Configuración Automática del TMA para GPUs de Alto Rendimiento

Nicolás Meseguer[1], Yifan Sun[2], Michael Pellauer[3], José L. Abellán[1], Manuel E. Acacio[1]

Resumen— **Para lograr el máximo rendimiento en GPUs, resulta esencial optimizar la localidad de los datos y hacer uso de la ejecución asíncrona, con el fin de reducir al mínimo los costes de acceso a la memoria y solapar cómputo con transferencias de memoria. Si bien características hardware como el *Tensor Memory Accelerator* (TMA) y la especialización a nivel de warp contribuyen a abordar estos desafíos, su complejidad de uso limita a menudo a los programadores.**

En este trabajo presentamos ACTA (*Automatic Configuration of the Tensor Memory Accelerator* o, en castellano, configuración automática del acelerador TMA), una biblioteca software que simplifica y optimiza el uso del TMA. Al aprovechar la tabla de especificaciones de la GPU (*GPU Specification Table*, GST), ACTA determina dinámicamente los tamaños de *tile* y las configuraciones de cola óptimas para cada *kernel* y arquitectura. Su algoritmo garantiza un solapamiento eficiente entre memoria y cómputo, reduciendo drásticamente la complejidad de programación y eliminando la necesidad de una exploración exhaustiva del espacio de diseño.

Nuestra evaluación sobre un conjunto diverso de *kernels* muestra que ACTA logra un rendimiento apenas un 2,78 % por debajo del ideal y requiere tan solo una única pasada de configuración. Esto convierte a ACTA en una solución práctica y eficiente para optimizar las cargas de trabajo de las GPUs modernas, ya que combina un rendimiento casi óptimo con un esfuerzo de programación significativamente menor.

Palabras clave— **GPU, Ejecución asíncrona, Especialización a nivel de warp, Optimización de kernels, Biblioteca Software, TMA**

I. Introducción y Motivación

En la actualidad, las GPUs se han convertido en componentes fundamentales de los centros de procesamiento de datos, y se han consolidado como la plataforma de cálculo dominante para acelerar un sinfín de aplicaciones, desde el aprendizaje profundo y la computación de alto rendimiento hasta el análisis masivo de datos, entre otros [1].

A pesar de su enorme rendimiento computacional y notable ancho de banda de memoria, las GPUs a menudo plantean importantes desafíos a los programadores para aprovechar al máximo su potencial. Estas dificultades se hacen especialmente evidentes en cargas de trabajo sensibles a la latencia de la memoria, que contribuyen significativamente a notables diferencias de rendimiento [2], [3], [4], [5].

Estas ineficiencias a menudo se deben a la incapacidad de los programas para solapar eficazmente las operaciones de memoria con el cómputo, lo que da lugar a recursos de la GPU ociosos e infrautilizados [6].

[1]Dpto. de Ingeniería y Tecnología de Computadores, Universidad de Murcia, e-mail: {n.meseguerib orra,jlabellan,meacacio}@um.es.

[2]Department of Computer Science, William & Mary University, e-mail: ysun25@wm.edu.

[3]NVIDIA, e-mail: mpellauer@nvidia.com.

A medida que las GPUs siguen evolucionando con la incorporación de hardware especializado (por ejemplo, los Tensor Cores [7]) y nuevos tipos de datos (por ejemplo, TF32 o FP4), destinados a mejorar el rendimiento de las aplicaciones para necesidades específicas actuales [8], las transferencias asíncronas de memoria se están convirtiendo en una característica estándar en las GPU de gama alta. En concreto, la arquitectura Hopper (H100) de NVIDIA [9] ha introducido recientemente una unidad hardware llamada *Tensor Memory Accelerator* (TMA). Esta unidad permite transferir tiles de forma asíncrona entre memoria global y memoria compartida en paralelo al cómputo. Además, técnicas como la especialización de warps permiten que distintos warps se encarguen de tareas específicas dentro de un *kernel*, lo que facilita un mejor solapamiento entre los accesos a la memoria y el cómputo [10], [11], [12], [13].

Desafortunadamente, las transferencias de memoria asíncronas suponen un reto importante para los programadores, que no sólo deben implementar las transformaciones de datos necesarias, sino también programar cuidadosamente los accesos a la memoria para garantizar que estén disponibles cuando se necesiten. Esto constituye un problema importante. Si echamos la vista atrás, el procesador Cell BE [14] permitía hacer transferencias de DMA asíncronas entre la memoria externa e interna del chip en sus *Synergistic Processing Elements* (SPEs), consiguiendo así un aumento significativo del rendimiento. Sin embargo, esta labor tan tediosa a la que se enfrentaban los programadores fue una de las causas principales por las que se canceló su producción, tan solo unos pocos años después del lanzamiento del Cell BE.

Aunque existan mecanismos especializados como el TMA y técnicas software como la especialización de warps, estas también añaden una capa de complejidad que hace que programar en la GPU sea cada vez sustancialmente más difícil y complicado de mantener.

Para aprovechar las nuevas unidades de TMA y la especialización a nivel de warp se necesitan conocimientos avanzados de GPU y una comprensión profunda de herramientas como el SDK de NVIDIA o librerías como CUTLASS, que proporcionan soporte de programación a alto nivel. Sin embargo, aunque estas herramientas ofrecen implementaciones de ejemplo que ayudan en la programación, adaptarlas a kernels en un entorno real suele ser complejo y requiere amplios conocimientos de hardware.

Para reducir la brecha entre la productividad y el rendimiento de las aplicaciones en plataformas de alto rendimiento con soporte para el TMA, presenta-

mos ACTA (Automatic Configuration of the Tensor Memory Accelerator). ACTA es una biblioteca software que, mediante un enfoque heurístico, establece el tamaño de los tiles y la disposición de la memoria compartida en cada aplicación, disminuyendo significativamente la carga de trabajo del programador y aprovechando el TMA para mejorar el rendimiento.

