ACTA: Automatic Configuration of the Tensor Memory Accelerator for High-End GPUs

Nicolás Meseguer
Universidad de Murcia, Spain
n.mesegueriborra@um.es

Yifan Sun
William & Mary, USA
ysun25@wm.edu

Michael Pellauer
NVIDIA, USA
mpellauer@nvidia.com

José L. Abellán
Universidad de Murcia, Spain
jlabellan@um.es

Manuel E. Acacio
Universidad de Murcia, Spain
meacacio@um.es

Abstract

Achieving peak GPU performance requires optimizing data locality and asynchronous execution to minimize memory access costs and overlap computation with transfers. While features like the Tensor Memory Accelerator (TMA) and warp specialization address these challenges, their complexity often limits programmers.

In this work, we present ACTA (Automatic Configuration of the Tensor Memory Accelerator), a software library that simplifies and optimizes TMA usage. By leveraging the GPU Specification Table (GST), ACTA dynamically determines the optimal tile sizes and queue configurations for each kernel and architecture. Its algorithm ensures efficient overlap between memory and computation, drastically reducing programming complexity and eliminating the need for exhaustive design space exploration.

Our evaluation across a diverse set of GPU kernels demonstrates that ACTA achieves performance within 2.78% of exhaustive tuning while requiring only a single configuration pass. This makes ACTA a practical and efficient solution for optimizing modern GPU workloads, combining near-optimal performance with significantly reduced programming effort.

ACM Reference Format:
Nicolás Meseguer, Yifan Sun, Michael Pellauer, José L. Abellán, and Manuel E. Acacio. 2025. ACTA: Automatic Configuration of the Tensor Memory Accelerator for High-End GPUs. In *Proceedings of General Purpose Processing Using GPU (GPGPU '25).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction and Motivation

GPUs are nowadays fundamental building blocks in contemporary data centers, as they have been consolidated as the dominant compute platform for accelerating a myriad of application domains, including deep learning, high-performance computing, big data analytics, among others [8]. Despite their massive computational performance and memory throughput, GPUs often pose challenges to programmers in harnessing their full potential. These difficulties are particularly evident in workloads that are highly sensitive to memory latency, which significantly contribute to performance disparities [6, 9, 14, 15]. These inefficiencies often stem from the inability of programs to effectively overlap memory operations with computation, resulting in idle and underutilized GPU resources [7].

As GPU architectures continue to evolve with the addition of specialized hardware (e.g., Tensor Cores [16]) and new data formats (e.g., TF32 or FP4), aimed at enhancing application performance for specific current necessities [13], asynchronous memory transfers are becoming a standard feature in high-end GPUs. In particular, NVIDIA's Hopper (H100) architecture [3] has recently introduced the Tensor Memory Accelerator (TMA) unit, which improves data transfer efficiency by overlapping memory operations between global and shared memory with active computation. Additionally, techniques like warp specialization enable different warps to handle specific tasks within a kernel, facilitating better overlap between memory accesses and computation [1, 4, 5, 10].

Unfortunately, asynchronous memory transfers come with a significant challenge for programmers, who must not only implement the necessary data transformations but also carefully schedule memory block accesses to ensure they are available in shared memory precisely when needed. Certainly, this constitutes an important problem, as demonstrated by previous developments such as the Cell BE processor [18]. In particular, Cell BE enabled asynchronous DMA transfers between off-chip and on-chip memory in its Synergistic Processing Elements (SPEs) to bring significant performance boosts. Unfortunately, the significant challenges programmers faced in fully utilizing its capabilities, particularly in managing asynchronous memory transfers, led to the product's cancellation just a few years after its launch.

Similarly, while specialized mechanisms like TMA and warp specialization can significantly boost performance, they also add a layer of complexity that makes GPU programming substantially more challenging and harder to maintain. Consequently, successfully leveraging the new TMA units and warp specialization features requires advanced GPU skills and a thorough understanding of tools such as NVIDIA's SDK or libraries like CUTLASS, which provide high-level programming support. While these tools offer example implementations that assist in programming, adapting them to real-world GPU application kernels is often complex and requires extensive hardware knowledge.

