes o simulaciones físicas [12].

Caches L1 y L2

Además de las memorias anteriores, las GPUs modernas incluyen niveles de cache similares a los de una CPU [13]. La *cache L1* suele estar asociada a cada SM y puede

usarse conjuntamente con la memoria compartida. La *cache L2* está situada a nivel de dispositivo y es común a todos los SM.

2.3. Mecánicas de ejecución de threads y bloques

Cuando se lanza un kernel, la GPU divide todo el trabajo en los bloques de threads, asignándolos a distintos SM [12,13]. Cada SM puede ejecutar varios bloques concurrentemente, dependiendo de los recursos que estén consumiendo cada uno (como el uso de registros o memoria compartida) [12, 15]. Esto hace que no todos los bloques empiecen al mismo tiempo, y que algunos tengan que esperar en cola hasta que haya recursos disponibles [12]. Una vez un bloque es asignado a un SM, se divide en warps.

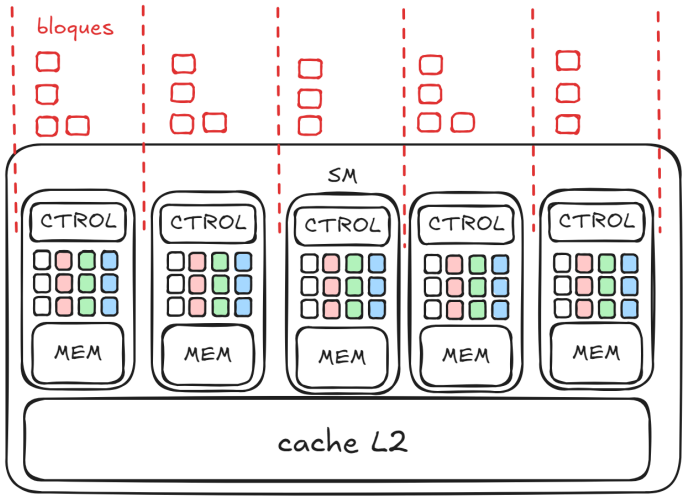

Figura 2: División de bloques en los distintos SM

2.3.1. Políticas de planificación de Warps

El orden en que se ejecutan los warps viene determinado por la política de planificación [12]. Aunque no está completamente documentada por NVIDIA, lo más habitual es que se utilice una estrategia *GTO* (Greedy Then Oldest), en la que se recorren todos los warps activos de forma cíclica, como puede verse en la figura 2. Sin embargo, en algunas arquitecturas también se aplican políticas más avanzadas, como dar prioridad a los warps que no tienen operaciones de acceso a memoria pendientes [14], o priorizar aquellos que están más cerca de terminar [14].

2.3.2. Primitivas de sincronización

Sincronización dentro de un bloque

La forma más habitual de sincronizar threads en CUDA es con la función `syncthreads()` [11]. Esta instrucción actúa como una barrera: todos los threads de un bloque deben llegar a esa línea del código antes de que cualquiera de ellos pueda avanzar. Es especialmente útil cuando los threads están cooperando en tareas como cargar datos en la memoria compartida, realizar operaciones por partes, o construir

resultados intermedios antes de continuar con la siguiente fase del algoritmo [15]. Si un thread no alcanza la barrera (por ejemplo, debido a un condicional mal estructurado), el resto se quedará esperando indefinidamente, provocando un interbloqueo (*deadlock*) en el kernel. Por eso, es importante que todas las rutas de ejecución posibles dentro del bloque pasen por la misma llamada a `syncthreads()` [11].

Sincronización dentro de un warp

A nivel de warp, no es necesario sincronizar manualmente: todos los threads avanzan juntos de forma implícita, ya que ejecutan la misma instrucción al mismo tiempo [12]. Esto significa que si estamos seguros de que todos los threads implicados están dentro del mismo warp, podemos omitir la barrera de sincronización sin riesgo [12]. En CUDA más recientes (a partir de Volta), también existen instrucciones como `syncwarp()` que permiten sincronizar únicamente un subconjunto de threads dentro del warp [14], aunque su uso es menos común si no se trabaja con patrones de ejecución muy específicos [14].

Sincronización entre bloques

Una de las grandes limitaciones del modelo de ejecución en CUDA es que no se puede sincronizar directamente entre bloques [11]. No existe una primitiva como `syncthreads()` pero a nivel global. Esto es porque los bloques pueden ejecutarse en cualquier orden, e incluso en paralelo, sin que la GPU garantice cuándo o dónde se ejecuta cada uno [11].

Si realmente es necesario sincronizar entre bloques, la única opción es dividir el trabajo en múltiples kernels: se lanza un primer kernel, se espera a que termine y después se lanza el siguiente [11].

Operaciones atómicas

Además de las barreras de sincronización, CUDA también ofrece operaciones atómicas que permiten acceder y modificar una posición de memoria de forma segura, incluso si varios threads intentan hacerlo a la vez [11]. Estas operaciones garantizan que solo un thread a la vez pueda leer-modificar-escribir un valor concreto [11].

2.4. Arquitectura y operación de los Tensor Cores

Con la llegada de arquitecturas como Volta, Turing y Ampere, NVIDIA introdujo una nueva unidad de cómputo especializada dentro de los multiprocesadores: los **Tensor Cores** [17]. Estos núcleos están diseñados específicamente para acelerar operaciones de álgebra lineal, sobre todo las multiplicaciones y sumas de matrices, que son muy comunes en tareas de inteligencia artificial, simulaciones físicas y cómputo científico en general [17].

2.4.1. Integración de los Tensor Cores con el SM

Como muestra la figura 3 Cada SM moderno incluye un conjunto de Tensor Cores integrados directamente en su arquitectura [17]. Esto significa que, además de las unidades tradicionales de cómputo, cada SM cuenta con lógica dedicada a realizar operaciones matriciales a gran velocidad.

La cantidad de Tensor Cores por SM depende de la arquitectura: por ejemplo, en Volta hay 8 por SM, mientras que en Ampere esta cifra puede subir hasta 16 o más [18]. Estos núcleos trabajan en paralelo con el resto del hardware y se activan únicamente cuando el código utiliza instrucciones específicas de matriz, como las de la API `wmma` (warp matrix multiply-accumulate) de CUDA [19].

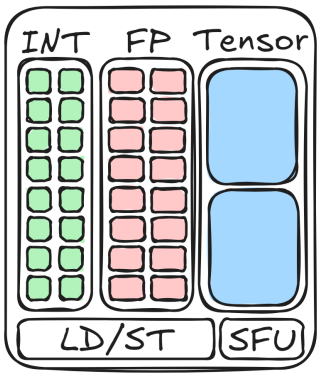

Figura 3: Integración de los Tensor Cores en el SM

2.4.2. Mixed-Precision Matrix-Multiply-Accumulate Pipelines

El núcleo de su funcionamiento es la operación **MMA** (Matrix Multiply-Accumulate). Esta operación realiza, en un único paso, la multiplicación de dos matrices y la suma del resultado con una tercera matriz:

$$D = AB + C \tag{1}$$

Una de las claves del rendimiento de los Tensor Cores es que trabajan con formatos de precisión reducida [17]. Están optimizados para formatos como `FP16` (punto flotante de 16 bits), `bfloat16`, `TF32`, `INT8` o incluso `INT4`, dependiendo de la arquitectura [18]. Esta reducción en la precisión permite mayor paralelismo a nivel de hardware, ya que se pueden ejecutar más operaciones en el mismo espacio físico y coste energético [18].

En arquitecturas recientes como Ampere, se introdujo el soporte para `TF32`, un nuevo formato que intenta ofrecer una precisión parecida al `FP32`, pero con un rendimiento mucho mayor [18]. Esto permite usar Tensor Cores incluso para cargas de trabajo que antes necesitaban precisión completa [18].

2.5. Flujo de datos y memoria en los Tensor Cores

Para que los Tensor Cores puedan rendir al máximo, no solo es importante lanzar las operaciones de forma correcta, sino también alimentar esos núcleos con los datos adecuados, en el momento preciso [19]. Esto implica entender cómo funciona el acceso a memoria y cuál es el recorrido que siguen las matrices desde que se declaran hasta que se procesan [19]. En el anexo A puede verse un ejemplo de un kernel Tensor.

Carga y almacenamiento de datos

El flujo de datos en los Tensor Cores no empieza directamente en ellos [19]. En realidad, los datos se cargan desde memoria global o memoria compartida por parte de los threads de un warp, que luego los almacenan en estructuras intermedias

llamadas **fragmentos** [19]. Estos fragmentos se definen usando las primitivas de la API `wmma` y representan porciones pequeñas de matrices (por ejemplo, un bloque de 16x16) que luego se van a multiplicar [19].

Interacción con la memoria compartida

En kernels que hacen uso intensivo de Tensor Cores, es habitual ver que los datos se copian primero desde memoria global a memoria compartida, para luego ser reutilizados por varios threads dentro del mismo bloque [19]. Esta estrategia permite reducir significativamente el número de accesos a memoria global, que son mucho más lentos y pueden acabar saturando el ancho de banda disponible.

Además, los Tensor Cores requieren que los datos estén correctamente alineados y organizados para poder aprovechar el acceso en bloque. Si no se respetan estos patrones, el hardware no puede realizar la carga de forma eficiente, y se pierde rendimiento aunque se estén utilizando los núcleos especializados [19].

Ejecución de la operación y escritura del resultado

Una vez que cada thread ha cargado su parte correspondiente de las matrices A y B (y opcionalmente de la C, si se quiere hacer acumulación), se lanza la operación `mma`, que es ejecutada directamente en los Tensor Cores [17]. Este cálculo ocurre dentro del SM, sin necesidad de acceder a otras unidades de cómputo ni a memoria externa.

2.6. Rendimiento

El rendimiento que se puede alcanzar con Tensor Cores es muy superior al de las unidades de ejecución tradicionales [17]. Esto se debe a que son capaces de procesar operaciones matriciales completas en pocos ciclos, en lugar de tener que descomponerlas en muchas instrucciones escalares. Si los datos están bien organizados y se aprovecha su uso correctamente, se puede llegar a multiplicar por 10 o incluso más el throughput de una operación de multiplicación y acumulación respecto a la versión sin Tensor Cores [17].

Este rendimiento extra no es automático: hay que adaptar los kernels y usar primitivas específicas como `wmma` para lanzar las operaciones. Además, el programador tiene que tener cuidado con el tipo de datos, ya que los Tensor Cores funcionan mejor (y a veces exclusivamente) con formatos de precisión reducida como FP16, BF16, INT8 o TF32, dependiendo de la arquitectura [19].

2.6.1. Integración

A nivel de integración, NVIDIA ha ido aumentando progresivamente tanto el número como las capacidades de los Tensor Cores en cada nueva generación [17]. En Volta, por ejemplo, cada SM tenía una pequeña cantidad de estos núcleos, y estaban muy ligados al uso de FP16. En Turing y Ampere, se introdujo soporte para más formatos y se mejoró la eficiencia [17]. Con Hopper y Ada, ya no solo se ha aumentado su número, sino que también se ha refinado la forma en que interactúan

con los otros elementos del SM, como los pipelines tradicionales, los registros y la memoria compartida.

Otra cosa importante es que NVIDIA ha optimizado también su software (como cuBLAS, cuDNN o TensorRT) para que, cuando se detecta una GPU con Tensor Cores disponibles, se usen automáticamente para acelerar operaciones comunes [17]. Esto permite que incluso sin escribir código a bajo nivel, las aplicaciones que dependen de estas bibliotecas puedan beneficiarse del hardware especializado.

2.6.2. Diferencias en el profiling entre Tensor Cores y CUDA Cores

Cuando se analiza el rendimiento de un kernel, no se puede medir igual el trabajo realizado por los CUDA Cores que el de los Tensor Cores [20]. Aunque ambos forman parte del SM y ejecutan instrucciones, su naturaleza y el tipo de operaciones que procesan son completamente distintos, y eso se refleja en cómo se hace el profiling [20].

Los CUDA Cores ejecutan instrucciones como: sumas, multiplicaciones, accesos a memoria, comparaciones, etc [20]. Estas operaciones están bien reflejadas en herramientas como Nsight Compute o nvprof, y es fácil interpretar métricas como el número de instrucciones emitidas, ciclos por instrucción o utilización del SM. Cada thread ejecuta su propio conjunto de instrucciones, y el perfilado está bastante bien segmentado por warps, funciones, bloques, etc [20].

Sin embargo, los Tensor Cores no ejecutan este tipo de instrucciones comunes, sino que funcionan a partir de instrucciones especializadas como `wmma/mma`. Por tanto, cuando usamos Tensor Cores, la métrica ya no es tanto "cuántas instrucciones se han ejecutado", sino cuántas operaciones de matriz se han lanzado, qué formatos de datos se han utilizado, y si el pipeline de tensor está saturado o infrautilizado [21].

Otro punto importante es que las instrucciones para los Tensor Cores no se ejecutan de forma transparente como las instrucciones regulares. Se tiene que saber si el compilador realmente las ha generado o si, por algún motivo, ha revertido a usar código escalar convencional [21].

Implicaciones para el Programador en CUDA

Para aprovechar al máximo los SMs, el código CUDA debe diseñarse siguiendo ciertos principios. Es crucial organizar los accesos a memoria global de forma que los threads de un warp accedan a posiciones contiguas, facilitando operaciones coalescentes [15, 16]. La memoria compartida debe usarse como cache intermedia para datos reutilizados, reduciendo la dependencia de la memoria global [15]. Además, las condiciones lógicas deben estructurarse para afectar a warps completos, evitando divergencias dentro de un mismo grupo [11].

3. Metodología

3.1. Preparación del entorno experimental

3.1.1. Hardware y Software utilizados

Para llevar a cabo la experimentación, se ha optado por configurar una máquina virtual con Ubuntu 22.04 sobre WSL 2 en un entorno Windows. Esta decisión responde principalmente a la necesidad de trabajar en un sistema Linux, dado que muchas de las herramientas y librerías del ecosistema CUDA presentan una integración mucho más fluida en este entorno, sin renunciar por ello a la comodidad y versatilidad de Windows 11. La máquina host cuenta con una *NVIDIA GeForce RTX 4070*, una tarjeta gráfica de gama media-alta basada en la arquitectura *Ada Lovelace*. Concretamente, dispone de 5888 núcleos CUDA y 184 Tensor Cores de cuarta generación, junto con 12 GB de memoria GDDR6 y un ancho de banda de 504 GB/s.

El entorno de desarrollo está basado en *CUDA Toolkit 12.4*, acompañado de librerías estándar como `iostream`, `stdio.h` y `time.h`, necesarias para la entrada/-salida y la medición de tiempos de ejecución. También se han utilizado cabeceras propias de CUDA como `cuda_runtime.h`, `mma.h` y `cuda_fp16.h`. Esta última es especialmente relevante, ya que buena parte del trabajo con Tensor Cores requiere operar en formatos reducidos como FP16. La cabecera `mma.h` permite acceder a las primitivas `wmma`, diseñadas específicamente para invocar operaciones sobre Tensor Cores a través de matrices tipo *fragment*, que son gestionadas por el compilador y mapeadas directamente sobre el hardware especializado.

Además del entorno de compilación y ejecución, se ha hecho uso de herramientas de depuración y *profiling*. Entre ellas, `cuda-gdb` permite depurar los kernels directamente desde la consola de Ubuntu, necesario para que *NVIDIA Nsight Compute* y *Nsight Systems* funcionen de manera correcta. En particular, *Nsight Compute* permite observar el uso de los distintos *pipelines*, los *stalls* por dependencias de datos, la ocupación de registros y memoria compartida, así como métricas específicas relacionadas con los Tensor Cores. Desde el sistema Windows, se ha lanzado *Nsight Compute* con conexión directa a WSL 2, comunicándose vía ssh.