Para ello, ACTA requiere conocer las características de la GPU, como la frecuencia de reloj, la capacidad de cómputo (por ejemplo, el número de *Streaming Multiprocessors*, SMs), los detalles de la jerarquía de caché (tamaño de línea, capacidad total, latencias), el espacio total disponible de la memoria compartida y el ancho de banda máximo de memoria, entre otros. Asimismo, considera las particularidades algorítmicas de cada kernel, como la intensidad aritmética y el tipo de vector (*streaming* o *stationary*) [15], para ajustar de manera óptima su configuración.

Con esta información y basándose en la ley de Little [16], ACTA infiere una configuración lo más cercana posible a la óptima para aquellos parámetros que determinan el rendimiento del TMA (véase la sección III para más detalles). Como resultado, ACTA consigue un equilibrio entre complejidad manejable y alto rendimiento, ofreciendo una solución práctica y eficiente para extraer el máximo rendimiento de las GPUs de gama alta modernas que integran unidades TMA.

Nuestras contribuciones son:

- ACTA introduce un enfoque sistemático para automatizar la configuración de colas.
- ACTA consigue abstraer los intrincados detalles del TMA, lo que permite a los desarrolladores conseguir implementaciones de alto rendimiento, en un menor tiempo y con un menor esfuerzo.

II. Background

La latencia y el ancho de banda de memoria son elementos críticos en las GPUs contemporáneas, ya que afectan directamente al rendimiento y la eficiencia. La disparidad entre las unidades de cómputo y los sistemas de memoria—relativamente más lentos—crea cuellos de botella, especialmente cuando se manejan aplicaciones que procesan grandes volúmenes de datos. La memoria compartida de la GPU (la *shared memory* en NVIDIA) desempeña un papel clave en la mitigación de estos problemas, al actuar como almacenamiento intermedio de alta velocidad. Sin embargo, la utilización eficaz de la memoria compartida depende, en gran medida, de los patrones de acceso. Una gestión ineficiente de la memoria, que incluya accesos no coordinados o una dependencia excesiva de la memoria global, puede provocar una grave degradación en el rendimiento.

Para aprovechar los recursos de una GPU, la especialización de warps [10] asigna diferentes tareas a warps individuales dentro de un bloque de hilos (*threadblock* en NVIDIA). Este enfoque difiere de la expectativa típica de que todos los hilos de una GPU ejecuten las mismas instrucciones simultáneamente. Al permitir que un warp se centre en tareas específicas, como la gestión de las transferencias de memoria, mientras otros se encargan del cómputo, la especialización introduce heterogeneidad entre ellos. Sin embargo, este esquema requiere una coordinación muy cuidadosa para evitar ineficiencias, sobre todo en la sincronización entre estas transferencias y cómputo.

Partiendo de este concepto, el Tensor Memory Accelerator (TMA), una nueva unidad hardware especializada introducida en la arquitectura Hopper (H100) de NVIDIA, amplía las capacidades de la especialización de warps para implementar un esquema productor-consumidor. Así, un pequeño número de hilos (productores) orquestando el TMA son capaces de gestionar un gran volúmen de transferencias asíncronas de datos entre memoria global y memoria compartida, mientras que el resto de hilos (consumidores) se centran en el cómputo. Esto beneficia notablemente a aplicaciones modernas que manejan grandes estructuras de datos regulares como los tensores (arrays N dimensionales) del aprendizaje profundo. Además, el TMA introduce mecanismos de sincronización acelerados por hardware, que permiten a los hilos consumidores esperar eficientemente los datos sin causar esperas en el *pipeline* de ejecución.

Las operaciones de TMA se inician mediante un descriptor, una estructura compacta que especifica la dirección de la memoria global, el tensor y el número de elementos a copiar. Una vez iniciada la operación por uno o varios hilos productores, el TMA gestiona de forma autónoma el cálculo y la generación de direcciones de memoria, los posibles cálculos de *strides* y las condiciones de sincronización, lo que reduce significativamente la carga del programador. En cada petición, los bloques de datos pueden transferirse sin problemas entre la memoria global (GMEM) y la memoria compartida (SMEM), optimizando la utilización del ancho de banda y minimizando la latencia.

Una innovación clave en el TMA es su modelo de sincronización, que introduce barreras asíncronas especializadas para optimizar la coordinación entre hilos productores y consumidores.

En particular, el TMA emplea barreras de transacción asíncronas, que dividen la sincronización en dos fases: llegada y espera. Los hilos productores señalan su progreso ejecutando un comando de llegada cuando los datos están listos (un ACK). Esta operación no se bloquea, lo que permite a los productores pasar a realizar tareas independientes sin bloquearse (es decir, cargar un nuevo bloque de datos). Los hilos consumidores, por su parte, emiten una orden de espera sólo cuando necesitan los datos, bloqueándose hasta que todos los productores han señalado su llegada. Este proceso en dos pasos permite a los hilos iniciales utilizar los ciclos ociosos para otros cálculos.

Al aprovechar estas barreras asíncronas aceleradas por hardware y la sincronización basada en transacciones, el TMA tiene el potencial de solapar las transferencias de memoria con el cómputo.

Por desgracia, para sacar el máximo partido, el programador debe involucrarse en el manejo del

TMA. Una mala gestión de las dependencias, como un orden incorrecto de las operaciones de memoria, puede provocar condiciones de carrera, bloqueos o resultados incorrectos, lo que complica el proceso de depuración. Además, la configuración de las operaciones de TMA requiere un conocimiento detallado de la disposición de los datos subyacentes y de la carga de trabajo, lo que exige precisión a la hora de definir parámetros como las dimensiones del tensor y los *strides* de memoria.