To bridge the gap between programming productivity and application performance on high-end TMA-supported GPU platforms, we propose ACTA (Automatic Configuration of the Tensor Memory

GPGPU '25, March 01, 2025, Las Vegas, NV

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Accelerator). ACTA is a software library that heuristically determines efficient tile sizes and shared memory layouts on a per-application basis, significantly reducing programmer effort while effectively exploiting the TMA units to enhance performance in memory-latency-sensitive GPU applications. To this end, ACTA requires essential GPU specifications as its inputs, such as clock frequency, compute throughput related capabilities (e.g., number of SMs), cache hierarchy details (e.g., line size, total size, cache latencies), the total available shared memory space, the peak memory bandwidth, among others. Additionally, ACTA accounts for unique algorithmic characteristics of each GPU application kernel, such as arithmetic intensity and vector type (streaming or stationary) [2], to tailor the configuration for each kernel. Based on this information, and by relying on the Little's law, ACTA is able to infer a configuration close to the optimal for the parameters that ultimately determine the performance of a TMA-based GPU application (details in Section 3).

As a result, ACTA tradeoffs complexity and performance, offering a practical and efficient pathway to fully utilize modern GPU capabilities. Our key contributions are as follows:

- ACTA introduces a systematic approach to automate the configuration of advanced GPU mechanisms like the Tensor Memory Accelerator (TMA) Queues.
- ACTA abstracts the intricate details of TMA configuration, enabling developers to achieve high-performance implementations with minimal effort.

2 Background

Memory latency and bandwidth are critical features in modern GPU architectures, directly affecting performance and efficiency. The disparity between high-speed computational units and relatively slower memory systems creates bottlenecks, especially when handling large-scale data-intensive applications. Shared memory plays a key role in mitigating these issues by acting as a high-speed intermediate storage. However, effective utilization of shared memory depends heavily on memory access patterns. Inefficient memory management, including uncoalesced access or excessive reliance on global memory, can lead to severe performance degradation.

To improve resource utilization, warp specialization [1] assigns different roles to individual warps within a threadblock. This approach diverges from the typical expectation that all threads in a GPU execute the same instructions simultaneously. By enabling one warp to focus on specific tasks, such as managing memory transfers, while others handle computation, warp specialization introduces heterogeneity among warps. However, this scheme requires careful coordination to avoid inefficiencies, particularly in synchronizing memory and computation.

Building on this concept, the Tensor Memory Accelerator (TMA), a new specialized hardware unit introduced in NVIDIA H100 [3], extends the capabilities of warp specialization to implement an efficient producer-consumer scheme. A small number of threads (producers) manage asynchronous global-to-shared memory transfers, while most threads (consumers) focus on general-purpose computations. To orchestrate the growing number of on-chip accelerators and diverse groups of threads, TMA introduces hardware-accelerated mechanisms for synchronization, enabling consumer threads to efficiently wait for data readiness without stalling the entire execution pipeline.

TMA operations are launched through a copy descriptor, a compact structure that specifies the global memory address, tensor layout, and number of elements to copy. Once the operation is initiated by a single thread within a warp, the TMA hardware autonomously manages address generation, stride calculations, and boundary conditions, significantly reducing programmer overhead. Large blocks of data can be seamlessly transferred between global memory (GMEM) and shared memory (SMEM), optimizing bandwidth utilization and minimizing latency.

A key innovation in the TMA is its synchronization model, which introduces specialized asynchronous barriers to optimize coordination between producer and consumer threads. In particular, TMA employs asynchronous transaction barriers, which split synchronization into two phases: arrive and wait. Producer threads signal their progress by executing an arrive command when shared data is ready. This operation is non-blocking, allowing producers to move on to independent work without stalling (i.e. loading a new block of data). Consumer threads, on the other hand, issue a wait command only when they need the data, blocking until all producers have signaled their arrive. This two-step process enables early threads to utilize idle cycles for other computations, overcoming, in this way, the inefficiencies of busy-wait synchronization.

By leveraging these hardware-accelerated asynchronous barriers and transaction-based synchronization, TMA has the potential of overlapping memory transfers and computation. Unfortunately, the programmer must be directly involved to harness its full potential. Mismanagement of dependencies–such as incorrect ordering of memory operations–can lead to race conditions, deadlocks, or incorrect results, complicating the debugging process. Additionally, configuring TMA operations requires detailed knowledge of the underlying data layout and workload, demanding precision in defining parameters such as tensor dimensions and memory strides.

To reduce the programming complexity of the TMA, CUDA introduces the *cuda::pipeline* API with single and multiple stages for managing asynchronous memory operations. In our work, we implement a custom solution called OperandQueues, inspired by [17], providing a streamlined abstraction tailored to our approach.

Similar to how TMA operations are launched using a copy descriptor, Operand Queues are initialized with a queue descriptor. This descriptor defines key parameters for memory transfers, such as starting addresses, vector sizes, tile dimensions, memory strides, and shared memory destinations. Once initialized, Operand Queues automatically manage the underlying TMA copy descriptors, further abstracting the details of data movement.