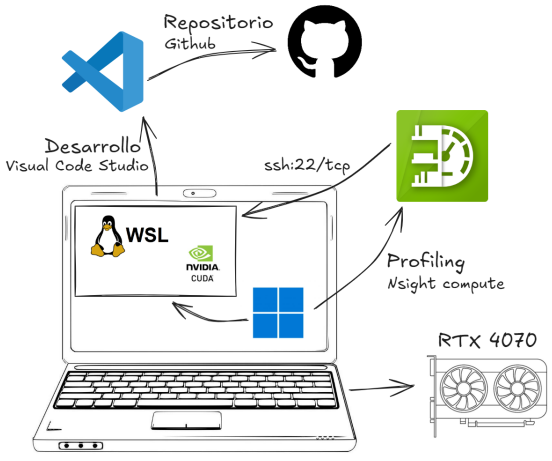

Figura 4: Diagrama de la configuración del entorno

Ha sido necesario validar que los drivers de NVIDIA estaban correctamente expuestos a través de WSL, comprobar que la versión del toolkit y los *paths* de las librerías estaban bien definidos, y asegurarse de que las herramientas como *Nsight* podían acceder correctamente a la GPU desde el subsistema Linux. Además, ha sido necesario modificar ciertas configuraciones internas de la GPU para permitir que los contadores de rendimiento fueran accesibles desde WSL.

3.1.2. Objetivos del benchmark

El objetivo principal del proyecto ha sido implementar y comparar dos versiones de un filtro gaussiano 2D: una basada en CUDA Cores y otra que explota directamente los Tensor Cores mediante instrucciones `wmma`. Programar estas unidades en bajo nivel, en vez de usar librerías de alto nivel, nos permite examinar de forma precisa las limitaciones y ventajas de este hardware especializado.

Más allá de la funcionalidad básica (que ambos filtros apliquen correctamente la convolución), el interés se centró en analizar su comportamiento a nivel de rendimiento: tiempos de ejecución, grado de paralelismo, distribución de instrucciones en los pipelines, y aprovechamiento real de los recursos de la GPU. El benchmarking no fue solo una simple medición de tiempos, sino un estudio de lo que ocurre dentro del SM: cuántas instrucciones se lanzan realmente, ciclos en espera de datos, y proporción entre operaciones aritméticas y transferencias de memoria.

Este contraste permite identificar costes clave: preparación de fragmentos para Tensor Cores, restricciones de tamaño de kernel, y beneficios cuando los datos están alineados correctamente. También revela cómo el uso explícito de Tensor Cores altera el perfil del programa, no solo en instrucciones ejecutadas, sino en cómo compiten por recursos del SM. Esto es crítico al escalar a kernels mayores o volúmenes de datos más grandes, donde las ineficiencias se amplifican.

3.1.3. Datos de entrada y tamaños de filtro

Los filtros gaussianos utilizados fueron de tamaño 3×3, 9×9, 27×27, 81×81 y 243×243, con el objetivo de observar cómo varía el rendimiento conforme crece la carga computacional. En cuanto a las imágenes, se han utilizado resoluciones variables, partiendo de Full HD (1920x1080) hasta tamaños superiores al 4K, lo cual permite analizar también cómo se comportan las implementaciones ante entradas de mayor escala y complejidad.

Cabe destacar que los Tensor Cores no operan sobre cualquier disposición arbitraria de datos, sino que requieren que las matrices involucradas respeten unas dimensiones específicas según el tipo de datos. En el caso de FP16 (tipo `__half`), por ejemplo, NVIDIA permite operar con bloques de tamaño $16 \times 16 \times 16$, $32 \times 8 \times 16$ y $8 \times 32 \times 16$, tanto para acumulación en precisión simple (`float`) como en media precisión (`__half`). Estas combinaciones están definidas por la arquitectura y el soporte del hardware subyacente, como se puede ver en la documentación oficial de CUDA:

Cuadro 1: Combinaciones soportadas por Tensor Cores en términos de tipos de datos y tamaños de matriz.

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
__half	__half	float	16x16x16
__half	__half	float	32x8x16
__half	__half	float	8x32x16
__half	__half	__half	16x16x16
__half	__half	__half	32x8x16
__half	__half	__half	8x32x16
unsigned char	unsigned char	int	16x16x16
unsigned char	unsigned char	int	32x8x16
unsigned char	unsigned char	int	8x32x16
signed char	signed char	int	16x16x16
signed char	signed char	int	32x8x16

En este trabajo se ha optado por usar una configuración TENSOR de tamaño $32 \times 8 \times 16$, ya que, tras realizar varias pruebas, se observó que esta configuración ofrece un buen equilibrio entre flexibilidad y rendimiento. Al emplear un número reducido de filtros, este tamaño permite una mejor ocupación de los recursos que la utilización de matrices cuadradas, evitando ineficiencias por fragmentación del trabajo. Además, al estructurar los datos con esta disposición, se mejora la localidad espacial en memoria, lo que reduce los accesos dispersos y facilita el aprovechamiento de los mecanismos de cache del propio SM.

Esto obliga, por tanto, a reformular la operación de convolución clásica (que originalmente trabaja con fragmentos desplazando sobre una imagen) como una multiplicación matricial. Para ello, la imagen de entrada debe transformarse en un conjunto de submatrices alineadas con los bloques esperados por los Tensor Cores, lo cual se hace cargando los datos como *fragments* a través del API WMMA. De esta forma, es posible delegar la computación directamente a los Tensor Cores, aprovechando su paralelismo masivo y alta eficiencia energética.

3.1.4. Profiling con NVIDIA Nsight

Una herramienta clave a lo largo de este trabajo ha sido *Nsight Compute*, el profiler oficial de NVIDIA que permite obtener una visión extremadamente detallada del comportamiento interno de cada kernel. A diferencia de herramientas más generales como `nvprof` o incluso Nsight Systems, *Nsight Compute* se orienta a ofrecer información específica a nivel de ejecución de threads, warps, y SMs, algo especialmente útil cuando se trabaja con Tensor Cores o instrucciones WMMA.

Durante el benchmarking, no solo se midieron tiempos de ejecución, sino que se fue mucho más allá: se inspeccionó el número exacto de instrucciones lanzadas, tanto aritméticas como de carga/almacenamiento, incluyendo también operaciones específicas sobre Tensor Cores. Esto permitió observar de manera precisa cuántos FLOPs reales se estaban ejecutando en cada kernel, cuántas instrucciones estaban siendo predicadas y descartadas, y si se estaban utilizando bien las unidades vectoriales y especializadas.

También fue posible medir el porcentaje de ciclos que el SM dedicaba a cada tipo de operación, lo que permitió ver si, por ejemplo, había cuellos de botella en las instrucciones de carga global, o si los Tensor Cores estaban infrautilizados debido a problemas de alineamiento o formatos de datos incompatibles. Además, el profiler permitió ver claramente la ocupación real del SM, tanto a nivel de warps activos como de ocupación teórica, teniendo en cuenta limitaciones por registros, memoria compartida y bloques activos por SM.

Otro punto clave es que *Nsight Compute* permite comparar kernels entre sí directamente en la misma interfaz, lo que ha sido especialmente útil al evaluar distintas variantes del mismo algoritmo o configuraciones internas del lanzador.

También ha sido necesario ajustar la configuración interna de la GPU para que ciertos contadores de bajo nivel (especialmente los relacionados con registros e instrucciones en Tensor Cores) estuviesen accesibles desde entornos WSL2. Sin estos ajustes, muchos contadores aparecen simplemente como "not collected", ya que por defecto el acceso a ciertos subsistemas hardware está limitado en virtualización.

3.2. Filtro Gaussiano

Como caso de estudio para analizar el rendimiento de los Tensor Cores en operaciones reales, se ha optado por implementar un filtro gaussiano sobre imágenes. Esta elección tiene un propósito, ya que el filtrado gaussiano es una de las operaciones más utilizadas en procesamiento de imagen, tanto en tareas de suavizado como en la etapa previa a muchos algoritmos de visión por computador, como detección de bordes, reducción de ruido o antialiasing de imagen.

El filtro gaussiano consiste, esencialmente, en aplicar una convolución entre la imagen de entrada y una matriz de pesos cuyos valores siguen la distribución gaussiana bidimensional. Esto implica que los valores centrales de la matriz tienen mayor peso que los de los bordes, lo que permite suavizar una región sin perder completamente la estructura local. A nivel de implementación, cada píxel de la imagen de salida se calcula como una combinación ponderada de los píxeles vecinos, donde las ponderaciones vienen dadas por los valores de la matriz gaussiana.

3.2.1. ¿Qué es una convolución?

A nivel conceptual, una convolución es una operación matemática que permite extraer información local de una señal, aplicando una función que se desliza por todo su tamaño. En el contexto del procesamiento de imágenes, esta función se representa como una matriz (llamado filtro) que se aplica sobre cada píxel considerando su vecindario. El resultado es una nueva imagen donde cada valor depende no solo del píxel original, sino también de su entorno más próximo.

La idea principal detrás de una convolución es recorrer toda la imagen aplicando, en cada posición, una suma ponderada de los valores del vecindario local. Para ello, se multiplica cada elemento del kernel por el píxel correspondiente de la ventana actual, y luego se suman todos los productos. Esa suma es el nuevo valor del píxel en la imagen de salida.

Lo que distingue a la convolución de una simple multiplicación o media local es que el kernel puede tener una forma y unos valores determinados que modulan

el tipo de operación: si es un filtro gaussiano, suaviza la imagen; si es un filtro de Sobel, puede detectar bordes; si se usan kernels más complejos, se pueden extraer patrones o texturas concretas. En ese sentido, la convolución actúa como una herramienta general para transformar la imagen según la función que queramos aplicar localmente, por lo que el código realizado es compatible con una gran variedad de casuísticas y necesidades particulares.

Desde un punto de vista matemático, si denotamos la imagen como una función discreta $I(x, y)$ y el kernel como $K(i, j)$, entonces la convolución $C(x, y)$ en una posición (x, y) se define como:

$$C(x, y) = \sum_i \sum_j K(i, j) \cdot I(x + i, y + j)$$

Esta definición puede ajustarse ligeramente en función de si se usa convolución clásica o correlación cruzada (cross-correlation), pero la idea es la misma: aplicar una función local desplazándose por la imagen. Usando determinados tipos de filtros se pueden obtener resultados como estos:

Figura 5: Imagen original

Figura 6: Imagen difuminada

3.2.2. Características del filtro

Una particularidad importante del filtro es que la matriz del kernel debe tener dimensiones impares. Esto es fundamental porque el valor central de la matriz debe corresponder con el píxel sobre el que se está calculando la convolución. Si el tamaño fuera par, no existiría un centro definido y el desplazamiento relativo de los vecinos perdería sentido. Por tanto, tamaños como 3×3, 5×5, 27×27, etc., son válidos, pero no lo serían matrices como 4×4 o 8×8. Para solventar esta casuística, internamente estas matrices pares se transforman a su matriz impar de mayor tamaño más próxima ($4 \times 4 = 5 \times 5$). Si se requiriese realizar obligatoriamente un cálculo con una matriz de tamaño par, se puede obtener creando una matriz impar y completando de 0 la fila y columna sobrante.

Además, es necesario que la suma de todos los elementos del kernel sea exactamente igual a 1. Esto asegura que, en zonas de la imagen donde todos los valores son iguales (por ejemplo, una región constante), el valor de salida sea el mismo,

evitando que el filtro introduzca un sesgo hacia valores mayores o menores. En la práctica, esto se consigue generando la matriz gaussiana a partir de la fórmula de la distribución normal y normalizándola al final, dividiendo todos sus elementos entre la suma total.

3.2.3. Transformando la convolución en un producto matricial

Para poder aprovechar los Tensor Cores, ha sido necesario reformular el algoritmo clásico de convolución como una operación matricial. Esto se debe a que los Tensor Cores no están diseñados para aplicar directamente filtros sobre una imagen, sino para ejecutar productos de matrices densas con una eficiencia extremadamente alta. Por tanto, el objetivo fue convertir la operación de convolución en una multiplicación entre dos matrices, lo que no solo es compatible con los Tensor Cores, sino que además permite un mayor aprovechamiento del paralelismo de la GPU.

3.3. CUDA-Core Version

3.3.1. Implementación de la convolución con CUDA Cores

Una vez realizada la transformación matricial del problema, se ha llevado a cabo una implementación completa usando únicamente los CUDA Cores. Esta versión sirve como baseline de comparación frente a la posterior implementación con Tensor Cores, permitiendo analizar de forma clara las diferencias de rendimiento.

Lanzamiento de threads y cálculo del píxel correspondiente

El kernel `Cuda` sigue una estrategia de paralelización en la que cada thread CUDA es responsable de calcular el resultado de aplicar uno o varios filtros sobre un único píxel y canal de la imagen.

Primero se calcula el índice global del thread en base al bloque y al thread dentro del bloque:

```
int pos = (blockIdx.x + initialBlock) * blockDim.x +
    threadIdx.x;
```

Este índice se traduce después a un triplete $(x, y, canal)$ mediante una operación de desenrollado, teniendo en cuenta que los canales se tratan como dimensión adicional. Ver anexo B.

Este cálculo asegura una distribución regular y predecible de los threads en la imagen, permitiendo que cada warp se enfoque en una región concreta. También se incorpora un parámetro opcional `initialBlock`, que permite distribuir diferentes regiones de la imagen en diferentes lanzamientos del kernel, útil para dividir el trabajo o realizar pruebas independientes.

Aplicación de los filtros sobre los píxeles Una vez identificado el píxel a procesar, el thread entra en un bucle que recorre todos los filtros a aplicar sobre esa posición. Para cada filtro:

- Se inicializa un acumulador `blurredPixel` en tipo `half`, que irá sumando las contribuciones de cada vecino del píxel actual.

- Se recorre el filtro en dos dimensiones (alto y ancho), con desplazamiento relativo respecto al píxel central.
- Se aplica un **clamping** de coordenadas: si un píxel vecino cae fuera de los límites de la imagen, se sustituye por el valor más cercano dentro de la misma. Esto garantiza que no se acceda fuera de los límites del buffer, pero también introduce bordes suavizados (no padding con ceros).
- Se multiplica el valor del píxel vecino por el coeficiente correspondiente del filtro, y se acumula.

El índice del filtro se calcula linealizando su posición 2D en un array plano, con un desplazamiento adicional cuando hay múltiples filtros:

```
int filterIndex = (filterY + halfFilterWidth) * filterWidth +
    (filterX + halfFilterWidth);
blurredPixel += pixel * filter[filterIndex + i * filterSize];
```

Al finalizar el procesamiento del filtro i, el resultado se almacena en la posición correspondiente de la imagen de salida:

```
blurredImage[((y * width + x) * channels) + canal + (i * width
    * height * channels)] = (uint8_t)__half2float(blurredPixel);
```

Se convierte explícitamente el resultado `half` a `float`, y de ahí a `uint8_t`, dado que la imagen de salida debe tener formato estándar de 8 bits por canal. Este paso puede implicar cierta pérdida de precisión, pero es inevitable si se desea conservar el formato de imagen final tradicional.

Características numéricas y precisión Aunque esta versión no explota las instrucciones especializadas de los Tensor Cores, sí se beneficia del uso del tipo `half`. La elección de FP16 reduce el ancho de banda y el uso de registros, permitiendo más threads activos por SM (mayor ocupación), aunque a costa de menor precisión numérica. Para operaciones de filtrado, sin embargo, esta precisión suele ser más que suficiente, y la reducción de consumo de recursos lo compensa ampliamente.