Para reducir la complejidad de programación del TMA, CUDA introduce la API *cuda::pipeline* con múltiples etapas para gestionar las operaciones de memoria asíncronas. En nuestro trabajo, implementamos una solución personalizada denominada *OperandQueues*, a la que nos referiremos como colas de aquí en adelante, inspirada en [17], que proporciona una abstracción adaptada a nuestro enfoque.

Al igual que las operaciones de TMA se inician mediante un descriptor de copia, las colas se inicializan con un descriptor de cola. Este descriptor define parámetros clave para las transferencias de memoria y el uso del TMA, como la dirección de inicio de la memoria global, el tamaño de los vectores, las dimensiones de los tiles, el número de elementos de cada cola, y el destino en la memoria compartida. Una vez inicializadas, las colas gestionan automáticamente los descriptores del TMA subyacentes, abstrayendo más aún los detalles del movimiento de datos.

En este esquema, el warp productor utiliza la cola para transferir datos y se sincroniza con los warps consumidores mediante funciones de cola especializadas. Los consumidores recuperan y procesan los datos de la cola, sincronizando sus operaciones perfectamente con las transferencias de memoria. Tras consumir un tile, los consumidores envían una señal a la cola, lo que permite al productor cargar el siguiente tile automáticamente. Aunque la sincronización sigue siendo necesaria, las colas simplifican enormemente el proceso al proporcionar funciones intuitivas de alto nivel que abstraen las tareas de coordinación de bajo nivel. Sin embargo, el problema que aún persiste es la definición de las configuraciones de las colas, que en última instancia dependen tanto de las características de la propia GPU (por ejemplo, tamaño de la memoria compartida, FLOPS pico, ancho de banda de la memoria, ...) como de las particularidades de cada *kernel* (por ejemplo, intensidad aritmética, número de colas, ...).

III. ACTA

En esta sección, presentamos ACTA, una biblioteca software diseñada para inferir y proporcionar configuraciones óptimas de tamaños de tile y número de elementos de la cola para el uso del TMA en GPUs de gama alta. De este modo, en lugar de automatizar completamente el proceso de configuración, ACTA ofrece a los desarrolladores los parámetros críticos necesarios para inicializar y utilizar eficazmente las colas del TMA, agilizando el proceso de configuración del *kernel* y mejorando el rendimiento.

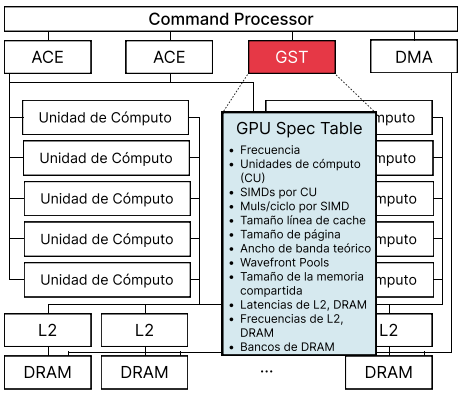

Fig. 1: Visión a alto nivel de la GPU, con la GST, la nueva estructura hardware requerida por ACTA, resaltada en rojo.

ACTA se nutre de varios parámetros arquitectónicos de la GPU importantes para determinar las configuraciones óptimas de las colas y las operaciones de memoria. Suponemos que los valores de estos parámetros para una GPU concreta son accesibles a través de una estructura hardware que cada fabricante debe proporcionar, que denominamos tabla de especificaciones de GPU (*GPU Specification Table*, GST)[1]. La figura 1 muestra la organización del modelo de GPU asumido en este trabajo con la GST añadida. El hecho de que ACTA dependa de los parámetros de configuración de la GPU garantiza su aplicabilidad a una amplia gama de arquitecturas, lo que convierte a ACTA en una solución robusta y adaptable.

A. ACTA y su flujo de trabajo

Para entender el funcionamiento de ACTA es necesario considerar tanto la implementación por parte del host como su interacción con el hardware de la GPU. El listado 1 ilustra un ejemplo típico de código del lado del host que emplea ACTA[2].

En el lado del host, ACTA amplía el SDK de la GPU proporcionando una API para configurar dinámicamente las colas del *kernel*. Tras la típica fase de configuración—copiar los datos del *host* al *device* e inicializar las *command queues* (colas de comandos) (líneas 1-3)—se llama a la función *InitACTA* para preparar la ejecución del kernel. *InitACTA* toma varios parámetros, como la intensidad aritmética del kernel (por ejemplo, *MEDIUM*), el número de wavefronts consumidores (por ejemplo, 8) y las unidades de cómputo (CUs) de la GPU para, de esta manera, poder guiar la optimización y la ocupación del planificador (*scheduler*).

Durante la fase de inicialización, ACTA interactúa con el hardware de la GPU para consultar su tabla de especificaciones (GST), a la que se accede a través del procesador de comandos como se muestra en la figura 1. La GST contiene detalles críticos de la ar-

[1]En caso de no estar disponible, ACTA podría hacer un proceso de *microbenchmarking* para inferir estos parámetros dinámicamente.