In this scheme, the producer warp uses the queue to transfer data and synchronizes with consumer warps through specialized queue functions. Consumers retrieve and process data from the queue, synchronizing their operations seamlessly with memory loads. After consuming a tile, consumers signal the queue, enabling the producer to load the next tile automatically. While synchronization remains necessary, Operand Queues greatly simplify the process by providing intuitive, high-level functions that abstract low-level coordination tasks. The problem that still remains, however, is the definition of the configurations of the queues, which ultimately depend on both the characteristics of the own GPU (e.g.

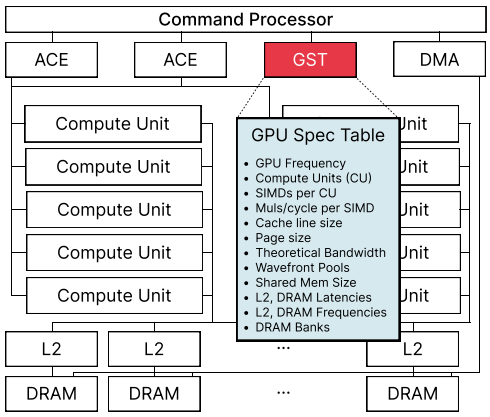

Figure 1: GPU Overview. The GST, the new hardware structure required by ACTA, is highlighted in red.

```
1  driver.MemCopyH2D(b.device_A, b.MatrixA)
2  driver.MemCopyH2D(b.device_B, b.MatrixB)
3  driver.CreateCommandQueue()
4
5  // Init ACTA for configuring the Queues
6  driver.InitACTA(MEDIUM, 8, 64)
7
8  // Register the Queues
9  driver.RegisterQueue(K, 4, TYPE_STREAMING)
10 driver.RegisterQueue(K, 4, TYPE_STATIONARY)
11
12 // Obtain the Queues sized in FIFO order
13 a_queue = driver.SizeQueue()
14 b_queue = driver.SizeQueue()
15
16 // Load kernel arguments using the QuCo
17 kernArg := KernelArgs{
18     b.device_A, b.device_B, b.device_Z, M, K, N,
19     K0, a_queue.TileSize, b_queue.TileSize, K2, M0, M1,
       M2,
20     a_queue.QueueTiles, b_queue.QueueTiles, ConsumerWfs
21 }
22
23 driver.EnqueueLaunchKernel(binary, kernArg)
```

Listing 1: High-Level Host Code Example for a Matrix-Matrix multiplication. ACTA functions are highlighted in blue.

size of the shared memory, compute bound, memory bandwidth,...) and the particularities of the application (e.g. arithmetic intensity).

3 ACTA

In this section, we introduce ACTA (Automatic Configuration of the Tensor Memory Accelerator), a software library designed to infer and provide optimal tile sizes and queue slots configurations for Tensor Memory Accelerator (TMA) kernels in high-end GPUs.

Rather than fully automating the configuration process, ACTA offers developers the critical parameters needed to efficiently initialize and use TMA Operand Queues, streamlining the kernel setup process and enhancing performance.

ACTA relies on several important architectural parameters to determine optimal configurations for kernel queues and memory operations. We assume that the values of these parameters for a particular GPU setup are accessible through a vendor-provided hardware structure which we call the GPU Specification Table (GST)[1]. Figure 1 shows the organization of the GPU model assumed in this work with the GST added. The fact that ACTA relies on the configuration parameters of the target GPU, ensures applicability across a wide range of GPU architectures, making ACTA a robust and adaptable solution.

3.1 ACTA Functionality and Workflow

Understanding the functionality of ACTA requires considering both the host-side implementation and its interaction with the GPU hardware. Listing 1 illustrates a typical example of host-side code employing ACTA [2].

On the host side, ACTA extends the GPU SDK by providing an API for dynamically configuring kernel queues. After the typical setup phase—copying data to the device and initializing command queues (lines 2-4)—the *InitACTA* function is called to prepare the library for kernel execution. *InitACTA* takes several parameters such as the kernel's arithmetic intensity (e.g., MEDIUM), the number of consumer wavefronts (e.g., 8), and the GPU's compute units to guide optimization and scheduler occupancy.

During the initialization phase, ACTA interacts with the GPU hardware to query the GPU Specification Table (GST), which is accessible through the Command Processor as shown in Figure 1. The GST contains critical architectural details such as available LDS Scratchpad (or Shared Memory in NVIDIA terminology) for the queues, computational capacity, memory bandwidth and latencies, among other (Figure 1 lists the parameters considered by ACTA). Using this information, ACTA validates these parameters against the kernel's requirements and registers kernel queues via the *RegisterQueue* function (lines 10 and 11). This process includes specifying the queue type (e.g., streaming or stationary), the length of the vector (e.g., K dimension in a matrix), and the data type size (e.g., a 4-byte float).