Accesos a memoria y patrones de eficiencia Cada thread accede a múltiples regiones de la imagen de entrada, pero no se emplea ningún tipo de cache manual ni uso compartido de memoria (shared memory). Esto implica que los accesos a memoria global pueden ser no coalescentes, especialmente cuando el filtro es grande o el warp está procesando píxeles que no están contiguos en memoria.

Esta es, sin duda, una de las razones por las que esta implementación es más lenta que su versión optimizada con Tensor Cores. Sin embargo, es una buena forma de visualizar claramente el impacto del patrón de acceso a memoria y del uso de recursos de hardware más especializados.

Flexibilidad y escalabilidad Este kernel está diseñado para ser totalmente general: soporta múltiples canales (por ejemplo, imágenes RGB, argb, escala de grises, multiespectrales, hiperespectrales...), un número arbitrario de filtros y filtros de cualquier tamaño impar. Además, la estructura modular permite lanzar el kernel por

bloques, particionando la imagen en regiones para realizar pruebas independientes, medir tiempos parciales o aplicar diferentes configuraciones.

3.3.2. Configuración de bloques y threads

La distribución de threads CUDA se organiza mediante una jerarquía de bloques y warps, diseñada para procesar exhaustivamente todos los píxeles y canales de la imagen. Cada bloque contiene un número fijo de warps determinado por el parámetro `NUM_WARPS`, donde cada warp agrupa 32 threads que ejecutan instrucciones de forma sincronizada. El tamaño unidimensional del bloque se calcula como:

```
int threadsPerBlock = NUM\_WARPS * 32;
dim3 blockDim(threadsPerBlock);
```

La dimensión del grid se determina para cubrir el trabajo total (píxeles × canales), aplicando la fórmula:

```
int gridDim((width * height) / ((threadsPerBlock) /
    channels) + 1);
```

Esta expresión equivale a $\left\lceil \frac{\texttt{width} \times \texttt{height} \times \texttt{channels}}{\texttt{threadsPerBlock}} \right\rceil + 1$, garantizando que se asignen bloques suficientes para procesar todos los elementos de la imagen. El término `threadsPerBlock / channels` representa la cantidad de píxeles completos que puede manejar un bloque, considerando que cada thread procesa exclusivamente un canal de un píxel específico.

El `+1` asegura el manejo de casos residuales cuando la división no es exacta, creando un bloque adicional para los elementos sobrantes. Durante la ejecución del kernel, los threads que exceden el rango válido ($\texttt{pos} \geq \texttt{width} \times \texttt{height} \times \texttt{channels}$) se desactivan automáticamente para evitar accesos inválidos a memoria.

La posición global de cada thread se calcula mediante:

```
int pos = (blockIdx.x + initialBlock) * blockDim.x +
    threadIdx.x;
```

donde `initialBlock` permite subdividir el procesamiento en múltiples lanzamientos, facilitando pruebas modulares y análisis de rendimiento parcial. Este índice lineal `pos` se transforma posteriormente en coordenadas espaciales (x, y) y de canal mediante operaciones de descomposición modular, como se detalló en la sección anterior.

Para una imagen Full HD (1920×1080) RGBa (4 canales) con `NUM_WARPS=4`, se generan 128 threads por bloque (4 warps) y 64800 bloques en la grid. Esta configuración procesa 8294400 elementos (píxeles × canales), donde los threads del último bloque no serán ejecutados debido a que la división es exacta.

3.4. Tensor Core Version: Más a fondo

3.4.1. Implementación de la convolución con Tensor Cores

En la versión que aprovecha los Tensor Cores, la principal diferencia con la implementación basada en CUDA Cores consiste en transformar la operación de convolución en un producto matricial, de modo que la GPU pueda explotar sus unidades especializadas al máximo. Para ello, es necesario reorganizar los datos de

la imagen y del filtro de tal forma que encajen exactamente en los tamaños que admiten los Tensor Cores, que en nuestra arquitectura están restringidos a bloques de 16×32 (16 elementos del vecindario por 32 píxeles simultáneamente).

La idea central es aplanar el vecindario de cada píxel en una fila y colocar cada filtro completo en una columna correspondiente, así como se muestra en la figura 7. De este modo, la multiplicación matricial entre esas dos estructuras reproduce exactamente el efecto de una convolución tradicional: para cada píxel, se realiza un producto punto entre la fila que contiene sus vecinos aplanados y la columna con los coeficientes del filtro. En esencia, colocar los vecinos en filas y los filtros en columnas simula la operación estándar de desplazar el kernel sobre la imagen y sumar las multiplicaciones peso–valor.

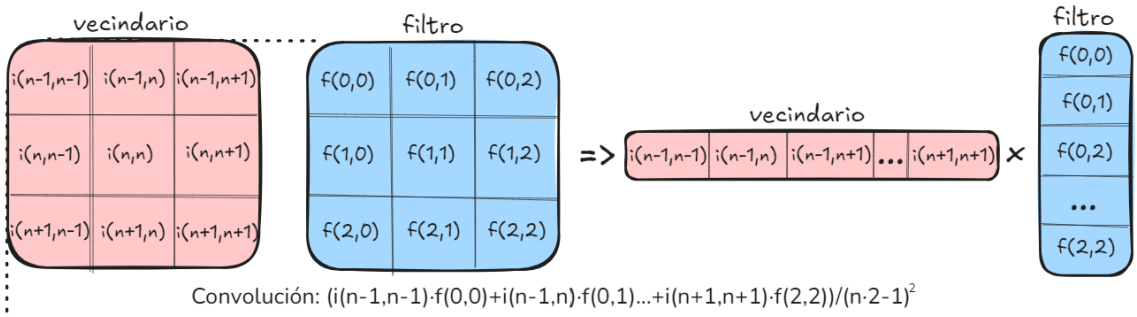

Figura 7: Representación Im2Col

Para que esto encaje con los Tensor Cores, hemos escogido un tamaño de fragmento de 16×32. El número 32 corresponde exactamente al tamaño de un warp, de modo que cada thread en el warp se responsabiliza de procesar el canal de un pixel distinto de forma simultánea. Al usar 32 threads, se garantiza que no hay threads inactivos: cada thread se encarga de cargar los valores correspondientes en la matriz, que posteriormente será transmitida a los tensor cores. El tamaño 16 responde a la restricción de máxima longitud del vecindario que puede procesarse por iteración; cuando el filtro tiene más de 16 vecinos (por ejemplo, un 5×5 o un 9×9), se fragmenta en varias etapas de 16 posiciones, acumulando parciales en un fragmento de acumulación.

Dentro del kernel, cada thread primero calcula su posición global $(x, y, canal)$ en la imagen, de manera muy similar a la versión de CUDA Cores, pero adaptada para que cada warp coordine 32 valores consecutivos en una misma fila de la imagen original, como se muestra en la figura 8. A partir de ahí, se inicia un bucle que recorre el filtro en bloques de 16 elementos (WMMA_K = 16). En cada iteración, se rellenan dos buffers locales alineados: la `localMatrix` (o "Input fragment"), que recoge los 16 vecinos correspondientes a ese grupo de filtros, y la `filterMatrix`, donde se colocan los 16 coeficientes de cada filtro para todos los filtros activos en paralelo. Una vez cargados, se llama a `wmma::load_matrix_sync` para incorporar ambos fragmentos a los registros internos de los Tensor Cores.

A continuación, la instrucción `wmma::mma_sync` multiplica simultáneamente los 16 valores de cada uno de los 32 píxeles por sus correspondientes coeficientes en el filtro, produciendo productos parciales que se suman en el fragmento de acumulador. Como los filtros suelen ser más grandes que 16 posiciones, el bucle se repite tantas

veces como sea necesario (por ejemplo, 81/16 iteraciones en el caso de un filtro 9×9), y cada iteración va reduciendo el número de valores pendientes. Esta estrategia de "trocear y acumular" es esencial para ajustarse a la restricción de matriz de los Tensor Cores y, al mismo tiempo, simular la convolución completa. Cuando todas las partes del filtro se han procesado, el fragmento acumulado contiene la suma total del producto para cada uno de los 32 canales (8 pixeles en argb), que corresponde a los valores filtrados de la imagen.

Por último, una vez completadas todas las iteraciones, el fragmento de acumulación se almacena de vuelta en memoria compartida o global mediante `wmma::store_matrix_sync`. Desde allí, cada thread toma el valor final correspondiente a su píxel y canal, lo convierte de `float` a `uint8_t` y lo escribe en el buffer de salida `blurredImage`. De este modo, el resultado final de la convolución se genera de manera paralela para todos los píxeles agrupados en cada warp, aprovechando al máximo la capacidad de procesamiento en matrices densas de los Tensor Cores.

Al alinear los datos de modo que cada warp trabaje con 32 píxeles de forma simultánea, se garantiza que no hay threads ocioso dentro del warp, maximizando la ocupación del hardware.

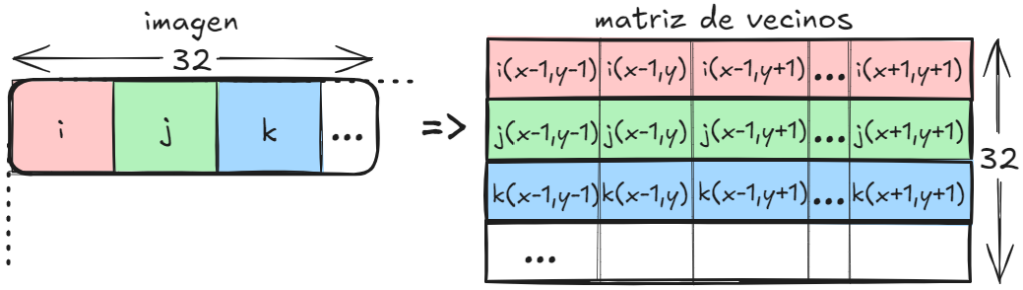

Figura 8: Carga de múltiples vecindarios en la misma matriz

3.4.2. Cálculo simultáneo de múltiples filtros

En la versión con Tensor Cores, uno de los grandes avances respecto a la implementación con CUDA Cores es que ya no es necesario iterar sobre cada filtro por separado. Gracias a las propiedades de la multiplicación matricial, podemos incluir varios filtros de una sola vez, organizándolos como columnas contiguas en la `FilterMatrix`, como se muestra en la figura 9. De esta forma, durante la operación `wmma::mma_sync` no solo estamos multiplicando un vecindario de píxel por un filtro, sino que, al mismo tiempo, estamos obteniendo los resultados de ese mismo vecindario frente a ocho filtros distintos.

Concretamente, la `InputMatrix` (o `localMatrix`) se dimensiona como 32×16. Esto significa que cada warp, formado por 32 threads, carga en cada iteración 16 valores del vecindario de cada uno de esos 32 píxeles. Al mismo tiempo, la `FilterMatrix` se construye con 16 filas (las mismas 16 posiciones de vecino procesadas) y 8 columnas: cada columna contiene los coeficientes de un filtro diferente. Cuando ejecutamos la instrucción `wmma::mma_sync`, los Tensor Cores multiplican estas matrices de tamaño 32×16 por 16×8, produciendo una `OutputMatrix` de dimensiones 32×8. En esta matriz resultante, cada fila corresponde a un píxel distinto y cada columna corresponde al valor filtrado por un filtro distinto.

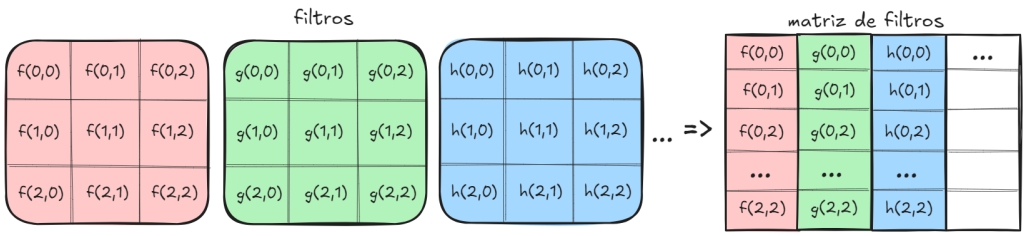

Figura 9: Carga de múltiples filtros en la misma matriz

De este modo, los ocho filtros se aplican en paralelo a los mismos 32 píxeles. En lugar de ejecutar un bucle que recorra los filtros uno a uno, logramos que, en cada llamada a `mma_sync`, los Tensor Cores computen simultáneamente los ocho productos punto: uno por cada combinación de píxel y filtro. Esto no solo reduce radicalmente el número de iteraciones necesarias, sino que también maximiza la ocupación de los pipelines de matriz del hardware, ya que cada warp está completamente ocupado procesando un bloque compacto de datos.

Finalmente, una vez acumulados todos los trozos parciales (en caso de que el filtro supere los 16 vecinos), el fragmento acumulador de tamaño 32×8 contiene el resultado final de la convolución para los 32 píxeles y los ocho filtros. Cada thread extrae entonces el valor correspondiente a su propio píxel y canal, lo convierte de `float` a `uint8_t` y lo escribe en la imagen de salida. De esta manera, organizando los vecinos en filas y los filtros en columnas, simulamos de forma directa la convolución tradicional, pero aprovechando por completo el paralelismo masivo y las unidades especializadas de los Tensor Cores.

Filtros de distinto tamaño

Para manejar filtros de distintos tamaños dentro de la misma `FilterMatrix`, se parsean las matrices más pequeñas y se rellenan con ceros hasta igualar las dimensiones del filtro más grande. De este modo, cuando las posiciones se cargan en las matrices intermedias (como `interMatrix` o `localMatrix`), todos los coeficientes quedan perfectamente alineados con la ventana de vecinos correspondiente, independientemente de que el filtro original fuese 3×3, 9×9 o 27×27. Esta técnica de padding con ceros garantiza que las posiciones que cada thread extrae de `shared` siempre coincidan con el filtro correcto, simplificando la lógica de carga.

3.4.3. Transferencia de datos desde memoria global a fragmentos WMMA

Para que los Tensor Cores puedan operar, primero es necesario cargar los bloques de datos desde la memoria convencional de CUDA (por ejemplo, registros o memoria compartida) en las estructuras especiales que utiliza la API WMMA, llamadas *fragments*. Estas estructuras funcionan como contenedores intermedios en los registros que los propios Tensor Cores leerán directamente.

En el anexo A, la función `load_matrix_sync` actúa como puente entre la memoria de CUDA y los registros que alimentan a los Tensor Cores. Concretamente:

- La primera llamada a `load_matrix_sync(data, localMatrix, WMMA_K)` copia un bloque de datos de tamaño $WMMA\_M \times WMMA\_K$ (en nuestro caso, 32×16) desde la dirección apuntada por `localMatrix` hacia el *fragment* `data`. Aquí, `matrix_a` y el atributo `row_major` indican que los datos están almacenados por filas contiguas en memoria. La constante `WMMA_K` define el stride (column stride) necesario para interpretar correctamente la organización de la matriz en memoria.

- La segunda llamada, `load_matrix_sync(mask, filterMatrix, WMMA_K)`, realiza una operación análoga pero en estilo `matrix_b` y con `col_major`, copiando un bloque de tamaño $WMMA\_K \times WMMA\_N$ (en nuestro caso, 16×8) desde la dirección `filterMatrix` hacia el fragment `mask`. El formato `col_major` asegura que los coeficientes de cada filtro estén alineados por columnas contiguas, lo cual es esencial para que un solo `mma_sync` produzca simultáneamente los valores correspondientes a los diferentes filtros.