[2]En lo sucesivo, aunque ACTA es agnóstico al fabricante de GPU, adoptaremos la terminología de AMD para mayor claridad y para ajustarnos a nuestra configuración experimental basada en AMD.

```
1  driver.MemCopyH2D(b.device_A, b.MatrixA)
2  driver.MemCopyH2D(b.device_B, b.MatrixB)
3  driver.CreateCommandQueue()
4
5  // Inicializa ACTA
6  driver.InitACTA(MEDIUM, 8, 64)
7
8  // Registra las colas
9  driver.RegisterQueue(K, 4, STREAMING)
10 driver.RegisterQueue(K, 4, STATIONARY)
11
12 // Obten las colas configuradas en orden
       FIFO
13 a_queue = driver.SizeQueue()
14 b_queue = driver.SizeQueue()
15
16 // Cargar el kernel usando ACTA
17 kernArg := KernelArgs{
18     b.d_A, b.d_B, b.d_Z, M, K, N,
19     K0,a_queue.TileSize,b_queue.TileSize,
20     K2, M0, M1, M2,
21     a_queue.QueueTiles,b_queue.QueueTiles
22 }
23
24 driver.EnqueueLaunchKernel(kernArg)
```

Listado 1: Ejemplo de código host de alto nivel para una multiplicación matriz-matriz. Las funciones de ACTA están resaltadas en azul.

quitectura, como el tamaño de la memoria compartida, *Local Data Share* (LDS) (*scratchpad* en terminología de NVIDIA) para las colas, la capacidad de cómputo, el ancho de banda de memoria y las latencias, entre otros (la figura 1 enumera los parámetros considerados por ACTA). Con esta información, ACTA valida estos parámetros frente a los requisitos del kernel y registra las colas mediante la función *RegisterQueue* (líneas 9 y 10). Este proceso incluye la especificación del tipo de cola (por ejemplo, *streaming* o *stationary*), el tamaño del vector (por ejemplo, la dimensión K en una matriz) y el tamaño del tipo de datos (por ejemplo, un float de 4 bytes).

A continuación, ACTA dimensiona dinámicamente las colas en orden FIFO (a través de la función *SizeQueue* de las líneas 13 y 14), aprovechando la información almacenada en el GST para determinar las configuraciones óptimas, incluidos el tamaño de los tiles y el número de elementos, *slots* de ahora en adelante, asignados a cada cola. Los valores extraídos de la GST guían el proceso de optimización del kernel, garantizando que este aproveche eficazmente las características arquitectónicas de la GPU. Esta estrecha integración entre ACTA y la GST permite configurar los recursos de la GPU de forma eficiente y adaptable a cargas de trabajo específicas.

Finalmente, el host prepara los argumentos del kernel (líneas 18-21) utilizando las optimizaciones proporcionadas por ACTA. La estructura *KernelArgs* incorpora parámetros como el tamaño de tile y la configuración de las colas. Por último, el *kernel* se lanza mediante la función *EnqueueLaunchKernel*.

B. El algoritmo detrás de ACTA

El algoritmo de ACTA es el mecanismo central responsable de inferir y seleccionar el tamaño óptimo de tiles y slots de las colas utilizando la estructura

Algoritmo 1: Cálculo óptimo de tile

Rango de tamaños de tile: `[min, max]`, Wavefronts Consumidores, Ar.I., GST
Output: Tamaño Óptimo de Tile
Function *optimal_tile_size()*
  for *tile* ∈ *[min, max]* **do**
    meritFactor ← `evalua(`*procesamiento vs transferencia para un tile*`)`;
    costFunction ← `estima(`*uso de memoria*`)`;
    weightedMerit ← `combina(`*meritFactor, costFunction para computar el valor final*`)`;
    if *tile es mejor que best* **then**
      actualiza *best*;
  best ← `ajusta(`*basado en un factor de escalado y Ar.I.*`)`;

Algoritmo 2: Función para el cálculo del factor de mérito

Tamaño de Tile, GST **Output:** Factor de mérito
Function *evaluate()*
  `// Paso 1: Computa el mejor tiempo de scheduler para procesar el tile`
  $bestScheduling \leftarrow \frac{\text{TileSize}}{\text{SIMDMuls/Cycle} \times \text{mín}(\text{ConsumerWfs}, 4)}$
  `// Paso 2: Calcula el tiempo de procesamiento, incluyendo el overhead del scheduler`
  $procTime \leftarrow bestScheduling + (bestScheduling - 1) \times \text{mín}(\text{ConsumerWfs} - 1, \text{WfPools})$
  `// Paso 3: Calcula el tiempo de transferencia`
  $latencyTotal \leftarrow \text{TMACycles} + \text{DRAMLatency} + \text{L2Latency}$
  $memTransferTime \leftarrow \frac{\text{TileSize} \times \text{ElementSize}}{\text{Bandwidth}}$
  $cacheTransferTime \leftarrow 2 \times \frac{\text{TileSize} \times \text{ElementSize}}{\text{CacheLineSize}}$
  `// Paso 4: Agrega el tiempo de transferencia`
  $memTime \leftarrow latencyTotal + memTransferTime + cacheTransferTime$
  `// Paso 5: Devuelve el factor de mérito como el ratio cómputo vs transferencia`
  return $\frac{\text{procTime}}{\text{memTime}}$

propia *TMA Operand Queues* descrita anteriormente en la Sección II. Este proceso comienza registrando todas las colas mediante la función *RegisterQueue*. Registrar todas las colas de antemano es un paso crítico, ya que proporciona al algoritmo una visión completa del *kernel*: el número de colas, sus tamaños y su uso previsto.

Una vez registradas todas las colas, se llama a la función *SizeQueue* para calcular la configuración óptima para una cola específica. Si la configuración para la cola solicitada y el *kernel* ya se ha calculado, el algoritmo simplemente devuelve los valores precalculados. Estos valores incluyen el tamaño de los tiles, el número de slots, el tamaño en bytes de los elementos y el número total de tiles del vector. Este mecanismo de almacenamiento en cache elimina la necesidad de cálculos redundantes, agilizando el rendimiento. Si aún no se ha calculado la configuración para dicha cola/*kernel*, el algoritmo procede a determinar los valores óptimos mediante un enfoque sistemático.