Then, the driver dynamically sizes the queues in FIFO order (through the *SizeQueue* function in lines 14 and 15), leveraging the information stored in the GST to determine optimal configurations, including tile sizes and the number of slots allocated to each queue. The values extracted from the GST guide the kernel optimization process, ensuring that the kernel leverages the GPU's architectural features effectively. This tight integration between ACTA and the GST allows for efficient and adaptive configuration of GPU resources tailored to specific workloads.

Finally, the host prepares the kernel arguments (lines 18-22) using the optimized configurations provided by ACTA. The *KernelArgs* structure incorporates parameters such as the tile sizes and queue configurations, directly obtained from ACTA's sizing operations. The kernel is then launched using the *EnqueueLaunchKernel* function.

3.2 The ACTA Algorithm

The ACTA algorithm is the core mechanism responsible for inferring and selecting the optimal tile size and queue slots (tiles) using

[1]In case these data are not available, ACTA could use microbenchmarking to infer these parameters dynamically.
[2]Hereafter, while ACTA is agnostic regarding GPU vendors, we adopt AMD terminology for clarity and to align with our AMD-based experimental setup.

Algorithm 1: Optimal Tile Size Calculation

Input: Range of tile sizes: `[min, max]`, Math Wavefronts, Ar.I., GST
Output: Optimal tile size
Function `optimal_tile_size()`
  for *tile* ∈ `[min, max]` **do**
    $meritFactor \leftarrow$ `evaluate(`*processing vs memory efficiency for tile*`)`;
    $costFunction \leftarrow$ `estimate(`*memory usage for tile*`)`;
    $weightedMerit \leftarrow$ `combine(`*meritFactor, costFunction to compute final score*`)`;
    if *tile is better than the best* **then**
      update $best$;
    end
  end
  $best \leftarrow$ `adjust(`*based on scaling factor and arithmetic intensity*`)`;
end

Algorithm 2: Function for calculating the Merit Factor

Input: Tile Size, GST
Output: Merit Factor
Function `evaluate()`
  `// Step 1: Compute the best-case scheduling time for processing the tile`
  $bestScheduling \leftarrow \frac{\text{TileSize}}{\text{SIMDMulsPerCycle} \times \min(\text{ConsumerWfs}, 4)}$
  `// Step 2: Calculate processing time, including scheduling roundtrip overhead`
  $procTime \leftarrow bestScheduling + (bestScheduling - 1) \times \min(\text{ConsumerWfs} - 1, \text{WfPools})$
  `// Step 3: Compute memory transfer latency and times`
  $latencyTotal \leftarrow \text{TMACycles} + \text{DRAMLatency} + \text{L2Latency}$
  $memTransferTime \leftarrow \frac{\text{TileSize} \times \text{ElementSize}}{\text{Bandwidth}}$
  $cacheTransferTime \leftarrow 2 \times \frac{\text{TileSize} \times \text{ElementSize}}{\text{CacheLineSize}}$
  `// Step 4: Aggregate memory transfer time`
  $memTime \leftarrow latencyTotal + memTransferTime + cacheTransferTime$
  `// Step 5: Return the merit factor as the ratio of processing time to memory time`
  return $\frac{\text{procTime}}{\text{memTime}}$
end

the TMA OperandQueues previously described in Section 2. This process begins by registering all queues using the *RegisterQueue* function. Registering all queues beforehand is a critical step, as it provides the algorithm with a complete understanding of the queue landscape, including the number of queues, their sizes, and their intended use.

Once all queues are registered, the *SizeQueue* function is called to compute the optimal configuration for a specific queue. If the configuration for the requested queue and kernel has already been computed, the algorithm simply returns the precomputed values. These include the tile size, the number of slots, the element byte size, and the total number of tiles for the vector. This caching mechanism eliminates redundant computations, streamlining performance. If the configuration has not been computed yet, the algorithm proceeds to determine the optimal values using a systematic approach.

The first step in the ACTA algorithm is to determine the optimal tile size (Algorithm 1). This function iteratively evaluates tile sizes over a predefined range, starting from a min, e.g. 64 elements—a fixed value representing the minimum cache line size that can be requested from memory—and extending up to a max, e.g. 8192 elements-a limit determined through our design space exploration. For each candidate tile size, it calculates a merit factor trough an *evaluate()* function, which represents the ratio of processing time to memory transfer time for a tile. Processing time accounts for compute cycles based on the kernel's arithmetic operations and wavefront utilization. Memory time incorporates key architectural parameters such as DRAM latency, cache transfer times, and bandwidth. Notably, these computations rely on GPU-specific information retrieved from the GST, ensuring that the algorithm is tailored to the hardware's characteristics (see Algorithm 2 for further explanation about the *evaluate()* function).