En ambos casos, `load_matrix_sync` se encarga de reorganizar los datos si es necesario, alineándolos internamente para que coincidan con la arquitectura física del Tensor Core. Una vez que ambos fragments (`data` y `mask`) están completamente cargados, se invoca `wmma::mma_sync(result, data, mask, result)`. Esta instrucción ejecuta la multiplicación matricial de $WMMA\_M \times WMMA\_K$ por $WMMA\_K \times WMMA\_N$ y acumula el resultado $WMMA\_M \times WMMA\_N$ directamente en el fragment `result`. De este modo, los datos ya no residen en memoria convencional, sino que circulan por los registros especializados que alimentan el pipeline de Tensor Cores, maximizando el paralelismo y minimizando la latencia.

3.5. Aprovechando al máximo la memoria compartida

3.5.1. Memoria compartida en la implementación con Tensor Cores

Para obtener el máximo rendimiento de los Tensor Cores, no basta con organizar la convolución como una serie de multiplicaciones matriciales; también es fundamental minimizar los accesos a memoria global, que son muy costosos en términos de latencia. Por este motivo, en nuestro kernel hemos recurrido a la memoria `shared`, aprovechando que esta memoria se comparte al nivel de bloque y tiene un ancho de banda mucho mayor que la memoria global. El propósito principal consiste en cargar tanto los datos de la imagen (los vecinos de cada píxel) como los coeficientes del filtro en `shared`, de modo que todos los threads de un bloque puedan reutilizar esos datos sin volver a acceder a memoria global en cada iteración.

No obstante, en CUDA no es posible declarar tamaños de memoria `shared` que dependen de variables que solo se resuelven en tiempo de ejecución (por ejemplo, el número de threads por bloque o el número de warps activos). Esta restricción implica que, si queremos reservar memoria `shared` suficiente para albergar todas las matrices intermedias (datos, filtros y acumuladores) debemos calcular el espacio necesario antes de compilar y especificarlo como un valor constante o como parámetro en el lanzamiento del kernel. En nuestro caso, hemos usado un cálculo previo en el host que determina el tamaño total de memoria `shared` que requiere el kernel, y ese valor se pasa al momento de invocar el kernel.

En el anexo C se puede comprobar cómo el cálculo del tamaño de la memoria compartida se ejecuta en el host antes de lanzar el kernel. A continuación, paso a paso, se explica cómo se organiza esa memoria:

En primer lugar, reservamos $(WMMA\_M \times WMMA\_K) \times NUM\_WARPS \times sizeof(half)$ bytes para las matrices `localMatrix`. Cada una de estas matrices, con dimensión 32×16, corresponde a un fragmento de vecinos de píxeles que un warp va a procesar en una iteración de `mma_sync`. Como cada bloque puede contener varios warps (es decir, `NUM_WARPS`), necesitamos un área separada por warp. El tipo `half` (FP16) reduce el consumo de espacio, pero al final este primer término asegura que, para cada warp del bloque, hay una zona contigua de 32×16 valores.

A continuación, se reserva $(WMMA\_N \times WMMA\_K) \times sizeof(half)$ bytes para la `FilterMatrix`. En este caso, solo necesitamos una instancia de la matriz de filtro, pues todos los warps pueden leer sus coeficientes desde la misma región. Esa matriz de tamaño 16×8 (16 filas por 8 columnas) almacena los coeficientes de hasta ocho filtros distintos, y el uso de `half` de nuevo reduce la carga en `memoria compartida`. Al compartir esta única copia entre todos los warps, garantizamos que todos tengan acceso a los mismos coeficientes sin duplicar memoria.

El tercer bloque de memoria se destina a las `resultMatrix`, que guardan resultados intermedios o acumulados antes de escribir el valor final en la imagen de salida. Su tamaño se calcula como $(WMMA\_M \times WMMA\_N) \times NUM\_WARPS \times sizeof(float)$. Esta matriz será almacenada almacenará los resultados en formato float, ya que la funcionalidad de los Tensor Cores así lo permite. Cada warp tiene su propia sección de 32×8 resultados en `float`, y los duplicamos para poder manejar la transición entre iteraciones sin perder datos.

Por último, el cuarto término, $(threadsPerBlock \times 5) \times sizeof(half)$, está reservado para la *matriz intermedia* (`interMatrix`). Esta matriz de tamaño máximo $threadsPerBlock \times 5$ (en nuestro caso $blockDim.x \times 5$) se utiliza para almacenar temporalmente todos los vecinos de los píxeles del bloque antes de reordenarlos en las submatrices de 32×16. El factor 5 surge del peor caso en la sección de cálculo, cuando el filtro necesita abarcar varias filas y se debe guardar información intermedia mientras se recorre la imagen. Con esto garantizamos que haya espacio suficiente para almacenar los vecinos de todos los threads antes de cargar cada fragmento en `localMatrix`.

Una vez calculado `sharedMemorySize`, se pasa al lanzamiento del kernel, de modo que CUDA reserve exactamente ese bloque de `shared` por cada bloque de threads. Dentro del kernel, se declara la memoria compartida:

```
extern __shared__ half sharedMemory[];
```

A partir de ahí, distribuimos las direcciones de memoria para cada warp, usando punteros calculados en tiempo de ejecución. Aunque la memoria compartida es visible para todos los threads del bloque, nuestro diseño se basa en que cada warp ocupe una región específica y no haya solapamientos accidentales. Por ejemplo, las primeras `NUM_WARPS` porciones de tamaño $WMMA\_M \times WMMA\_K$ se destinan a `localMatrix`, y cada warp calcula su propio $offsetLocalMatrix = WMMA\_M \times WMMA\_K \times warpId$, de modo que cada warp trabaja en su zona sin interferir con los demás. De manera similar, la sección que sigue, iniciada en `WMMA_M * WMMA_K * NUM_WARPS`, contiene la `FilterMatrix` y es compartida (offset

único), mientras que el espacio después, de tamaño $(WMMA\_M \times WMMA\_N) \times NUM\_WARPS$, se asigna a `resultMatrix` para cada warp por separado, otra vez usando un `offsetResultMatrix` que depende de `warpId`. Finalmente, la última sección, iniciada en un desplazamiento mayor, se reserva para `interMatrix`, donde cada thread copia sus vecinos antes de reorganizarlos.

Con esta estrategia, aunque todos los threads del bloque comparten la misma región física de `sharedMemory`, cada warp trabaja en su porción privada, calculada dinámicamente mediante aritmética de punteros basada en `warpId` y `threadIdx.x`. De esta manera, evitamos colisiones y nos aseguramos de que cada fragmento de datos fluya hacia los Tensor Cores sin bloqueos ni lecturas inconsistentes.

3.5.2. Reutilización de datos en memoria `shared`

Para maximizar el rendimiento del kernel con Tensor Cores, es esencial evitar accesos repetidos a memoria global. La estrategia que hemos empleado consiste en cargar, en cada iteración, exactamente los valores de vecinos de los píxeles que van a procesar todos los threads del bloque, guardándolos en una región denominada `interMatrix` dentro de la memoria `shared`. De este modo, cada thread accede a esa memoria veloz para leer sus propios datos, en lugar de volver a solicitarlos a la memoria global, mucho más lenta.

La forma en que se organiza la carga de `interMatrix` obedece a dos principios fundamentales: por un lado, que todos los threads del bloque cooperen para traer los datos necesarios de la imagen; y por otro, que esos accesos se realicen de la manera más consecutiva posible para aprovechar la coalescencia. Para ello, partimos de un puntero base que se calcula solo una vez por el primer thread del primer warp: ese puntero apunta a la primera posición de la imagen que se debe cargar en la iteración actual. A partir de él, cada thread determina de forma independiente cuál es el elemento exacto que le corresponde traer, de modo que todos los threads juntos cubren la región completa de vecinos que se necesita.

Supongamos que tenemos un bloque de `threadsPerBlock` threads. En la primera pasada, sabemos que necesitamos cargar `threadsPerBlock` valores a partir del píxel $(x - halfFilterWidth, y - halfFilterWidth)$ hasta cubrir toda la primera línea de vecinos que incluirá cada thread. Sin embargo, el último thread de este grupo requiere un valor adicional que, por su posición, no estaría cargado si solo se contaran los `threadsPerBlock` elementos. Por eso, cuando calculamos la cantidad a transferir, sumamos siempre `threadsPerBlock + 1` valores, de manera que ese thread "extra" obtenga la pieza que falta sin problemas. En la práctica, el primer thread se encarga de leer ese dato adicional de más, pero todos lo almacenan en su posición correspondiente dentro de `interMatrix`.

Cuando la región de vecinos abarca más de una fila de la imagen, el proceso se repite de forma análoga. Primero se calculan cuántos datos quedan por cargar en la fila actual: si, por ejemplo, faltan R vecinos para completar la ventana en esa línea, entonces cargamos esos R valores más los `threadsPerBlock` habituales, de nuevo sumando uno extra para el último thread. Una vez que hemos cubierto la parte que falta de esa fila, avanzamos a la siguiente. Allí, volvemos a cargar el segmento correspondiente, sumando otra vez `threadsPerBlock + 1` elementos, y así sucesivamente hasta que se hayan llenado todos los valores de vecinos requeridos

por cada thread en esa iteración.

En cada uno de estos pasos, los threads acceden a posiciones consecutivas de la imagen porque, al partir del puntero base, cada thread calcula su propia posición de lectura desplazándose simplemente un índice respecto al thread anterior. De esta forma, se aprovecha la coalescencia: los `threadsPerBlock` threads leen datos contiguos en memoria global, accediendo a direcciones adyacentes, lo que minimiza la latencia y maximiza el ancho de banda disponible.

Una vez que cada thread ha copiado su parcela de datos en `interMatrix`, se produce una barrera de sincronización (`__syncthreads()`) para asegurarnos de que todos los datos necesarios estén disponibles antes de reorganizarlos en `localMatrix`. A partir de ahí, cada warp extrae su bloque de 32×16 elementos de `localMatrix` para cargarlos en los fragments WMMA, sin necesidad de volver a consultar memoria global. Debido a que `interMatrix` está en `shared`, la lectura es extremadamente rápida, y como cada thread tiene su posición reservada, no hay conflictos ni contenciones.

3.5.3. Accesos coalescentes en loads y stores

Uno de los pilares para lograr un buen rendimiento en GPU es asegurarse de que los accesos a memoria global se realicen de forma coalescida. En palabras sencillas, esto significa que los threads de un mismo warp deben leer o escribir direcciones contiguas, de modo que la GPU pueda agrupar esas solicitudes en una única transacción de memoria. Si, en cambio, cada thread pide direcciones distantes entre sí, la gpu tendrá que lanzar múltiples transacciones, aumentando la latencia y reduciendo el ancho de banda efectivo.

En nuestro kernel con Tensor Cores, cada vez que cargamos datos desde la imagen original hacia `interMatrix`, procuramos que los threads lean posiciones adyacentes en memoria global. La GPU ve una secuencia continua de direcciones y puede responder con un único bloque de datos. Este patrón se mantiene incluso al tener que saltar de fila: primero se cargan los valores restantes de la fila actual (todos contiguos), añadiendo un "+1" para que el último thread del warp también tenga su elemento extra; luego pasamos a la siguiente fila y, de nuevo, los threads leen posiciones contiguas entre sí. Como resultado, los loads a `interMatrix` se agrupan en transacciones mínimas y se amortiza el coste de la latencia global.

Algo similar ocurre con la carga de la `FilterMatrix`. Dado que almacenamos los coeficientes de ocho filtros en columnas contiguas, cuando un warp entra a la función `wmma::load_matrix_sync(mask, filterMatrix, WMMA_K)`, los threads acceden a direcciones que están alineadas y consecutivas en memoria. En concreto, cada thread toma el coeficiente correspondiente a su índice de columna y fila en el tile de 16×8, de modo que, de nuevo, se generan accesos adyacentes que la GPU agrupa en bloques de 128 bytes (dependiendo de la arquitectura), minimizando el número de operaciones de lectura. Gracias a este diseño en orden `col_major`, no solo trabajamos con datos alineados, sino que también nos aseguramos de que la carga de coeficientes sea más eficiente.

A la hora de escribir el resultado, el mismo principio se aplica. Cuando volcamos el fragmento acumulador desde `resultMatrix` a `blurredImage`, cada thread toma la posición de memoria global que le corresponde para su propio píxel y canal.

La organización está pensada para que los threads de un warp escriban direcciones consecutivas: primero, cada thread escribe el valor filtrado por el primer filtro; luego, el siguiente filtro; y así sucesivamente. Así, incluso en la etapa de store, la GPU puede compactar todas esas escrituras en la menor cantidad posible de transacciones. De este modo, los stores no quedan "esparcidos" por la memoria sino alineados, aprovechando el ancho de banda al máximo.

3.6. Cálculo con filtros de gran tamaño

Cuando el tamaño del filtro excede el que pueden manejar los Tensor Cores en una única operación (por ejemplo, cuando los valores del filtro ocupan más de 16 columnas), se adopta una estrategia basada en el cálculo por partes. Recordemos que las multiplicaciones de matrices que ejecutan los Tensor Cores, en nuestro caso, están limitadas a tiles de tamaño 16×8 en la matriz de filtros. Esto significa que, para un filtro más ancho (por ejemplo, de 5×5), no podemos cargar todos sus coeficientes de golpe. La solución que se ha implementado consiste en dividir ese filtro en bloques de 16 valores (o menos, si estamos al final) y procesarlos por franjas, acumulando los resultados intermedios.

Concretamente, se parte el conjunto de valores de entrada. En cada iteración, se extrae una franja de 16 valores consecutivos de ambas matrices: por un lado, se cargan los 32×16 datos correspondientes desde la imagen, y por otro, los 16×8 coeficientes de los filtros correspondientes a esa franja. Estos valores se reformatean para ser cargados como fragmentos en los Tensor Cores mediante las instrucciones `wmma::load_matrix_sync`. Una vez preparados los fragmentos, se ejecuta la multiplicación con `wmma::mma_sync`, pero, en lugar de sobrescribir el acumulador, se acumulan los resultados parciales dentro del fragmento `result`.

Este proceso se repite en bucle hasta que se han recorrido todas las franjas necesarias para completar el cálculo completo del filtro, así como se representa en la figura 10. En cada paso, los datos se recargan desde la memoria compartida, se formatean de nuevo, y se vuelven a lanzar las instrucciones de Tensor Core, actualizando el acumulador en cada iteración. Al finalizar, el contenido del acumulador contiene la suma total de las multiplicaciones necesarias, como si se hubiese procesado el filtro completo en una sola operación.

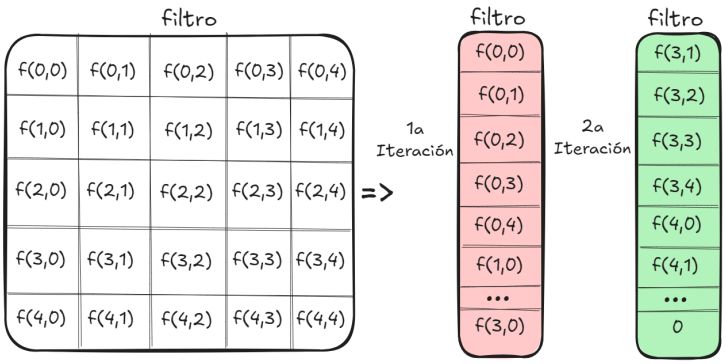

Figura 10: Iteración en filtros de gran tamaño

3.7. Optimizando el kernel

3.7.1. Reduciendo las condicionales

Para exprimir al máximo el rendimiento en GPU, ha sido esencial eliminar cualquier bifurcación (branch) innecesaria dentro de los kernels. Las divergencias de ejecución en los saltos provocan que los threads de un mismo warp tomen caminos diferentes, lo que a su vez obliga a serializar la ejecución y desperdicia ciclos de reloj., Revisamos detenidamente cada condicional presente, especialmente aquellos que se ejecutan dentro de bucles críticos o dentro del bucle principal de procesamiento. En lugar de usar sentencias `if` para comprobar si un píxel está dentro de los límites o si cierto valor es cero, preferimos reestructurar el código de manera que todos los threads del warp sigan el mismo flujo.