El primer paso de ACTA consiste en determinar el tamaño óptimo de tiles (Algoritmo 1). Esta función evalúa iterativamente los tamaños de tile en un

Algoritmo 3: Cálculo del número óptimo de slots para las colas

```
Colas streamings y stationary, Ar.I., Unidades de
        Cómputo Output: Número óptimo de slots
        para cada cola
Function optimal_num_slots()
    cuenta número de colas streaming y stationary;
    if hay colas streaming then
        numSlots ← useLittlesLaw();
        numSlots ← roundToPowerOfTwo(numSlots);
        numSlots ← roundBasedOnCUs(numSlots);
        if suficente espacio en la memoria
          compartida then
            allocateSpace(colas streaming);
        else
            numSlots ← useArithmeticIntensity();
            reduce numSlots si falta espacio;
            allocateSpace(colas streaming);
    if hay colas stationary then
        . calcula el espacio disponible para cada cola;
        determina cuantos slots caben en el espacio
          restante;
        numSlots ← roundToPowerOfTwo(numSlots);
        numSlots ← roundBasedOnCUs(numSlots);
        reduce numSlots si falta espacio;
        allocateSpace(colas stationary);
```

rango predefinido, empezando por un mínimo, por ejemplo, 64 elementos—un valor fijo que representa el tamaño mínimo de línea de cache que puede transferirse de memoria—y extendiéndose hasta un máximo, por ejemplo, 8 192 elementos—un límite determinado a través de nuestra exploración del espacio de diseño. Para cada tile, se calcula un factor de mérito mediante la función *evaluate()* (Algoritmo 2), que representa la relación entre el tiempo de procesamiento y el tiempo de transferencia de memoria. El tiempo de procesamiento se determina por los ciclos de cómputo: operaciones aritméticas del kernel y la utilización de los wavefronts. El tiempo de memoria incorpora parámetros arquitectónicos clave como la latencia de la DRAM, los tiempos de transferencia de la cache y el ancho de banda. Estos cálculos se basan en información específica de la GPU, que es obtenida de la GST, lo que garantiza que el algoritmo se adapte a las características del hardware.

Además del factor de mérito, el algoritmo calcula una función de coste para evaluar el uso de recursos al transferir un tile, teniendo en cuenta la latencia, el ancho de banda y las restricciones de cache. Posteriormente, el factor de mérito y la función de coste se combinan obteniendo así una puntuación ponderada, que determina la idoneidad de cada tile. De este modo, se garantiza que el tile seleccionado ofrezca el equilibrio óptimo entre eficiencia de cómputo y transferencia.

Después de iterar sobre los posibles tamaños de tile, el algoritmo se ajusta a la intensidad aritmética (*arithmetic intensity*, Ar.I.) del kernel: incrementa el tamaño de tile en kernels con baja Ar.I. (por ejemplo, *Elementwise* o *Dot-Product*) para mejorar el rendimiento de memoria y lo reduce en kernels con alta Ar.I. (por ejemplo, *Matrix-Matrix*) con el fin de equilibrar el solapamiento entre memoria y cómputo. Esto garantiza que el tamaño del tile se ajuste a las características de cada kernel.

El siguiente paso de ACTA consiste en determinar el número óptimo de slots para cada cola, labor que gestiona la función *optimal_num_slots()* (Algoritmo 3). Este proceso empieza contando la cantidad de colas streaming y estacionarias (stationary), dado que la estrategia de asignación prioriza las colas streaming para maximizar el rendimiento, reservando los recursos restantes para las colas stationary.

Para las colas streaming, el número óptimo de slots se determina empleando la ley de Little [16], que describe la relación entre la velocidad a la que los ítems entran en un sistema, el tiempo que tardan en procesarse y la cantidad media de ítems. La observación de Little contribuye a que los recursos no queden infrautilizados ni saturados, manteniendo así un flujo eficiente y constante. Esta ley ha sido ampliamente aplicada en gestión de colas y arquitectura de computadores [16]. En el contexto de ACTA, la ley de Little se adapta para calcular el número ideal de slots requerido en una cola streaming, teniendo en cuenta la rapidez con la que los tiles se cargan en la memoria compartida mediante operaciones del TMA, así como el tiempo total necesario para computar y procesar un tile.

Después de calcular el número de slots, el valor se redondea (hacia arriba o abajo) a la potencia de dos más próxima y se ajusta en función del número de unidades de cómputo (CUs). A continuación, el paso final garantiza que la cantidad de slots calculada encaje en la memoria compartida disponible. Si los slots no se ajustan, se redimensionan en base a la Ar.I. de la carga de trabajo (*kernel*). Para cargas de trabajo con Ar.I. baja, se asignan más slots para mejorar el rendimiento de memoria y conseguir un mayor cómputo; en cargas de trabajo con Ar.I. alta, se eligen menos slots para reducir la presión sobre la memoria y mejorar el solapamiento entre cómputo y transferencias. Una vez validado, se reservan los slots para las colas streaming.

Tras finalizar la configuración de las colas streaming, se repiten los pasos anteriores para las colas stationary (estacionarias), después de dividir de manera equitativa la memoria compartida restante entre ellas.

En llamadas posteriores a *SizeQueue*, los tamaños de tile y las configuraciones de slots precomputados se reutilizan, evitando cálculos extra y aumentando la eficiencia durante la ejecución del *kernel*. La función *InitACTA* puede invocarse múltiples veces antes de cada *kernel*, lo que brinda la flexibilidad de programar una serie de *kernels* con configuraciones personalizadas (mejorando así el rendimiento y la utilización de los recursos). Cada llamada a *InitACTA* restablece los valores en cache, obligando a ACTA a recalcular las configuraciones óptimas de las colas. No obstante, este proceso requiere invocar las cuatro funciones clave (registrar colas, determinar el tamaño de tile, asignar slots y adaptarse a los requisitos específicos del kernel).

Tabla I: Kernels y espacio de diseño ahorrado usando ACTA.