In addition to the merit factor, the algorithm computes a cost function to evaluate resource usage for transferring a tile, considering latency, bandwidth, and cache-line constraints. Together, the merit factor and cost function are combined into a weighted merit score, which determines the suitability of a given tile size. This ensures that the selected tile provides the optimal balance between computational efficiency and memory efficiency.

After iterating over possible tile sizes, the algorithm adjusts for the kernel's arithmetic intensity (Ar.I.): scaling up the tile size for low Ar.I. kernels (i.e., *Elementwise* or *Dot-Product*) to improve memory throughput and scaling down for high Ar.I. kernels (i.e., Matrix-Matrix) to balance memory and computation overlap. This ensures the tile size aligns with the kernel's characteristics.

The next step in the ACTA algorithm is to determine the optimal number of slots for each queue, a process handled by the *calculateOptimalNumSlots* function (Algorithm 3). This step begins by counting the number of streaming and stationary queues, as the allocation strategy prioritizes streaming queues to maximize performance, while reserving remaining resources for stationary queues.

For streaming queues, the optimal number of slots is determined using the Little's Law, which provides a relationship between the rate at which items enter a system, the time they spend being processed, and the average number of items. The observation done by Little helps ensure that resources are neither underutilized nor overwhelmed, maintaining an efficient flow, and has been widely applied within the fields of operations management and computer architecture [11]. In the context of ACTA, Little's law is adapted to calculate the ideal number of slots required for a streaming queue, considering the rate at which tiles are loaded into the shared memory by TMA operations and the total time needed to compute and process a tile.

After calculating the number of slots, the value is rounded up or down to the nearest power of two, and adjusted based on the number of compute units. Subsequently, the last step ensures that the calculated number of slots fits the available shared memory. If the slots do not fit, the number of slots is scaled based on the Ar.I. of the workload. For low Ar.I. workloads, more slots are allocated to improve memory throughput. For high Ar.I. workloads, fewer slots are chosen to reduce memory pressure and better overlap computation and memory accesses. Once validated, the slots are allocated for the streaming queues.

After finalizing the streaming queues, the former steps are repeated for the stationary queues, after the remaining shared memory is equitably divided among them.

Algorithm 3: Optimal Number of Slots Calculation

```
Input: Streaming and stationary queues, Ar.I., Compute Units
Output: Optimal number of slots for each Queue
Function optimal_num_slots()
    count streaming and stationary queues;
    if there are streaming queues then
        numSlots ← useLittlesLaw();
        numSlots ← roundToPowerOfTwo(numSlots);
        numSlots ← roundBasedOnCUs(numSlots);
        if sufficient space in Shared Memory then
            allocateSpace(streaming queues);
        end
        else
            numSlots ← useArithmeticIntensity();
            reduce numSlots if necessary to fit the data;
            allocateSpace(streaming queues);
        end
    end
    if there are stationary queues then
        calculate available space for each stationary queue;
        determine how many slots can fit into the remaining space;
        numSlots ← roundToPowerOfTwo(numSlots);
        numSlots ← roundBasedOnCUs(numSlots);
        reduce numSlots if necessary to fit the data;
        allocateSpace(stationary queues);
    end
end
```

On subsequent calls to *SizeQueue*, the precomputed tile sizes and slot configurations are reused, eliminating the need for recalculations and enhancing efficiency during kernel execution. The *InitACTA* function can be invoked multiple times before each kernel execution, providing the flexibility to schedule a series of kernels with tailored configurations (enhancing thus performance and resource utilization). Each call to *InitACTA* resets cached values, requiring ACTA to recalculate optimal queue configurations. However, this process also involves invoking the four key functions—registering queues, determining tile sizes, allocating slots, and adapting to kernel-specific requirements—to fully configure the queues for the new workload.

4 Evaluation Methodology

4.1 Simulation Environment

To quantify the performance benefits of ACTA, we use MGPUSim [19], a microarchitectural cycle-level simulator that accurately models the AMD R9 Nano GPU (Table 2) with a GCN3 Instruction Set Architecture (ISA). Since AMD GPUs do not currently incorporate TMA units, we extended MGPUSim with a TMA model inspired by the functionality of the NVIDIA Hopper's TMA. This way, we refer to our simulated scenarios as TMA"-Like". Note that, although our simulations are based on the AMD R9 Nano, ACTA is architecture-agnostic and can benefit GPUs, from any vendor, that implement a TMA-enabled platform.