También se reemplazó las comprobaciones de "último valor en matriz" por operaciones aritméticas que calculan índices válidos, rellenando automáticamente con ceros los valores fuera de rango. El resultado es un kernel lo más libre posible de bifurcaciones internas en su bucle principal.

3.7.2. Optimizar los accesos a memoria

Para que los accesos a memoria, tanto global como `shared`, sean eficientes y libres de conflictos, es muy importante alinear correctamente cada bloque de datos y, en ocasiones, introducir un pequeño padding. Esto asegura que cuando un warp carga o almacena un fragmento de 32×16 valores de tipo `half`, cada thread accede a un banco distinto y la transacción se realiza en un único paso.

4. Resultados

4.1. Códigos evaluados

Para poder evaluar adecuadamente el impacto del uso de Tensor Cores en este trabajo, se han realizado distintas pruebas comparativas entre varias versiones del kernel. En particular, se han contrastado dos implementaciones diferentes que integran el cálculo con CUDA y el cálculo con Tensor Cores, pero que se diferencian en la forma en que ambos tipos de procesamiento están organizados.

La primera versión corresponde a un enfoque en el que el kernel de Tensor y el kernel tradicional de CUDA están completamente separados. En este caso, es la CPU la que, desde el host, realiza la llamada a ambos kernels de manera independiente. Es decir, primero se lanza el kernel CUDA para procesar un conjunto de filtros o imágenes, y después se lanza el kernel de Tensor para completar el procesamiento. Este enfoque implica que el control sobre la secuencia de ejecución queda en manos del host, lo que puede introducir penalizaciones en cuanto a latencia y sincronización entre lanzamientos.

En cambio, la segunda versión opta por una integración más fina: CUDA y Tensor Cores conviven dentro de un único kernel. Es decir, se lanza un solo kernel desde el host, y la lógica interna del propio kernel se encarga de decidir qué parte del procesamiento se realiza con código tradicional CUDA y cuál se ejecuta sobre Tensor Cores. De este modo, se evitan los costes derivados de realizar dos llamadas separadas desde el host, y se gana flexibilidad en cuanto a planificación y reutilización de datos, ya que toda la información reside en el kernel y no necesita ser preparada nuevamente entre llamadas.

Además, esta segunda versión permite una planificación mucho más eficiente de la carga de trabajo, ya que es posible adaptar dinámicamente la proporción de cómputo tradicional frente a cómputo en Tensor Cores en función del tipo de filtro o del tamaño del bloque, algo que no es posible cuando la decisión está fijada desde el host.

4.1.1. Método de evaluación del rendimiento

Para analizar el impacto real de las optimizaciones propuestas y comparar el rendimiento entre las distintas versiones del código, se realizaron medidas de tiempo de ejecución utilizando la librería estándar `time.h`.

La idea era sencilla: registrar una marca de tiempo justo antes de lanzar el kernel y otra justo después de la llamada a `cudaDeviceSynchronize()`. Esta sincronización garantiza que el kernel ha terminado completamente su ejecución, por lo que la diferencia entre ambas marcas nos da una estimación directa del tiempo que ha tardado el cómputo en el dispositivo.

Este enfoque tiene la ventaja de que ignora la mayor parte del overhead introducido por la CPU y se centra exclusivamente en el tiempo que la GPU tarda en procesar los datos. Obviamente, no es una medición extremadamente precisa al nivel de ciclos de reloj o uso de memoria, pero sí resulta muy útil para tener una visión general del rendimiento relativo entre distintas implementaciones.

Además, al repetir las mediciones varias veces y trabajar con promedios, pudimos minimizar la variabilidad entre ejecuciones y detectar diferencias reales cuando una

optimización aportaba mejoras consistentes.

4.1.2. Análisis con NVIDIA Nsight

Para poder afinar realmente el rendimiento del kernel y entender con detalle qué estaba ocurriendo, utilizamos la herramienta *NVIDIA Nsight Compute*. Esta herramienta nos permitió ir mucho más allá de simplemente saber cuánto tiempo tarda un kernel en ejecutarse; nos dio acceso a métricas concretas y muy valiosas.

Con Nsight pudimos, por ejemplo, medir de forma directa la utilización de los *Tensor Cores*, comprobar si los accesos a memoria estaban coalescidos o no, o ver cuánta memoria compartida y registros estaba consumiendo cada bloque. Una métrica especialmente útil fue el porcentaje de utilización efectiva del *pipeline* de cómputo, que nos ayudó a entender si el kernel estaba haciendo buen uso de los recursos del SM o si había ineficiencias.

4.1.3. Escenarios de evaluación y pruebas realizadas

Para entender realmente el comportamiento de cada una de las implementaciones y evaluar su rendimiento, realizamos un conjunto de pruebas variando distintos parámetros clave que afectan directamente al uso de recursos y al coste computacional.

Uno de los primeros aspectos que analizamos fue el reparto de trabajo entre los Tensor Cores y el código clásico en CUDA. Se probaron configuraciones donde el 100 % de la carga se realizaba en CUDA, el 100 % en Tensor Cores, y una serie de combinaciones intermedias en las que el *balancer* dividía la imagen para que cada tipo de kernel se encargase de una fracción diferente. Esto nos permitió estudiar qué proporciones ofrecían un mejor equilibrio entre rendimiento y utilización de recursos.

Además, se llevaron a cabo pruebas con distintos tamaños de filtros, desde convoluciones simples de 3×3 hasta filtros muy grandes de 243×243. Esta variación fue especialmente interesante para analizar cómo se comporta la implementación de Tensor Cores frente a la versión clásica de CUDA cuando aumenta el número de operaciones por píxel. Como era de esperar, los filtros más grandes ponen más presión en los accesos a memoria y el uso de acumuladores, lo que ayuda a destacar los puntos fuertes y débiles de cada código.

También se variaron los tamaños de las imágenes utilizadas en las pruebas, desde una resolución estándar Full HD (1920×1080) hasta una imagen de alta resolución de 6000×4000 píxeles. Esta diferencia de escala nos permitió ver cómo afectan las limitaciones de memoria y paralelismo.

Por último, se probó también con diferentes cantidades de filtros, desde una única convolución hasta 8 filtros simultáneos. En escenarios con varios filtros, se pudo comprobar cómo esta optimización permite aprovechar aún mejor el paralelismo interno de los Tensor Cores, debido a la naturaleza de la implementación realizada, aumentando la eficiencia total del cómputo.

4.2. Resultados de rendimiento

4.2.1. Same-Kernel CUDA y Tensor Fusion (Particionado a nivel de warp)

En esta estrategia, tanto el trabajo realizado por CUDA como el de Tensor Cores se ejecuta dentro de un único kernel. Esta fusión evita llamadas adicionales desde CPU y permite mantener los datos dentro del dispositivo, lo que en principio mejora la eficiencia.

Como puede observarse en la figura 2, en filtros pequeños como el de 3x3, los tiempos obtenidos apenas presentan variación al introducir Tensor Cores: se mantienen en torno a los 0.065 ms, ya que el coste de preparar los datos para este tipo de núcleos supera el beneficio obtenido al realizar tan pocas operaciones.

A medida que el tamaño del filtro aumenta, el comportamiento cambia. En el caso de 9x9, se observa un leve incremento en el tiempo de ejecución al usar Tensor Cores con un 50 % de carga (0.080 ms frente a 0.076 ms solo con CUDA). Esto se debe, probablemente, al sobrecoste de dividir y reorganizar los datos en bloques compatibles con los WMMA. En configuraciones con menor carga en Tensor Cores, como el 20 %, el tiempo mejora ligeramente (0.079 ms), lo que sugiere que un uso moderado puede ser más beneficioso para este rango de tamaños.

Con filtros de 27x27, el impacto de la estrategia es más evidente. El tiempo de ejecución con CUDA puro es de 0.159 ms, mientras que con un 50 % de uso de Tensor Cores se incrementa a 0.265 ms. No obstante, reduciendo la carga a un 20 %, el tiempo baja a 0.182 ms, más cercano al valor base de CUDA, aunque sin llegar a superarlo.

En el caso de filtros grandes como 81x81, las diferencias se amplifican. El tiempo base con CUDA es de 0.862 ms, y aunque el uso intensivo de Tensor Cores (50 %) lo eleva a 1.472 ms, con un uso más moderado (20 %) baja a 1.005 ms, indicando que el rendimiento mejora cuando los Tensor Cores se integran de forma parcial, evitando saturar los recursos o introducir excesiva fragmentación.

El filtro de 243x243 marca un punto de inflexión. Con CUDA, el tiempo se sitúa en 6.222 ms, mientras que con configuraciones como el 30 % o 20 % de carga en Tensor Cores, los tiempos ascienden a 9.297 ms y 8.268 ms, respectivamente. Aunque aún superiores al caso puramente CUDA, el speedup se mantiene constante independientemente del tamaño, anticipando que, con cargas aún mayores, el uso de Tensor Cores sigue sin aportar una mejora.

	100 %	**90 %**	**80 %**	**70 %**	**60 %**	**50 %**	**40 %**	**30 %**	**20 %**	**10 %**	**0 %**
3x3	0.061	0.065	0.063	0.064	0.064	0.065	0.069	0.064	0.074	0.073	0.065
9x9	0.087	0.083	0.081	0.080	0.080	0.082	0.087	0.089	0.079	0.091	0.076
27x27	0.281	0.276	0.279	0.275	0.268	0.265	0.214	0.217	0.182	0.191	0.159
81x81	1.917	1.858	1.792	1.627	1.582	1.472	1.228	1.184	1.005	1.048	0.862
243x243	16.634	15.590	13.880	13.250	12.670	11.455	10.349	9.297	8.268	8.665	6.222
729x729	149.89	138.39	132.24	125.40	112.23	101.87	92.49	83.38	74.03	77.40	56.51

Cuadro 2: Tiempos de ejecución para diferentes tamaños de filtro según el porcentaje de uso de Tensor Cores.

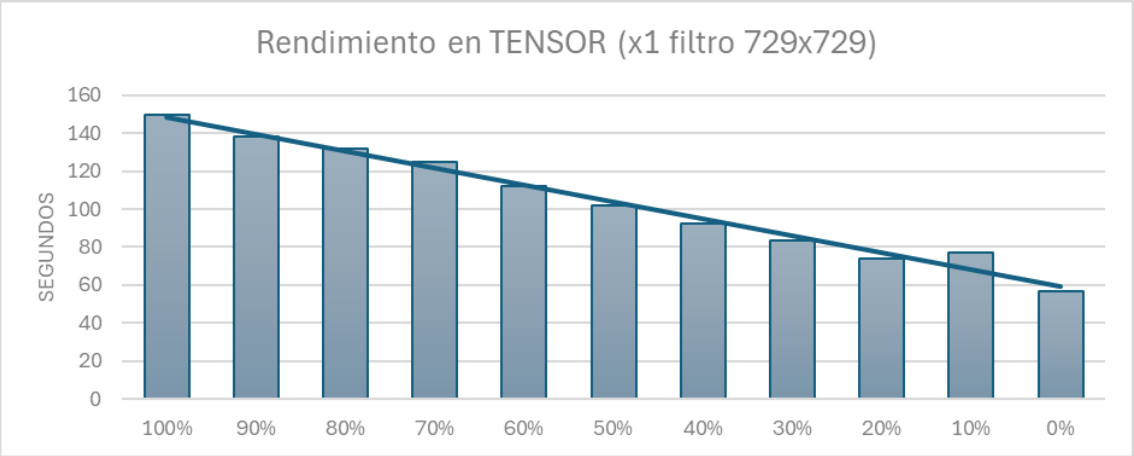

Figura 11: Rendimiento del código en relación a la carga del los Tensor Cores, 1 filtro de 729x729.

4.2.2. Particionado a nivel de kernel (Kernels separados)

En esta estrategia, se separa el trabajo en dos kernels distintos: uno para las operaciones realizadas por CUDA y otro para las llevadas a cabo por Tensor Cores. En filtros pequeños como el 3x3 o el 9x9, los resultados obtenidos son prácticamente idénticos a los del kernel unificado, lo que indica que el coste adicional de lanzar múltiples kernels y sincronizarlos es casi imperceptible a estas escalas.

Sin embargo, al trabajar con filtros medianos o grandes, esta diferencia se vuelve más relevante. En el filtro de 81x81, por ejemplo, el uso de dos kernels con un 20 % de carga en Tensor Cores da lugar a un tiempo ligeramente mayor que en la estrategia de kernel unificado (1.078 ms frente a 1.005 ms). Con el filtro 243×243, la diferencia también es apreciable: 8.900 ms frente a 8.268 ms. Esto demuestra que, aunque el enfoque de kernels separados puede simplificar la lógica del código, introduce una pequeña penalización en el rendimiento debida a la gestión adicional en CPU.

4.2.3. Mismo Kernel con múltiples filtros (8 filtros simultáneos)

Esta versión es especialmente eficiente. Se lanza un único kernel que aplica múltiples filtros a la vez (ocho en total), lo que permite reutilizar datos compartidos y minimizar el número de accesos a memoria global. Como consecuencia, el tiempo de ejecución por filtro se reduce de forma significativa.

Por ejemplo, al procesar ocho filtros 3x3 usando únicamente Tensor Cores, el tiempo total es de 0.006 ms, es decir, una media menor del 0.001 ms por filtro. Incluso con un 75 % de uso de Tensor Cores, el tiempo apenas sube a 0.009 ms. En tamaños más grandes como el 81x81, se obtiene un tiempo de 2.249 ms en total con Tensor Cores al 100 %, lo que equivale a unos 0.28 ms por filtro, una mejora sustancial si se compara con los 0.862 ms que se obtienen al aplicar un solo filtro con CUDA puro.

En el caso del filtro de 243x243, esta estrategia vuelve a demostrar su eficacia: se obtiene un tiempo total de 20.07 ms al usar solo Tensor Cores, lo que supone unos 2.5 ms por filtro, frente a los más de 6 ms por filtro cuando se usan de manera individual.

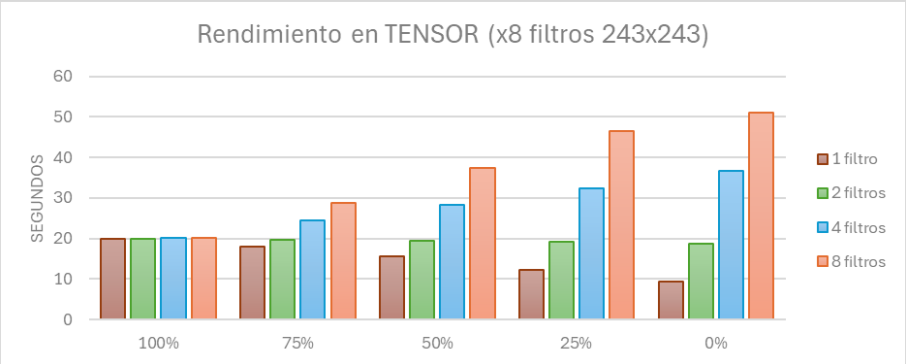

Figura 12: Rendimiento del código en relación a la carga del los Tensor Cores y número de filtros (243x243).