Kernel	Descripción	Dimensiones	# Colas	# Tiles	# Slots	# Combinaciones
ElementwiseK	Operaciones optimizadas para un alto rendimiento	16777216	1	5	5	25
Elementwise Sumvectors	Operaciones elemento a elemento de uso general	16777216	2	5	5	625
Dot-Product	Bloque fundamental en muchas aplicaciones Carga de trabajo crítica para diversas tareas de álgebra lineal	2097152				
Matrix-Vector	Un elemento esencial en la computación científica y el procesamiento de datos	$[2048, 2048] \times 2048$	8+1	8	5	2,6e+14
Matrix-Matrix	Fundamental para el álgebra lineal densa y las tareas de aprendizaje automático	$[512, 2048] \times [2048, 128]$				

Tabla II: Especificaciones de la GPU R9 Nano.

Parámetro	Propiedad	Unidades
Frecuencia	1.0 GHz	-
CUs	-	64
SIMDs	64 Muls/ciclo	64
L1 Cache Vectorial	16KB 4-vias	64
L1 Cache Inst.	32KB 4-vias	16
L1 Cache Escalar	16KB 4-vias	16
L2 Cache	256KB 16-vias	16
DRAM	512MB	16

IV. Metodología de evaluación

A. Herramienta de simulación

Para cuantificar los beneficios de ACTA, utilizamos MGPUSim [18], un simulador microarquitectural a nivel de ciclo que modela con precisión la GPU AMD R9 Nano (Tabla II) con el ISA Graphics Core Next 3 (GCN3). Dado que las GPUs de AMD no cuentan actualmente con unidades TMA, ampliamos MGPUSim con un modelo de TMA inspirado en la funcionalidad de Hopper [9]. De este modo, nos referimos a nuestros escenarios simulados como TMA"-Like". Nótese que, aunque nuestras simulaciones estén basadas en la AMD R9 Nano, ACTA no depende de una arquitectura en particular y puede beneficiar a cualquier GPU, independientemente del fabricante, que implemente una unidad TMA.

B. Kernels de álgebra lineal

Evaluamos ACTA usando un conjunto de kernels modernos, que abarcan un amplio espectro de cargas de trabajo computacionales (véase la Tabla I). Estos kernels son esenciales en diversos dominios de aplicación, entre ellos el aprendizaje profundo, análisis masivo de datos y la analítica, la genómica y el procesamiento de señales.

Para cada kernel, desarrollamos varias versiones (véase Figura 2), algunas empleando el TMA para reducir la latencia de memoria aprovechando el solapamiento oportuno de las transferencias con el cómputo. Nuestras observaciones muestran que las versiones con TMA logran aceleraciones significativas en comparación con sus versiones sin TMA.

C. Espacio de diseño

Para evaluar la eficacia de ACTA, llevamos a cabo un análisis detallado del espacio de diseño para las versiones con TMA. Este espacio de diseño abarca todas las posibles combinaciones de tamaños de tile y números de slots, como se resume en la Tabla I.

El número de configuraciones posibles crece de forma exponencial a medida que aumenta el número de colas y los tamaños de tile. Con el fin de acotar este estudio, limitamos nuestro espacio de diseño a opciones específicas: tamaños de tile que van desde 64 hasta 8 192 elementos y número de slots por cola de 1 a 8[3]. La regla general para calcular el número total de combinaciones se define como $(T \times S)^Q$, donde T es el número de tamaños de tile posibles, S es el número de opciones de slots para cada cola y Q es el número de colas. Esta fórmula ilustra el rápido incremento del espacio de diseño conforme aumenta Q (véase *#Combinaciones* en la Tabla I).

ACTA aborda este reto al automatizar la ardua tarea de explorar el espacio de diseño mediante unas pocas llamadas a la API del lado del host (Sección III), reduciendo la complejidad a una única ejecución del *kernel* y logrando configuraciones muy cercanas a la óptima.

V. Resultados experimentales

Los resultados de nuestra evaluación se muestran en la Figura 2, un gráfico de barras agrupadas que compara el rendimiento de seis versiones: i) *NoTMA Not-Tuned*; ii) *NoTMA Fine-Tuned*; iii) *TMA-Like Not-Tuned*; iv) *TMA-Like Informed-Tuned*; v) *TMA-Like Fine-Tuned*; y vi) *ACTA*, abarcando todos los benchmarks. Cada caso representa un nivel diferente de optimización y complejidad. Todas las implementaciones basadas en TMA utilizan las *OperandQueues*, colas, para gestionar las transferencias de memoria y el cómputo.

El eje Y del gráfico representa el rendimiento normalizado con respecto a una implementación ideal del TMA (representada por la línea horizontal roja), la cual marca el límite máximo de rendimiento. En este escenario ideal, el TMA opera con una memoria compartida ilimitada, de modo que todos los datos caben en ella y se pueden cargar de manera continua sin restricciones de memoria.

Los dos primeros casos, *NoTMA Not-Tuned* y *NoTMA Fine-Tuned*, evalúan kernels que no hacen uso del TMA. El caso de *NoTMA Not-Tuned* representa una implementación no optimizada, donde las operaciones de memoria y las de cómputo no

[3]Para kernels como *Elementwise* o *Dot-Product*, el tamaño de tile varía entre 512 y 8192, es decir, 5 posibilidades.