4.2 Linear Algebra kernels

We evaluated ACTA using a set of popular GPU kernels representing a wide range of computational workloads (see Table 1), which are fundamental to many modern application domains such as machine learning, data science and analytics, genomics, signal processing, among others.

We developed several versions of each kernel (see Figure 2), some of which use TMA to reduce memory latency by opportunistically overlapping memory transfers with computation. Our observations show that these TMA-enabled versions achieve significant speedups compared to their non-TMA counterparts.

4.3 Design Space

To evaluate the effectiveness of ACTA, we conducted a detailed analysis of the design space for TMA-based kernels. The design space encompasses all possible combinations of tile sizes and queue slots across multiple queues, as summarized in Table 1.

The number of possible configurations grows exponentially with the number of queues and options for tile sizes and queue slots. For the purposes of this study, we constrained the design space to specific options: tile sizes ranging from 64 to 8192 elements and queue slot sizes ranging from 1 to 8[3]. The general rule for calculating the total number of combinations is given by $(T \times S)^Q$, where T is the number of tile size options, S is the number of queue slot options, and Q is the number of queues. This formula illustrates the rapid growth of the design space as Q increases (see *#Combinations* in Table 1).

ACTA tackles this challenge by automating the extensive design exploration process through a few host-side API calls (Section 3), reducing complexity to a single kernel launch while achieving near-optimal configurations.

5 Experimental Results

The results of our evaluation are presented in Figure 2, a clustered bar chart comparing the performance of six execution cases: i) $NoTMA\ Not-Tuned$; ii) $NoTMA\ Fine-Tuned$; iii) $TMA-Like\ Not-Tuned$; iv) $TMA-Like\ Informed-Tuned$; v) $TMA-Like\ Fine-Tuned$; and vi) *ACTA*, across all benchmarks. Each case represents a different level of optimization and complexity in kernel execution. All TMA-based implementations utilize OperandQueues to manage memory transfers and computations.

The y-axis of the chart represents normalized performance, relative to an ideal TMA implementation (depicted by the horizontal red line as *speed of light*) representing the upper performance bound. In this ideal scenario, the TMA operates with an unbounded shared memory, allowing all data to fit into the shared memory and enabling continuous tile loading without memory constraints.

The first two cases, $NoTMA\ Not-Tuned$ and $NoTMA\ Fine-Tuned$, evaluate kernels that do not take advantage of TMA. $NoTMA\ Not-Tuned$ represents an untuned implementation, where memory operations and computations are poorly optimized. As shown in Figure 2, this approach results in low performance, as the lack of a TMA exacerbates the inefficiency of memory transfers. $NoTMA\ Fine-Tuned$, on the other hand, applies an extensive design space exploration to optimize kernel parameters, significantly improving performance across all benchmarks. This highlights that even without leveraging the TMA, careful tuning can achieve competitive results for simple workloads like *ElementwiseK*, *Elementwise*, and *Dot-Product*. However, for more complex kernels like

[3]For kernels like *Elementwise* or *Dot-Product*, the tile size ranges from 512 to 8192, thus 5 possibilities.

Table 1: Kernels and design-space saved by using ACTA

Kernel	Description	Dimensions	# Queues	# Tiles	# Slots	# Combinations
ElementwiseK	Operations optimized for high throughput	16777216	1	5	5	25
Elementwise Sumvectors Dot-Product	General element-wise operations Addition of vectors, a basic building block in many applications Critical workload for various linear algebra workloads	16777216 2097152	2	5	5	625
Matrix-Vector Matrix-Matrix	A staple in scientific computing and data processing Fundamental for dense linear algebra and machine learning tasks	$[2048, 2048] \times 2048$ $[512, 2048] \times [2048, 128]$	8+1	8	5	2.6e+14