	100 %	**75 %**	**50 %**	**25 %**	**0 %**
3x3	0.006	0.009	0.010	0.010	0.010
9x9	0.038	0.050	0.071	0.079	0.091
27x27	0.370	0.385	0.495	0.592	0.655
81x81	2.249	3.211	4.184	5.110	5.6154
243x243	20.070	28.670	37.320	46.370	50.940

Cuadro 3: Tiempos de ejecución (ms) según el porcentaje de uso de Tensor Cores usando 8 filtros

4.2.4. Conclusiones del análisis

A la vista de los resultados, puede concluirse que el uso de Tensor Cores no siempre mejora el rendimiento, especialmente en filtros pequeños o medianos. En esos casos, el tiempo dedicado a la preparación de datos, la acumulación intermedia y la coordinación de los threads tiende a contrarrestar cualquier ventaja computacional. En cambio, a partir de cierto umbral (aproximadamente desde filtros 81x81 en adelante) la aceleración proporcionada por los Tensor Cores empieza a compensar, especialmente si se integran de forma equilibrada dentro del kernel.

Por otro lado, procesar múltiples filtros en paralelo mediante un único kernel muestra mejoras claras en todos los casos, ya que permite aprovechar al máximo la arquitectura del dispositivo y distribuir mejor la carga entre CUDA y Tensor Cores. Este enfoque, además, reduce los costes de lanzamiento de kernels y mejora la eficiencia en el uso de memoria.

No existe una configuración única que funcione para todos los casos, pero sí patrones claros que permiten decidir, en función del problema concreto, cómo estructurar el cálculo para maximizar el rendimiento.

4.3. Resultados de Nsight

Las siguientes mediciones se han realizado con una distribución del 50 % del trabajo entre Tensor Cores y CUDA Cores, utilizando 8 filtros, cada uno de tamaño 81×81, sobre una imagen de 6000×4000 píxeles en formato ARGB. Esta configuración se ha elegido dado que cada Tensor Core opera sobre 32 valores (haciendo uso de 1 warp), una división 50/50 asegura una asignación a priori simétrica del trabajo.

Además, 8 filtros representan el máximo que puede ser gestionado simultáneamente en la matriz interna de los Tensor Cores, y un tamaño de 81×81 es lo suficientemente grande como para poner en evidencia las diferencias de rendimiento entre enfoques sin que los tiempos se disparen en exceso.

4.3.1. Particionado a nivel de kernel (Kernels separados)

En la versión que separa los kernels de Tensor Cores y CUDA Cores, el análisis del rendimiento de cada uno muestra diferencias interesantes. El kernel dedicado a Tensor Cores alcanzó un *Compute Throughput* del 72,24 %, lo que indica que, aunque los Streaming Multiprocessors (SMs) de la GPU pasan buena parte del tiempo ejecutando instrucciones útiles, todavía queda un 28 % de ciclos en los que no hay warps listos o se están esperando datos. Asimismo, su *Memory Throughput* quedó en un 55,91 %, reflejando que la mitad de la actividad de memoria se está usando con cierta intensidad, pero sin saturar el ancho de banda disponible. Con un IPC efectivo de alrededor de 2,89 inst/cycle y un SM Busy del 72,98 %, queda claro que el kernel de Tensor Cores no está completamente limitado ni por la capacidad de cálculo ni por el acceso a memoria, sino en una zona intermedia en la que ambos factores pueden mejorarse.

Throughput

En contraste, el kernel exclusivo de CUDA Cores obtuvo un *Compute Throughput* del 89,31 %, muy por encima del resultado del tensorial. Con una duración de 4,63 s frente a la ejecución de Tensor Cores, y un *Memory Throughput* que coincide con el 89,31 %, esta versión demuestra que, al tratarse únicamente de instrucciones de cómputo y carga–almacenamiento convencionales, los SMs quedan más saturados, utilizando durante más tiempo los pipelines de ALU (con un 64,9 % de ocupación en ciclos activos) y de LSU (hasta un 90,8 % en función de cuántas instrucciones de carga/almacenamiento se emiten). Su *IPC* activo de 2,97 inst/cycle y un *SM Busy* del 74,30 % confirman que el kernel CUDA explota mejor la capacidad teórica de cálculo de la GPU.

Esa superioridad en el throughput de cómputo y memoria se traduce también en una mejor eficiencia de las caches del CUDA puro: el *L1/TEX Hit Rate* roza el 98,91 %, mientras que en la versión Tensor apenas llegaba a un 69,02 %. Además, el *L1/TEX Cache Throughput* sube hasta un 90,84 % en el kernel de CUDA, comparado con el 56,49 % de la versión de Tensor, lo cual evidencia que buena parte de las referencias de memoria se están resolviendo de forma inmediata en L1. El *L2 Hit Rate* sigue siendo alto (cerca del 79,68 %), pero ahí ya se nota cierta presión extra, dado que el L2 Cache Throughput de la versión CUDA es del 2,68 %, ligeramente superior al 1,39 % del kernel de Tensor, lo que indica que, aunque algunas referencias fallan en L1, L2 las resuelve la mayoría de las veces.

Ciclos

En términos de ciclos totales, la ejecución Tensor necesitó unos 4.060.236.312 ciclos (con 4.018.655.809 ciclos activos de SM), mientras que el kernel CUDA requirió 7.855.776.529 ciclos (con 7.723.712.508 ciclos activos). La proporción de ciclos activos

en CUDA es más alta (aproximadamente un 98 % de los ciclos, frente a casi un 99 % en L1 y L2 combinadas), lo que representa un uso más intenso y continuo de la GPU. En cambio, el kernel de Tensor Core, con un DRAM Throughput apenas del 0,48 %, muestra que la gran mayoría de accesos se resuelve en cache, pero al mismo tiempo sufre más latencia en la reorganización de datos para WMMA.

Memoria

Otro punto distintivo es la latencia de acceso a DRAM: el kernel de CUDA, aunque también minimiza las referencias a la memoria externa, presenta un DRAM Throughput del 1,51 %, tres veces el valor de la versión Tensor. Esto sugiere que, en la ejecución puramente CUDA, algunas referencias puntuales escapan a L2 y llegan a DRAM, pero el trade-off se equilibra con una mayor capacidad de cómputo. Por el contrario, el kernel Tensor depende en mayor grado de L1 y L2, pero su bajo throughput global de cache (especialmente en L1) provoca que, a veces, varias operaciones de fragmentación terminen pidiendo datos fuera de esos niveles.

En cuanto a ocupación, ambos kernels están limitados al 50 % por la cantidad de memoria compartida dinámica (aprox. 40,96 KB por bloque) y los 40–48 registros por thread. Sin embargo, el kernel CUDA logra mantener constantes 1.358,71 waves por SM con un Achieved Occupancy del 49,68 %, lo que demuestra que la latencia queda bien cubierta por warps adicionales en cola. El kernel de Tensor, por su parte, alcanza 2.717,40 waves por SM, prácticamente el doble, porque cada warp de Tensor Core genera más warps en cola para cubrir la fragmentación de los bloques de 32x8x16 y la acumulación parcial. Aun así, ninguno de los dos kernels supera el 50 % de ocupación, lo que indica que la configuración de shared memory impide llegar a una mayor concurrencia.

Saltos

En la parte de divergencias, el kernel CUDA no presenta prácticamente ramas divergentes: su Branch Efficiency es del 100 %, con 0 divergencias promedio, mientras que el kernel de Tensor registra un 98,77 % de eficiencia y unas 569.293,48 ramificaciones divergentes en promedio. Reducir esas bifurcaciones en el workflow tensorial ayudaría a elevar su eficiencia.

Coalescencia

En el diagnóstico de accesos no coalescidos, ambos kernels muestran porcentajes elevados: el kernel CUDA registra 30 % de sectores L2 excesivos con respecto a los producidos en caso de un coalescing óptimo (equivalente a 67.767.840.000 sectores no coalescidos), mientras que el tensor registra un 31 %. Esto implica que buena parte de lecturas globales se fragmentan en varias transacciones, desaprovechando ancho de banda. Para mitigar esto, sería necesario ajustar la organización de la `interMatrix` y de la `FilterMatrix` para que los 32 threads de cada warp accedan a 32 `half` consecutivos, garantizando transacciones coalescidas. Del mismo modo, los conflictos en memoria compartida persisten: el kernel Tensor sufre un 82 % de wavefronts excesivos, y el kernel CUDA aún presenta un L2 Compression Success Rate del 0 %, lo que indica que los patrones de datos no son adecuados para compresión.

4.3.2. Comparación: Kernels fusionados vs. Kernels particionados

En la versión que combina CUDA Cores y Tensor Cores en un único kernel, el *profiling* con NVIDIA Nsight muestra un Compute Throughput del 73,72 %, cifra que mejora ligeramente respecto al 72,24 % logrado por el kernel exclusivamente tensorial en la implementación separada, pero que queda por debajo del 89,31 % que alcanza el kernel puramente CUDA. Esto indica que, al intentar aunar ambos tipos de unidades dentro del mismo código, los SMs pasan más tiempo ejecutando instrucciones que cuando solo estaban dedicados a Tensor Cores, pero no llegan a saturarse tanto como en el caso de usar únicamente CUDA Cores. En general, la GPU permanece "ocupada" un 70,83 % del tiempo, cifra similar al 72,98 % (kernel tensor) y el 74,30 % (kernel CUDA), lo que refleja que el flujo de trabajo resulta más equilibrado al compartir recursos, pero sin dejar de existir ciclos de espera atribuibles tanto a la reorganización de datos en memoria compartida como a la posible congestión en los distintos pipelines.

Throughput

El *Memory Throughput* se sitúa en un 73,72 %, un salto notable frente al 55,91 % del kernel de Tensor Cores, pero ligeramente inferior al 89,31 % de la versión CUDA pura. Esto quiere decir que la fusión de ambos tipos de operaciones permite un uso mucho más intenso de la cache L1 y la L2, reduciendo significativamente las transferencias directas a DRAM (que en este caso suben solo al 1,55 %, frente al 0,48 % del kernel tensor) y, al mismo tiempo, mantiene una presión de memoria menor que la del kernel CUDA puro, donde la DRAM ocupaba el 1,51 % del ancho de banda.

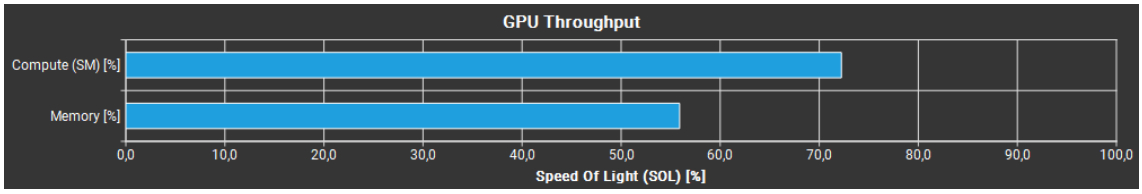

Figura 13: Throughput de la GPU en el kernel Tensor

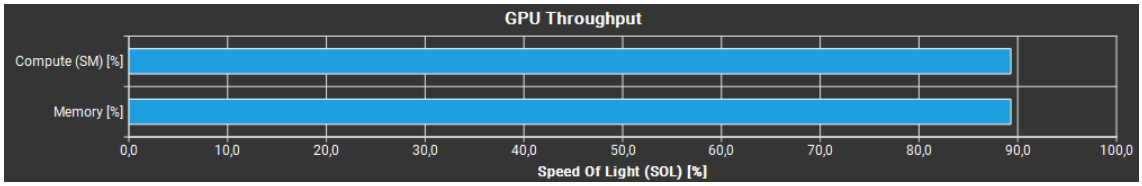

Figura 14: Throughput de la GPU en el kernel Cuda

Ciclos

En cuanto a la emisión de instrucciones, el IPC activo es de 2,83 inst/cycle, cercano al 2,97 inst/cycle del kernel CUDA, pero superior al 2,42 inst/cycle que

registraba el kernel de Tensor Cores. Ese nivel de IPC revela que la GPU gestiona casi tres instrucciones por ciclo cuando los SMs están activos, aunque aún no alcanza la cifra teórica máxima de cuatro [22]. El pipeline ALU se utiliza en un 61,0 % de los ciclos activos, mientras que el LSU (unidad de carga/almacenamiento) cae a un 75,0 % de ocupación, un reflejo de que, al unir ambos flujos de trabajo, las operaciones de carga y almacenamiento resultan más frecuentes y compiten en mayor medida con las operaciones aritmético-lógicas. En contraposición, el kernel puramente CUDA dedicaba un 64,9 % al ALU y un 90,8 % al LSU, lo que demostraba que, en ese escenario, la mayor parte del tiempo se emplea en mover datos de manera más intensiva.

El análisis de ciclos totales muestra 10.911.907.816 ciclos transcurridos, de los cuales 10.719.751.048 ciclos corresponden a SMs activos. La proporción entre ciclos activos y totales (casi el 98 %) sigue indicando que la GPU se emplea eficientemente, con muy pocos ciclos desperdiciados en espera absoluta.

Memoria

Esa mejora en cache se ve claramente en el *L1/TEX Hit Rate*: ahora alcanza un 98,66 %, en comparación con el 69,02 % del kernel tensorial y casi igual al 98,91 % del kernel CUDA. Gracias a esto, el *L1/TEX Cache Throughput* sube al 75,04 %, frente al 56,49 % que se obtenía en la versión separada de Tensor Cores. En consecuencia, gran parte de los accesos a datos se resuelven directamente en L1, evitando costosas solicitudes a L2 o DRAM. Sin embargo, esa combinación de ambos tipos de operaciones introduce más lecturas desde L2, por lo que el *L2 Cache Throughput* sube ligeramente al 2,61 %, comparado con el 1,39 % del kernel tensorial y el 2,68 % de CUDA. El *L2 Hit Rate* se sitúa en 79,24 %, un valor razonable que refleja un buen equilibrio entre fallos en L1 y éxito en L2, aunque todavía existe margen para mejorar.

Ventajas y desventajas

Uno de los puntos fuertes de este kernel unificado es la mejora en el aprovechamiento de cache: la reducción de accesos a DRAM, que baja al 1,55 %, se traduce en menor latencia global. Sin embargo, siguen teniendo problemas de acceso a memoria no coalescido: se contabilizan 76.597.570.000 sectores excesivos en la capa L2, equivalente al 31 % del total de sectores teóricos, frente al 45 % del kernel tensorial y el 30 % del kernel CUDA. Esto significa que, a pesar de la reorganización de datos en un solo kernel, siguen existiendo patrones de lectura que no garantizan que cada warp solicite bloques de 32 datos contiguos en memoria global.

Los conflictos en memoria compartida también persisten: el 82 % de los *wavefronts* presenta accesos excesivos en bancos compartidos, una cifra incluso ligeramente superior al 74 % de la versión tensorial y a valores similares en CUDA. Esto se explica porque, al pasar datos de forma intermedia entre la sección de CUDA y la de Tensor, se generan patrones que vuelven a cruzar threads de un mismo warp en los mismos sectores de memoria.