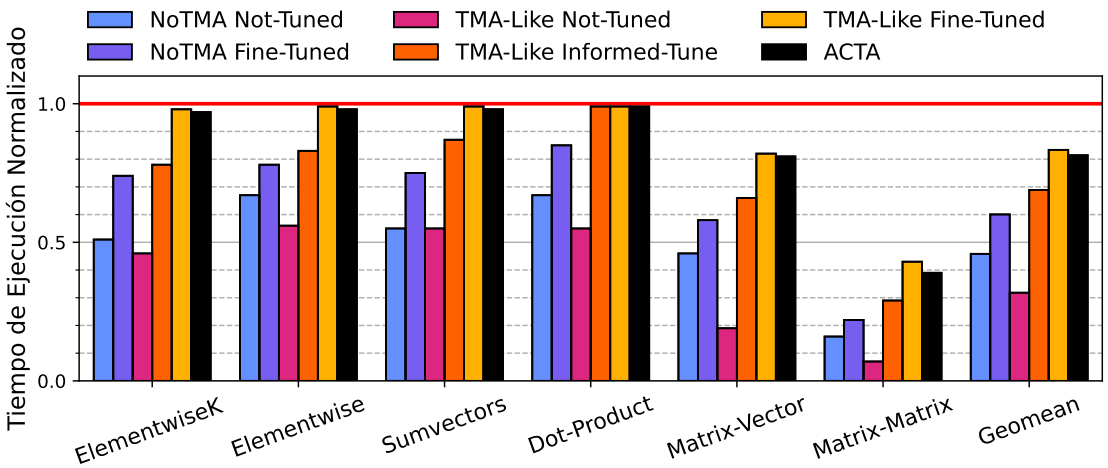

Fig. 2: Tiempo de ejecución de cada kernel normalizado a la cota de rendimiento máximo.

se encuentran necesariamente bien ajustadas. Como se observa en la Figura 2, este enfoque da lugar a un rendimiento drásticamente bajo, pues la ausencia de TMA agrava la ineficiencia de las transferencias de memoria y cómputo. En cambio, *NoTMA Fine-Tuned* aplica una exploración exhaustiva del espacio de diseño para optimizar los parámetros del kernel, mejorando significativamente el rendimiento en todos los benchmarks. Esto evidencia que, incluso sin emplear el TMA, un afinamiento cuidadoso puede lograr resultados competitivos en cargas de trabajo sencillas como *ElementwiseK*, *Elementwise* y *Dot-Product*. Sin embargo, en kernels más complejos como *Matrix-Vector* y *Matrix-Matrix*, la ausencia de TMA se convierte en un factor limitante, y el rendimiento se mantiene muy por debajo del ideal.

En cuanto a las implementaciones basadas en TMA, *TMA-Like Not-Tuned* representa un caso de referencia en el que se utiliza el TMA, pero sin configurar adecuadamente los tamaños de tile ni los slots de las colas. Este enfoque produce un rendimiento deficiente en todos los kernels. La falta de configuraciones precisas provoca un solapamiento ineficiente entre las transferencias y el cómputo, dejando recursos infrautilizados y generando diferencias notables respecto al límite superior de rendimiento.

TMA-Like Informed-Tuned incorpora configuraciones basadas en heurísticas inspiradas en recomendaciones de NVIDIA, utilizando tamaños de tile entre 64 y 256 elementos y entre 2 y 4 slots para las colas (doble o cuádruple buffering) [19]. Este enfoque ofrece un buen rendimiento en kernels más sencillos, como *ElementwiseK*, *Elementwise*, *Sumvectors* y *Dot-Product*, pero su desempeño en *Matrix-Vector* y *Matrix-Matrix* sigue siendo subóptimo debido a la mayor complejidad y demanda de recursos de estas cargas de trabajo.

La configuración *TMA-Like Fine-Tuned* supone una exploración exhaustiva del espacio de diseño para identificar las mejores configuraciones en cada kernel. Este procedimiento exige un gran esfuerzo computacional (ejecutando el kernel de la GPU una vez por configuración) y requiere afinamiento manual, pero logra un rendimiento sustancialmente mejor, especialmente en cargas de trabajo como *Matrix-Vector* y *Matrix-Matrix*. Estas configuraciones optimizadas permiten que dichos kernels aprovechen mejor los recursos de la GPU, alcanzando un rendimiento mucho más cercano al ideal. No obstante, la complejidad de este método lo hace poco práctico en la mayoría de escenarios reales (por ejemplo, como se muestra en la Tabla I, serían necesarias 2,6e+14 ejecuciones de kernel para las cargas de trabajo *Matrix-Vector* o *Matrix-Matrix* para encontrar la combinación ideal de parámetros).

Finalmente, ACTA calcula tamaños de tile y configuraciones de colas casi óptimos. Tan solo con lanzar una única vez el kernel configurado con ACTA, se elimina por completo la necesidad de un afinamiento exhaustivo. Tal y como refleja la Figura 2, ACTA alcanza un rendimiento ligeramente por debajo de *TMA-Like Fine-Tuned*, pero supera de forma consistente a *NoTMA Fine-Tuned*, *TMA-Like Not-Tuned* y *TMA-Like Informed-Tuned* en todos los benchmarks. Esto demuestra que ACTA proporciona configuraciones cercanas a la óptima con una complejidad significativamente menor, posicionándose como una solución práctica y eficiente para kernels basados en el uso del TMA.

En el caso del kernel Matrix-Matrix, todas las implementaciones (incluyendo *TMA-Like Fine-Tuned* y ACTA) se quedan muy por debajo del rendimiento ideal, ya que el escaso reuso de datos provoca cuellos de botella que impiden acercarse al escenario ideal de memoria compartida ilimitada.

A pesar de estos desafíos, como se ilustra en la categoría *Geomean*, en media geométrica considerando todos los kernels, ACTA se mantiene dentro de un **2.78 %** del caso *TMA-Like Fine-Tuned*, evidenciando su capacidad de lograr un rendimiento cercano al óptimo sin necesidad de un afinamiento exhaustivo. En cuatro kernels (*ElementwiseK*, *Elementwise*, *Sumvectors* y *Dot-Product*), ACTA ofrece un rendimiento que se sitúa dentro de un **1 %** del mejor desempeño alcanzable, mostrando su eficacia en cargas de trabajo sencillas. En kernels más complejos

como *Matrix-Vector* y *Matrix-Matrix*, ACTA logra más del 90 % del rendimiento ideal en *Matrix-Vector* y más del 35 % en *Matrix-Matrix*, ofreciendo resultados altamente competitivos a pesar de la complejidad inherente a estas tareas.