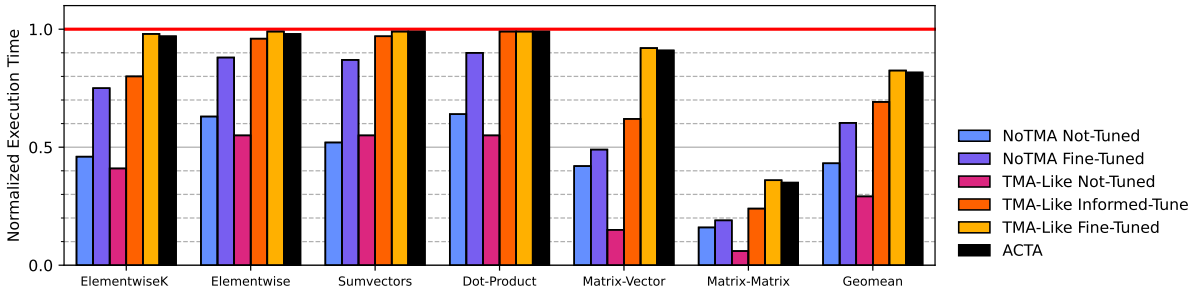

Figure 2: Kernel execution normalized to speed of light

Table 2: Specifications of the R9 Nano GPU

Parameter	Property	Amount
Frequency	1.0 GHz	-
CUs	-	64
SIMDs	64 Muls/cycle	64
L1 Vector Cache	16KB 4-way	64
L1 Inst Cache	32KB 4-way	16
L1 Scalar Cache	16KB 4-way	16
L2 Cache	256KB 16-way	16
DRAM	512MB	16

Matrix-Vector and *Matrix-Matrix*, the absence of a TMA becomes a limiting factor, and performance remains far below the ideal.

Moving to TMA-based implementations, $TMA-Like\ Not-Tuned$ represents a baseline case where the TMA is used but without proper configuration of tile sizes and queue slots. This approach results in poor performance across all kernels. The lack of informed configurations leads to suboptimal overlap between memory operations and computations, leaving resources underutilized and creating substantial performance gaps compared to the upper bound.

$TMA-Like\ Informed-Tuned$ incorporates heuristic-based configurations inspired by NVIDIA guidelines, using tile sizes between 64 and 256 elements and queue slots between 2 and 4 (double or quadruple buffering) [12]. This approach delivers strong performance for simpler kernels like *ElementwiseK*, *Elementwise*, *Sumvectors*, and *Dot-Product*, but its performance for *Matrix-Vector* and *Matrix-Matrix* remains suboptimal due to the increased complexity and resource demands of these workloads.

The $TMA-Like\ Fine-Tuned$ involves an exhaustive exploration of the design space to identify the best configurations for each kernel. This approach requires substantial computational effort (with the GPU kernel executed once per configuration) and manual tuning, but achieves significantly better performance, particularly for *Matrix-Vector* and *Matrix-Matrix* workloads. The optimized configurations allow these kernels to make better use of GPU resources, achieving performance much closer to the ideal. However, the complexity of this approach makes it impractical for most real-world scenarios (e.g., as shown in Table 1, 2.6e+14 kernel launches for *Matrix-Vector* or *Matrix-Matrix* workloads).

Finally, ACTA calculates near-optimal tile sizes and queue configurations, launching the ACTA-configured kernel only once and eliminating the need for exhaustive tuning. As shown in Figure 2, ACTA achieves performance that is slightly below $TMA-Like\ Fine-Tuned$ but consistently outperforms $NoTMA\ Fine-Tuned$, $TMA-Like\ Not-Tuned$, and $TMA-Like\ Informed-Tuned$ across all benchmarks. This demonstrates that ACTA provides near-optimal configurations with significantly reduced complexity, making it a practical and efficient solution for TMA-based kernels.

For the *Matrix-Matrix* kernel, all implementations, including $TMA-Like$; $Fine-Tuned$ and *ACTA*, fall significantly short of the ideal performance due to minimal data reuse in shared memory, resulting in bottlenecks that limit their ability to approach the performance of an idealized unlimited shared memory scenario.

Despite these challenges, as shown in the category *Geomean*, ACTA performs within **2.78%** of $TMA-Like\ Fine-Tuned$ case across all kernels, highlighting its ability to achieve near-optimal performance without the need for extensive tuning. For four kernels (*ElementwiseK*, *Elementwise*, *Sumvectors*, and *Dot-Product*), ACTA delivers performance that is within **1%** of the best achievable performance, demonstrating its effectiveness for simpler workloads. For more complex kernels like *Matrix-Vector* and *Matrix-Matrix*, ACTA achieves more 90% of the ideal performance in *Matrix-Vector* and over 35% in *Matrix-Matrix*, delivering highly competitive results despite complexity.

6 Conclusion

Using ACTA, developers can fully leverage cutting-edge GPU hardware features like the Tensor Memory Accelerator (TMA) without the burden of manually determining optimal kernel configurations. By dynamically selecting the best tile sizes and queue configurations based on the kernel and GPU architecture, ACTA abstracts the complexity of hardware-specific tuning while ensuring efficient and high-performance execution.