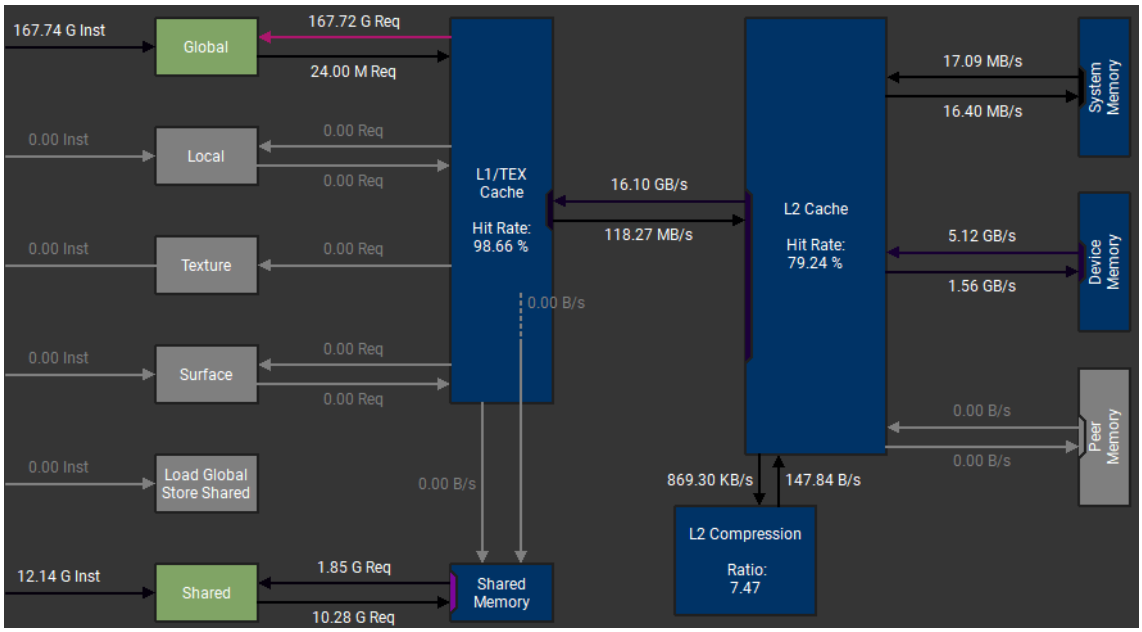

Figura 15: Representación del throughput medido entre distintas transacciones de memoria.

Saltos

En la parte de las bifurcaciones, el kernel combinado reduce su tasa de ramas divergentes: registra un *Branch Efficiency* del 99,82 % con un promedio de 279.211,96 bifurcaciones divergentes, la mitad de las 569.293,48 de la versión tensorial. Esto se debe a que la lógica de decisión para dividir trabajo entre CUDA y Tensor Cores se ha optimizado, eliminando algunas comprobaciones excesivas. Sin embargo, sigue existiendo un 0,02 % de las instrucciones totales que son bifurcaciones.

Finalmente, en términos de ocupación, el kernel mixto mantiene un 49,67 %, casi idéntico al 49,99 % del kernel tensor y al 49,68 % del kernel CUDA. En todos los casos, la limitación proviene de la memoria compartida (alrededor de 37,12 KB por bloque) y de los 40 registros por thread. Para superar esa barrera, sería necesario reducir el tamaño de las matrices intermedias o reorganizar el uso de registros, de modo que cada SM pueda alojar más warps y cubrir con mayor holgura las latencias de memoria.

4.3.3. Implementación de la matriz intermedia

La incorporación de la matriz intermedia (`interMatrix`) al kernel que fusiona CUDA Cores y Tensor Cores ha tenido un impacto mixto: por un lado, se observa cierta mejora en el comportamiento de memoria de largo plazo, pero por otro, el rendimiento global se ve penalizado por el coste adicional de cargar y reorganizar datos. A continuación se describen los cambios más relevantes en comparación con la versión anterior (sin `interMatrix`) y se explica por qué se producen esas mejoras o empeoramientos.

Throughput

En primer lugar, el Compute Throughput baja del 73,72 % al 70,67 %. Esto significa que los SMs pasan ahora un porcentaje ligeramente menor de ciclos ejecutando instrucciones efectivas. La razón principal es que añadir `interMatrix` implica más trabajo previo de recopilar datos en memoria compartida antes de lanzarlos a Tensor Cores o CUDA Cores, lo que introduce saltos de control y sincronizaciones adicionales. En la versión anterior, parte del tiempo del SM estaba dedicado directamente a cómputo (ya fuera en ALU o en los pipelines de Tensor), mientras que en esta variante buena parte de los warps ejecutan el bucle de carga a `interMatrix`. Dicho bucle consume ciclos de ALU y de acceso a `shared` para rellenar esa estructura, y por tanto se reduce el porcentaje de ciclos dedicados a las operaciones matriciales o al procesamiento puro de CUDA.

Memoria

El Memory Throughput también desciende, del 73,72 % al 66,38 %. Con la matriz intermedia, cada warp antes de realizar su cálculo carga de forma colaborativa un fragmento de la imagen o del filtro en `interMatrix`, con lo que disminuyen (en teoría) las lecturas redundantes de memoria global. A pesar de esta mejora, ese beneficio queda contrarrestado por la propia lectura desde `shared`, que en este caso implica sincronizaciones (`__syncthreads()`) y posibles conflictos de banco. En la versión anterior, los accesos a memoria global se hacían directamente en un solo paso (aunque con mayor redundancia), mientras que ahora se segmentan en dos fases: global $\rightarrow$ intermedio y luego intermedio $\rightarrow$ registros/fregmentos. Esa doble etapa añade latencia interna y reduce la tasa bruta de transferencias globales visibles, de modo que el throughput efectivo cae.

Cache

Si se examina el nivel de cache, el L1/TEX Hit Rate permanece muy alto en ambos casos (98,62 % con `interMatrix` vs. 98,66 % antes). Sin embargo, la métrica de L1/TEX Cache Throughput cae drásticamente: del 75,04 % al 67,46 %. Esto se debe a que, ahora, buena parte de las referencias que antes llegaban directamente a L1 tras acceder a memoria global pasan primero por `shared`. Desde la perspectiva de Nsight, esas referencias a `interMatrix` no contabilizan como aciertos de L1/TEX Cache, sino como accesos a memoria compartida. Por tanto, aunque la tasa de acierto siga muy alta, su "throughput" en esa capa parece reducido, pues muchos datos se sirven desde `shared` y no desde L1/TEX.

En segunda capa, el L2 Hit Rate mejora (85,73 % frente a 79,24 %), y el L2 Cache Throughput baja ligeramente (2,34 % vs. 2,61 %). Esto indica que menos accesos llegan a L2, porque la reorganización en `interMatrix` permite que varios warps reutilicen datos sin tocar L2. En la versión sin intermedia, cada warp podría volver a solicitar datos antiguos a L2, mientras que ahora esos datos ya residen en `shared`. El resultado es una reducción en la presión sobre L2 (menor throughput), pero un aumento en la proporción de aciertos cuando se necesita, de modo que la "eficacia" de L2 en aquellos pocos accesos que sí se producen sube a 85,73 %.

A pesar de ello, el DRAM Throughput apenas varía (1,57 % vs. 1,55 %), lo que muestra que la matriz intermedia no logra eliminar por completo las referencias a DRAM. En el caso anterior, las lecturas desde global llegaban en mayor número, provocando un 1,55 % de throughput DRAM. Al agregar `interMatrix`, algunas de esas lecturas se reducen, pero otras salen por DRAM al no caber la totalidad de datos en L1/L2/shared, manteniendo una proporción similar de tráfico externo.

Ciclos

El número de ciclos totales aumenta de 10.911.907.816 a 12.172.517.689. Dado que la cantidad de trabajo computacional (en términos de operaciones de filtro) se mantiene, la diferencia de $1,2610^9$ ciclos extras corresponde al overhead de cargar y almacenar datos en `interMatrix`. La frecuencia de SMs se mantiene casi igual (1,68 1,70 GHz), por lo que esa subida de ciclos no se justifica por cambios en reloj, sino puramente por la mayor carga de trabajo de preparación.

Desde el punto de vista de emisión de instrucciones, el IPC activo sube ligeramente de 2,78 a 2,83 inst/cycle, y el SM Busy pasa del 70,83 % al 71,82 %. Es decir, cuando el SM está activo, emite casi la misma cantidad de instrucciones, y el porcentaje de tiempo con al menos un warp listo mejora ligeramente. Esto se explica porque el flujo de trabajo con `interMatrix` genera más operaciones de carga/almacenamiento y menos de Tensor puro, equilibrando un poco mejor las instruccciones de memoria y aritméticas. El pipeline LSU desciende su ocupación del 75 % al 67,5 %, mientras que el ALU permanece en torno al 61 %. Esto implica que la parte de cargas en memoria compartida consume menos ciclos de LSU que los accesos directos a memoria global del kernel anterior.

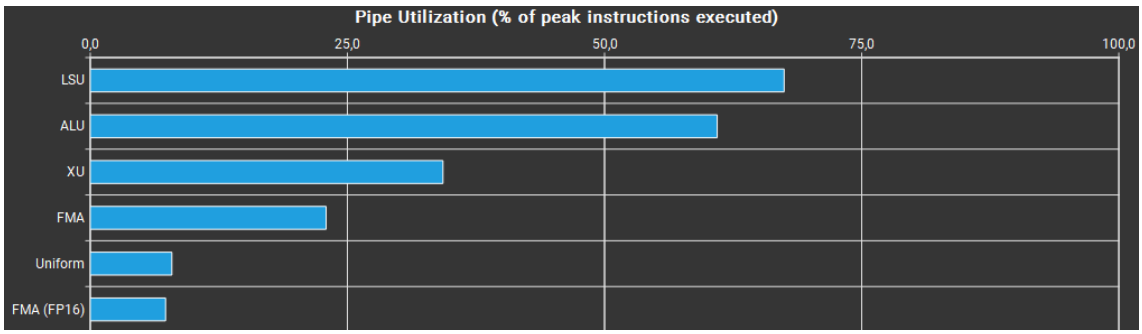

Figura 16: Utilización del pipeline con respecto el porcentaje máximo de instrucciones ejecutadas

En cuanto a la ocupación, sigue limitada al 50 % (49,67 % 49,83 %), ya que la cantidad de memoria compartida dinámica (37,12 KB) y 48 registros por thread no cambia. La diferencia es que, al bajar el uso de LSU y L1/TEX, la GPU puede cubrir mejor las latencias de memoria compartida, por lo que se experimenta un leve aumento de los waves per SM (2 717,40 vs. 2 717,40); en la práctica se mantienen iguales, lo que indica que la ocupación real del chip no mejora.

Coalescencia

El análisis de accesos no coalescidos muestra que la proporción de sectores L2 excesivos baja al 30 % (68.731.530.000 sectores), comparado con el 31 % anterior. Esto sugiere que el paso intermedio por `interMatrix` ayuda a reorganizar un poco mejor los datos, de modo que algunos warps acceden a bloques más contiguos en memoria global. Sin embargo, ese 30 % sigue siendo elevado: aún hace falta lograr que los 32 threads de cada warp lean 32 valores consecutivos de `half` para aproximarse a la coalescencia total. En memoria compartida, los bank conflicts descienden al 74 % de wavefronts excesivos, bajando desde el 82 % anterior. Esto confirma que `interMatrix` mejora ligeramente la distribución de accesos hacia bancos distintos, pero sigue habiendo una contención notable. Para erradicarla por completo, habría que ajustar la forma en que los threads calculan sus índices en `shared`.

Saltos

Finalmente, el Branch Efficiency se mantiene alta (99,65 % vs. 99,82 %), aunque el número de instrucciones de bifurcación crece (31.883.314.036 vs. 30.410.913.505). Esto es consecuencia de incluir más lógica condicional para gestionar las cargas en `interMatrix`, que agrega pequeñas comprobaciones dentro de bucles.

4.4. Impacto de múltiples filtros

Al analizar el comportamiento al aumentar de 1 a 8 filtros 81×81, se observa cómo los Tensor Cores logran mantener los tiempos prácticamente estables, mientras que la versión puramente en CUDA sufre un crecimiento enorme al añadir más filtros.

4.4.1. Resultados de tiempo

Empezando por las mediciones con time.h, con un solo filtro 81×81 la ejecución tarda 2,218 segundos. Al subir a 4 filtros, ese tiempo queda en 2,226 segundos, apenas 8 milésimas más, y con 8 filtros sólo sube a 2,249 segundos. En cambio, en CUDA los tiempos pasan de 1,028 segundos (un filtro) a 4,207 segundos (cuatro filtros) y se disparan hasta 8,295 segundos (ocho filtros). Esa diferencia tan brutal se debe a que los Tensor Cores procesan los ocho filtros simultáneamente en cada warp, mientras que CUDA Cores debe ejecutar cada filtro de forma secuencial o en lanzamientos sucesivos, acumulando el coste.

4.4.2. Resultados de Nsight

Las métricas de Nsight ayudan a entender por qué esta estabilidad se sostiene. Con un único filtro, el Compute (SM) Throughput ronda el 71,60 % y el SM Busy el 72,78 %. Cuando pasamos a 2 filtros, baja ligeramente a 69,60 % de Compute y 70,70 % de SM Busy; a 4 filtros sube un poco a 70,35 % y 71,47 %; y con 8 filtros cae a 67,81 % de Compute y 71,07 % de SM Busy. Estos números evidencian que, a medida que crece el número de filtros, algunos ciclos de SM empiezan a quedarse esperando datos, pero el descenso es moderado precisamente porque los Tensor Cores siguen alimentando con trabajo paralelo a los SMs.

En el apartado de memoria, el patrón es similar. Con 1 filtro, el Memory Throughput está en 76,74 %, la L1/TEX Hit Rate en 98,49 % y la L2 Hit Rate en 88,99 %. Al sumar filtros, estas cifras se degradan muy despacio: a 2 filtros, 74,68 % (Memory), 98,21 % (L1) y 69,51 % (L2); a 4 filtros, 75,47 % (Memory), 97,66 % (L1) y 73,70 % (L2). Sólo al llegar a 8 filtros baja de verdad la L2 al 42,51 % y el Memory Throughput al 72,95 %, lo que obliga a recurrir más a DRAM (sube de 1,47 % a 2,51 % del ancho de banda). Aun así, aunque en ese punto la cache empieza a saturarse, esa penalización en memoria solo alarga el tiempo total en unas décimas de segundo porque el grueso del cómputo (las multiplicaciones matriciales) sigue ejecutándose simultáneo en los Tensor Cores.

El IPC (instrucciones por ciclo) se mantiene cerca de 2,8 inst/cycle en todos los casos, y el LSU (pipeline de cargas/envíos) crece ligeramente (pasa de un 67,5 % a un 75 %) conforme se añaden más filtros, mostrando que los accesos a memoria aumentan. Sin embargo, la ALU sigue rondando el 61 % de ocupación, indicando que sigue habiendo un buen balance entre cómputo puro y transferencias de datos.

4.5. Impacto del tamaño del filtro

Al comparar diferentes tamaños de filtro (ejecutando 8 en los Tensor Cores), se aprecia que los tiempos apenas varían gracias al paralelismo masivo de estos núcleos.

4.5.1. filtro: 3x3

Por ejemplo, con un filtro 3×3 el proceso tarda entre 6 y 8 ms, ya sea 1, 4 u 8 filtros, porque los Tensor Cores hacen todos los cálculos simultáneamente. En cambio, en CUDA puro, pasan de 3 ms (un filtro) a 13 ms (ocho filtros).