VI. Conclusiones

Al emplear ACTA, los programadores pueden aprovechar plenamente las funcionalidades hardware de última generación, como el *Tensor Memory Accelerator* (TMA), sin tener que determinar manualmente las configuraciones óptimas de cada *kernel*. Al seleccionar de forma dinámica el tamaño de los tiles y las configuraciones de cada cola en función del *kernel* y la arquitectura, ACTA abstrae la complejidad de un afinamiento manual específico y garantiza una ejecución eficiente y de alto rendimiento.

A través de nuestra evaluación, demostramos la capacidad de ACTA para proporcionar un rendimiento cercano al óptimo, reduciendo de manera drástica el esfuerzo del programador. Más allá de sus aportaciones técnicas, ACTA simplifica el flujo de trabajo, ofreciendo una solución portable y escalable, independiente de implementaciones concretas de compiladores o arquitecturas. Estas cualidades hacen de ACTA un marco práctico para explotar al máximo el potencial de las GPUs modernas sin precisar de una exhaustiva exploración del espacio de diseño.

Gracias al algoritmo que se adapta a cada kernel y a cada arquitectura GPU, ACTA asegura compatibilidad en diversos entornos. Como trabajo futuro, planeamos evaluar ACTA en múltiples arquitecturas de GPU para confirmar su portabilidad.

Agradecimientos

Este trabajo ha sido financiado por MCIN/AEI/10.13039/501100011033/ y por el "ERDF A way of making Europe" de la UE, bajo la subvención PID2022-136315OB-I00; por MICIU/AEI/10.13039/501100011033 y la "Unión Europea NextGenerationEU/PRTR", bajo la subvención RYC2021-031966-I; y parcialmente respaldado por la NSF (EE.UU.) bajo los contratos 2246035 y 2402804. Nicolás Meseguer ha sido financiado mediante la beca 21803/FPI/22 de la Fundación Séneca, Agencia de Ciencia y Tecnología de la Región de Murcia.

Referencias

[1] Intel, "Why Data Center GPUs Are Essential to Innovation," `https://www.intel.com/content/www/us/en/products/docs/discrete-gpus/data-center-gpu/what-is-data-center-gpu.html`, 2024, [Online; accessed 13-December-2024].

[2] Hajar Falahati, Masoud Peyro, Hossein Amini, Mehran Taghian, Mohammad Sadrosadati, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad, "Data-aware compression of neural networks," *IEEE Computer Architecture Letters*, vol. 20, no. 2, pp. 94–97, 2021.

[3] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama, "Verified lifting of stencil computations," *SIGPLAN Not.*, vol. 51, no. 6, pp. 711–726, June 2016.

[4] Negin Nematollahi, Mohammad Sadrosadati, Hajar Falahati, Marzieh Barkhordar, and Hamid Sarbazi-Azad, "Neda: Supporting direct inter-core neighbor data exchange in gpus," *IEEE Computer Architecture Letters*, vol. PP, pp. 1–1, 10 2018.

[5] Saba Mostofi, Hajar Falahati, Negin Mahani, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad, "Snake: A variable-length chain-based prefetching for gpus," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2023, MICRO '23, p. 728–741, Association for Computing Machinery.

[6] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri E. Bal, "Optimization techniques for gpu programming," *ACM Comput. Surv.*, vol. 55, no. 11, Mar. 2023.

[7] NVIDA, "NVIDIA Tensor Cores. Unprecedented Acceleration for Generative AI," `https://www.nvidia.com/en-us/data-center/tensor-cores/`, 2024, [Online; accessed 13-December-2024].

[8] Michele Martone, Salvatore Filippone, Salvatore Tucci, Paweł Gepner, and Marcin Paprzycki, "Use of hybrid recursive csr/coo data structures in sparse matrix-vector multiplication," in *Proceedings of the International Multiconference on Computer Science and Information Technology*, 2010, pp. 327–335.

[9] Jack Choquette, "Nvidia hopper h100 gpu: Scaling performance," *IEEE Micro*, vol. 43, no. 3, pp. 9–17, May 2023.

[10] Michael Bauer, Sean Treichler, and Alex Aiken, "Singe: leveraging warp specialization for high performance on gpus," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2014, PPoPP '14, p. 119–130, Association for Computing Machinery.

[11] Ahmed ElTantawy and Tor M. Aamodt, "Warp scheduling for fine-grained synchronization," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 375–388.

[12] Neal C. Crago, Sana Damani, Karthikeyan Sankaralingam, and Stephen W. Keckler, "Wasp: Exploiting gpu pipeline parallelism with hardware-accelerated automatic warp specialization," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 1–16.

[13] Shin-Ying Lee and Carole-Jean Wu, "Caws: criticality-aware warp scheduling for gpgpu workloads," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, New York, NY, USA, 2014, PACT '14, p. 175–186, Association for Computing Machinery.

[14] M. W. Riley, J. D. Warnock, and D. F. Wendel, "Cell broadband engine processor: Design and implementation," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 545–557, 2007.

[15] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[16] John D. C. Little, "Little's law as viewed on its 50th anniversary," *Operations Research*, vol. 59, no. 3, pp. 536–549, May 2011.

[17] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W. Keckler, Christopher W. Fletcher, and Joel Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2019, ASPLOS '19, p. 137–151, Association for Computing Machinery.

[18] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli, "Mgpusim: Enabling multi-gpu performance modeling and optimization," in *Proceedings of the 46th International Symposium on Computer Architecture*, New York, NY, USA, 2019, ISCA '19, p. 197–209, Association for Computing Machinery.

[19] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu, "Benchmarking and dissecting the nvidia hopper gpu architecture," 2024.