Through our evaluation, we showcased ACTA's ability to deliver near-optimal performance while drastically reducing tuning effort. Beyond its technical contributions, ACTA simplifies the developer's workflow, offering a portable and scalable solution that is independent of specific compiler implementations or GPU architectures. These features make ACTA a practical framework for harnessing the full potential of modern GPUs without the need for extensive design space exploration.

By adapting its tuning process to specific GPU architectures, ACTA ensures compatibility across diverse setups. As future work, we plan to evaluate ACTA across multiple GPU architectures to validate its portability.

Acknowledgements

This work has been funded by the MCIN/AEI/10.13039/501100011033/ and the "ERDF A way of making Europe", EU, under grant PID2022-136315OB-I00; by MICIU/AEI/10.13039/501100011033 and the "European Union NextGenerationEU/PRTR", under grant RYC2021-031966-I; and partially supported by NSF (US) under award 2246035 and 2402804. Nicolás Meseguer is supported by fellowship 21803/FPI/22 from Fundacion Séneca, Agencia Regional de Ciencia y Tecnología de la Region de Murcia.

References

[1] Michael Bauer, Sean Treichler, and Alex Aiken. 2014. Singe: leveraging warp specialization for high performance on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/2555243.2555258

[2] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

[3] Jack Choquette. 2023. NVIDIA Hopper H100 GPU: Scaling Performance. *IEEE Micro* 43, 3 (May 2023), 9–17. https://doi.org/10.1109/MM.2023.3256796

[4] Neal C. Crago, Sana Damani, Karthikeyan Sankaralingam, and Stephen W. Keckler. 2024. WASP: Exploiting GPU Pipeline Parallelism with Hardware-Accelerated Automatic Warp Specialization. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1–16. https://doi.org/10.1109/HPCA57654.2024.00086

[5] Ahmed ElTantawy and Tor M. Aamodt. 2018. Warp Scheduling for Fine-Grained Synchronization. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 375–388. https://doi.org/10.1109/HPCA.2018.00040

[6] Hajar Falahati, Masoud Peyro, Hossein Amini, Mehran Taghian, Mohammad Sadrosadati, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2021. Data-Aware Compression of Neural Networks. *IEEE Computer Architecture Letters* 20, 2 (2021), 94–97. https://doi.org/10.1109/LCA.2021.3096191

[7] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri E. Bal. 2023. Optimization Techniques for GPU Programming. *ACM Comput. Surv.* 55, 11, Article 239 (March 2023), 81 pages. https://doi.org/10.1145/3570638

[8] Intel. 2024. 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. [Online; accessed 13-December-2024].

[9] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. *SIGPLAN Not.* 51, 6 (June 2016), 711–726. https://doi.org/10.1145/2980983.2908117

[10] Shin-Ying Lee and Carole-Jean Wu. 2014. CAWS: criticality-aware warp scheduling for GPGPU workloads. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (Edmonton, AB, Canada) *(PACT '14)*. Association for Computing Machinery, New York, NY, USA, 175–186. https://doi.org/10.1145/2628071.2628107

[11] John D. C. Little. 2011. Little's Law as Viewed on Its 50th Anniversary. *Operations Research* 59, 3 (May 2011), 536–549. https://doi.org/10.1287/opre.1110.0940

[12] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. 2024. Benchmarking and Dissecting the Nvidia Hopper GPU Architecture. arXiv:2402.13499 [cs.AR] https://arxiv.org/abs/2402.13499

[13] Michele Martone, Salvatore Filippone, Salvatore Tucci, Paweł Gepner, and Marcin Paprzycki. 2010. 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*. 327–335. https://doi.org/10.1109/IMCSIT.2010.5680039

[14] Saba Mostofi, Hajar Falahati, Negin Mahani, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2023. Snake: A Variable-length Chain-based Prefetching for GPUs. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 728–741. https://doi.org/10.1145/3613424.3623782

[15] Negin Nematollahi, Mohammad Sadrosadati, Hajar Falahati, Marzieh Barkhordar, and Hamid Sarbazi-Azad. 2018. Neda: Supporting Direct Inter-Core Neighbor Data Exchange in GPUs. *IEEE Computer Architecture Letters* PP (10 2018), 1–1. https://doi.org/10.1109/LCA.2018.2873679

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

[17] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W. Keckler, Christopher W. Fletcher, and Joel Emer. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 137–151. https://doi.org/10.1145/3297858.3304025

[18] M. W. Riley, J. D. Warnock, and D. F. Wendel. 2007. Cell Broadband Engine processor: Design and implementation. *IBM Journal of Research and Development* 51, 5 (2007), 545–557. https://doi.org/10.1147/rd.515.0545

[19] 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. 2019. MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 197–209. https://doi.org/10.1145/3307650.3322230