4.5.2. filtro: 9x9

Con el filtro 9×9, medimos 44–46 ms para uno, cuatro u ocho filtros en Tensor Cores; en CUDA, esos números suben de 16 ms a 127 ms. Nsight confirma que, aunque el SM Throughput baja al 62,8 % y el SM Busy al 64,1 %, la cache L1 aún resuelve el 91,6 % de lecturas y la L2 el 78 %, manteniendo los accesos a DRAM relativamente bajos (5,4 %).

4.5.3. filtro: 27x27

Al pasar a 27×27, el tiempo en Tensor Cores está en torno a 265 ms (uno) y 370 ms (cuatro u ocho). Aquí la GPU utiliza más cache (L1 al 95,6 % y L2 al 79,6 %), y el Compute Throughput sube al 69,5 %, con el SM Busy al 70,4 %. La presión en DRAM baja al 1,9 % porque la mayoría de datos siguen en L1/L2.

4.5.4. filtro: 81x81

Con el filtro 81×81, los tiempos van de 2,218s (1 filtro) a 2,249s (8 filtros). Nsight muestra que el Compute Throughput es 67,8 % y el SM Busy 71,1 %. La L1 todavía acierta el 96,6 % de los accesos, pero la L2 cae al 42,5 % y DRAM sube al 2,5 %. En ese punto, la cache no da para tanto, pero aun así añadir siete filtros extra apenas añade 31 ms.

5. Conclusiones

5.1. Visión general

Tras todas las pruebas realizadas, se ha demostrado que es posible obtener un rendimiento significativamente superior al aprovechar por completo los Tensor Cores mediante la aplicación de diversos filtros y estrategias de optimización. Esta aproximación, centrada en explotar al máximo las capacidades de cómputo especializado de WMMA, ha superado a otras alternativas híbridas. En particular, la idea de fusionar CUDA Cores y Tensor Cores dentro de un mismo kernel no logró superar los tiempos obtenidos al dedicar la carga de trabajo al 100 % a uno u otro tipo de unidad por separado. Aunque a priori mantener los datos en el dispositivo y evitar lanzamientos adicionales de kernel podía parecer más eficiente, en la práctica, la sobrecarga derivada de reorganizar los datos para WMMA y sincronizar ambas fases terminó por anular cualquier posible ventaja.

5.1.1. De la convolución a la multiplicación matricial

Convertir la operación de convolución en un producto de matrices fue uno de los retos principales. Cada columna de la matriz de entrada representa el vecindario de un píxel y cada fila contiene los coeficientes del filtro. Sin embargo, esa transformación conlleva duplicar muchos valores en memoria (ya que los vecindarios solapan) y obliga a sincronizar cada iteración para cargar fragmentos de datos en memoria compartida antes de lanzarlos a los Tensor Cores. En consecuencia, los filtros pequeños pierden cualquier ganancia de rendimiento, y solo cuando el tamaño crece lo suficiente (por ejemplo, 81×81) las operaciones matriciales amortizan la sobrecarga.

5.1.2. Limitaciones de memoria y cache

La memoria compartida demostró ser un recurso muy escaso: cada bloque acababa consumiendo casi 40 KB para almacenar las matrices intermedias (`localMatrix`, `filterMatrix` y `resultMatrix`). Dado que cada SM solo dispone de unos 96 KB para uso compartido, la ocupación real no pudo superar el 50 %. Esto tuvo dos consecuencias: por un lado, el número de warps por SM se quedó en 24, lo que limitó nuestra capacidad de ocultar latencias; por otro, los conflictos en los bancos de memoria compartida fueron muy frecuentes (entre el 74 % y el 82 % de wavefronts excesivos), lo que provocaba stalls constantes. A nivel de jerarquía de cache, la L1 presentó tasas de acierto muy elevadas con filtros grandes (más del 95 %), pero la L2 comenzó a fallar a medida que el filtro crecía; por ejemplo, con 81×81 y ocho filtros la L2 Hit Rate cayó hasta el 42,5 %, disparando el acceso a DRAM (alrededor del 2,5 % del ancho de banda) y alargando el tiempo total.

5.1.3. Rendimiento según tamaño y número de filtros

Las mediciones de tiempo con `time.h` y los datos de Nsight muestran que los Tensor Cores mantienen los tiempos casi constantes al multiplicar el número de filtros. En un filtro 3×3, el tiempo se mantiene en 6–8 ms tanto para 1 como para 8 filtros, mientras que en CUDA puro el salto iba de 3 ms a 13 ms. Para un filtro

9×9, Tensor Cores tarda 44–46 ms independientemente de que sean 1, 4 u 8 filtros, pero en CUDA va de 16 ms a 127 ms. Con un 27×27, Tensor Cores pasa de 265 ms a 370 ms al procesar ocho filtros; en CUDA, de 119 ms a 980 ms. Cuando el filtro alcanza 81×81, Tensor Cores oscila entre 2,218 s (un filtro) y 2,249 s (ocho filtros), mientras que CUDA salta de 1,028 s a 8,295 s. Y con 243×243, Tensor Cores sube de 19,87 s a 20,07 s, pero en CUDA pasa de 9,365 s a 73,23 s. En todas esas pruebas, el paralelismo masivo de los Tensor Cores se traduce en prácticamente la misma duración total al aumentar los filtros, mientras que CUDA puro multiplica el costo de forma lineal.

5.1.4. Lecciones y vías de mejora

En definitiva, el proyecto ha dejado en claro que fusiones demasiado agresivas entre CUDA y Tensor Cores no compensan los costos de reorganización de datos. Para extraer más rendimiento, habría que reducir los conflictos en memoria compartida ajustando el padding o el stride de las matrices intermedias, optimizar la coalescencia de accesos globales (asegurando que los 32 threads de cada warp lean 32 half contiguos) y minimizar al máximo las ramas y sincronizaciones dentro de loops. También convendría revisar el uso de registros y memoria compartida para elevar la ocupación por encima del 50 % y explorar sublanzamientos con streams para solapar aún más cómputo y transferencias. Aun con estas limitaciones físicas, ya hemos sentado unas bases sólidas y obtenido una comprensión detallada de dónde están los cuellos de botella, lo que permitirá extraer rendimiento adicional en futuras iteraciones.

Referencias

[1] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *ACM Queue*, vol. 6, pp. 40–53, Mar. 2008.

[2] J. Choquette, "Volta: Programmability and performance," in *Hot Chips 29*, 2017. NVIDIA Technical Presentation, Tesla V100 with 640 Tensor Cores.

[3] NVIDIA Corporation, "Gv100 gpu architecture whitepaper," tech. rep., NVIDIA, 2017.

[4] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance & precision," *arXiv preprint arXiv:1803.04014*, 2018.

[5] M. Fasi, N. J. Higham, M. Mikaitis, and S. Pranesh, "Numerical behavior of nvidia tensor cores," *PeerJ Comput. Sci.*, vol. 7, p. e330, 2021.

[6] NVIDIA Corporation, *CUDA C++ Programming Guide*, 2025. Versión 12.4.

[7] MolSSI, "Cuda programming model." https://education.molssi.org/gpu_programming_beginner/03-cuda-program-model.html, 2024.

[8] Damavis, "Cuda tutorial – blocks and grids." https://blog.damavis.com/en/cuda-tutorial-blocks-and-grids/, 2024.

[9] N. Corporation, "Using cuda warp-level primitives," *NVIDIA Technical Blog*, 2018.

[10] C. University, "Simt and warps," *Cornell Virtual Workshop*, 2024.

[11] M. Giles, "Lecture 3: Control flow and synchronisation." https://people.maths.ox.ac.uk/gilesm/cuda/lecs/lec3.pdf, 2025. Universidad de Oxford.

[12] W.-m. W. H. Wang, "Fundamental optimizations in cuda." https://developer.download.nvidia.com/GTC/PDF/1083_Wang.pdf, 2009. Presentación en NVIDIA GTC.

[13] E. Dev, "Cuda gpu organization hierarchy." https://eunomia.dev/others/cuda-tutorial/04-gpu-architecture/, 2025. Tutorial sobre arquitectura CUDA.

[14] N. Corporation, "Turing tuning guide." https://docs.nvidia.com/cuda/turing-tuning-guide/index.html, 2025. Documentación oficial de NVIDIA.

[15] N. Corporation, "Using shared memory in cuda c/c++." https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/, 2024. Blog técnico de NVIDIA.

[16] U. de Stack Overflow, "In cuda, what is memory coalescing, and how is it achieved?." https://stackoverflow.com/questions/5041328/in-cuda-what-is-memory-coalescing-and-how-is-it-achieved, 2011. Discusión en Stack Overflow.

[17] NVIDIA, “Programming tensor cores in cuda 9,” 2017.

[18] NVIDIA, “Nvidia ampere architecture in-depth,” 2020.

[19] wzsh, “Matrix multiply-accumulate with tensor cores by wmma api,” 2025.

[20] NVIDIA, “Profiling guide — nsight compute 12.9 documentation,” 2023.

[21] NVIDIA Developer Forums, “Tensor core metrics not showing up in nsight?,” 2023.

[22] U. de NVIDIA Developer Forums y Greg (NVIDIA), “Max ipc of 3080.” NVIDIA Developer Forums, 2020. Discusión sobre IPC teórico en GPUs con capacidad de cómputo CC 7.x–8.x (Volta, Turing, Ampere).

Anexos

A. Ejemplo de multiplicación matricial con Tensor Cores

```
__global__ void matrixMulTensorCore(half* d_A, half* d_B,
    float* d_C, int width) {
    // Cargar dimensiones
    int lda = width;
    int ldb = width;
    int ldc = width;
    // Repartir en grids 2d
    int warpM = (blockIdx.x * blockDim.x + threadIdx.x) / 32;
    int warpN = (blockIdx.y * blockDim.y + threadIdx.y) / 8;
    // Declarar los fragmentos
    nvcuda::wmma::fragment<nvcuda::wmma::matrix_a, 16, 16, 16,
        half, nvcuda::wmma::row_major> a_frag;
    nvcuda::wmma::fragment<nvcuda::wmma::matrix_b, 16, 16, 16,
        half, nvcuda::wmma::row_major> b_frag;
    nvcuda::wmma::fragment<nvcuda::wmma::accumulator, 16, 16,
        16, float> c_frag;
    nvcuda::wmma::fill_fragment(c_frag, 0.0f);

    // Iterar por los fragmentos
    for (int i = 0; i < width; i += 16) {
        int aRow = warpM * 16;
        int aCol = i;
        int bRow = i;
        int bCol = warpN * 16;
        if (aRow < width && aCol < width &&
            bRow < width && bCol < width) {
            // Load the inputs
            nvcuda::wmma::load_matrix_sync(a_frag, d_A + aRow *
                lda + aCol, lda);
            nvcuda::wmma::load_matrix_sync(b_frag, d_B + bRow *
                ldb + bCol, ldb);
            // Perform the matrix multiplication
            nvcuda::wmma::mma_sync(c_frag, a_frag, b_frag,
                c_frag);
        }
    }
    // cargar en el valor de d_C, escalarlo por beta, y agregar
        nuestro resultado escalado por alfa
    int cRow = warpM * 16;
    int cCol = warpN * 16;
    // Guardar el output
    nvcuda::wmma::store_matrix_sync(d_C + cRow * ldc + cCol,
        c_frag, ldc, nvcuda::wmma::mem_row_major);
}
```

B. Kernel CUDA para el filtro gaussiano

```
__global__ void Cuda(uint8_t* const blurredImage, const uint8_t*
    const rawImage, int width, int height, int channels, const
    half* filter, int filterWidth, int numFilters, int
    initialBlock) {
        // Identificadores de threads
        int pos = (blockIdx.x + initialBlock) * blockDim.x +
            threadIdx.x;
        int indexWarp = (threadIdx.x % (warpSize));
        int warpId = (threadIdx.x / warpSize);
        int temp = (pos / warpSize) * WMMAM + indexWarp;
        // pixel y canal a tratar
        int x = (temp / channels) % width;
        int y = (temp / channels) / width;
        int canal = temp % channels;
        // Comprobar thread util
        if (x >= width || y >= height || canal >= channels){
        return;}
        // mitad ancho del filtro
        int halfFilterWidth = filterWidth / 2;
        // numero de valores de filtrado
        int filterSize = filterWidth * filterWidth;

        for (int i = 0; i < numFilters; i++) {
        // pixel desenfocado
        half blurredPixel = 0;
        // Calcular el pixel desenfocado
        for (int filterY = -halfFilterWidth; filterY <=
            halfFilterWidth; filterY++) {
        for (int filterX = -halfFilterWidth; filterX
            <= halfFilterWidth; filterX++) {
            //comprobacion de limites
            int imageX = min(max(x + filterX, 0), width - 1);
            int imageY = min(max(y + filterY, 0), height - 1);
            // Calcular el indice del filtro
            int filterIndex = (filterY + halfFilterWidth) *
                filterWidth + (filterX + halfFilterWidth);
            // Pixel de la imagen a tratar
            half pixel = (half) rawImage[((imageY * width +
                imageX) * channels) + canal];
            blurredPixel += pixel * filter[filterIndex + i *
                filterSize];
        }
        }
            blurredImage[((y * width + x) * channels) + canal
                + (i * width * height * channels)] = (uint8_t)
                    __half2float(blurredPixel);
        }
}
```

C. Carga de datos en la GPU y llamada al kernel

```
// Definir punteros para la memoria de la GPU
uint8_t *d_originalImage, *d_blurredImage;
half *d_filter;

// Reservar memoria en la GPU para las imagenes
cudaMalloc((void**)&d_originalImage, width * height *
    channels * sizeof(uint8_t));
cudaMalloc((void**)&d_blurredImage, width * height *
    channels * numFilters * sizeof(uint8_t));
cudaMalloc((void**)&d_filter, filterWidth * filterWidth
    * numFilters * sizeof(half));
// Copiar desde la memoria del host a la memoria de la GPU
cudaMemcpy(d_originalImage, originalImage, width * height
    * channels * sizeof(uint8_t), cudaMemcpyHostToDevice);
cudaMemcpy(d_filter, filters, filterWidth * filterWidth
* numFilters * sizeof(half), cudaMemcpyHostToDevice);

//procedimiento
int threadsPerBlock = NUM_WARPS * 32;
dim3 blockDim(threadsPerBlock);
int gridDim((width * height) / ((threadsPerBlock) / channels)
    + 1);
// initial balancer value
float balancer = 0.5f;
int blocksTensor = balancer * gridDim;
dim3 gridDimTensor(blocksTensor);
dim3 gridDimCuda(gridDim - blocksTensor);
// espacio de memoria compartida
size_t sharedMemorySize = (WMMA_M * WMMA_K) * NUM_WARPS
    * sizeof(half) +
    (WMMA_N * WMMA_K) * sizeof(half) +
    (WMMA_M * WMMA_N) * NUM_WARPS * sizeof(float) +
    (threadsPerBlock * 5) * sizeof(half);

//GaussianBlur<<<gridDim, blockDim, sharedMemorySize>>>
    (d_blurredImage, d_originalImage, width, height,
    channels, d_filter, filterWidth, numFilters, balance);
Tensor<<<gridDimTensor, blockDim, sharedMemorySize>>>
    (d_blurredImage, d_originalImage, width, height,
    channels, d_filter, filterWidth, numFilters);
Cuda<<<gridDimCuda, blockDim, sharedMemorySize>>>
    (d_blurredImage, d_originalImage, width, height,
    channels, d_filter, filterWidth, numFilters, blocksTensor);

cudaDeviceSynchronize();
// Copiar la imagen final desde la GPU al host
cudaMemcpy(blurredImage, d_blurredImage, width * height *
    channels * numFilters * sizeof(uint8_t), cudaMemcpyDeviceToHost);
```