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

Nicolas Meseguer[1], Yifan Sun[2], Michael Pellauer[3], Jose L. Abellan[1], Manuel E. Acacio[1]

[1]University of Murcia, Spain
[2]William & Mary, USA
[3]NVIDIA, USA

Outline

Introduction

- GPUs are the fundamental compute platform in data centers (HPC, DL, Big-Data, etc.). El Capitan #1 Top500 (June 2025) [1]
- More and more specialized hardware units (TC, RT, AMP, TMA, ...)
- Difficult to harness their full potential, specially in kernels that are highly sensible to memory latency.
- The trend is to give programmers more tools to overlap memory operations.

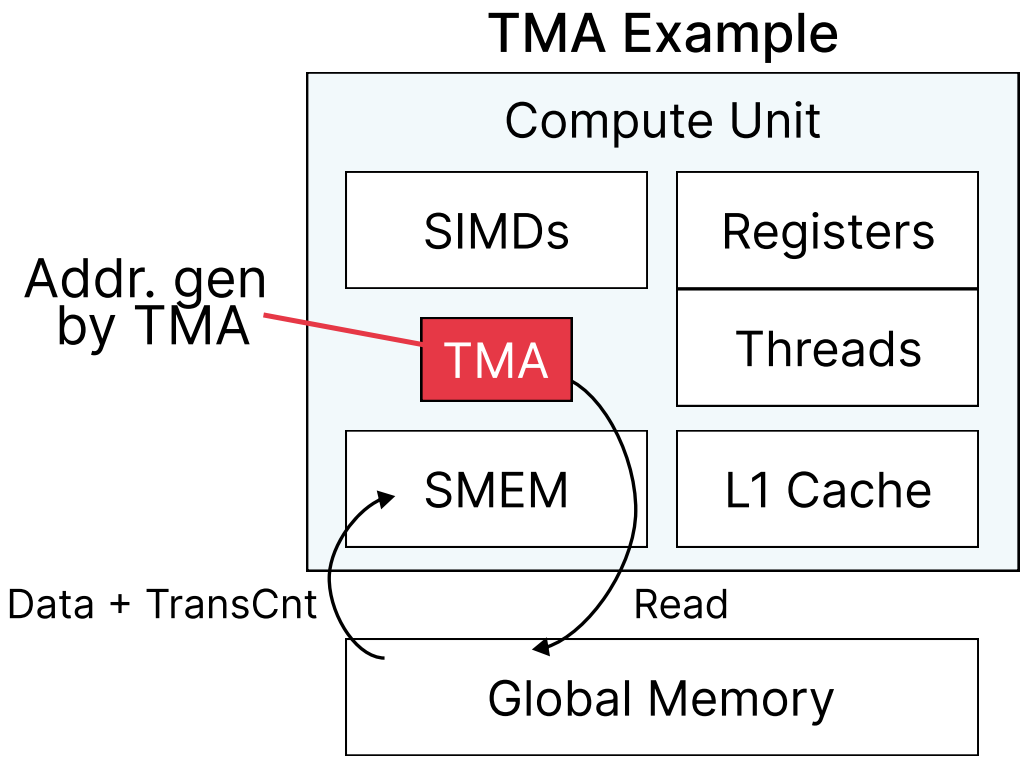
[1] 43,808 AMD MI300A GPUs

Introduction

- Techniques like Warp Specialization.
  - One warp is doing do a very specific job (divergency in gpu is bad!)
  - Usually a consumer-producer scheme
  - Synchronization is very difficult (fine-grained).
  - Usually implemented as busy-loop waiting = consumes GPU resources and further degrades the performance

Introduction

- New NVIDIA accelerator, Tensor Memory Accelerator (TMA), can transfer large blocks of data asynchronously.
  - TMA Descriptor a new data structure located in the SMEM to store different parameters: memory addresses, data length, offsets...

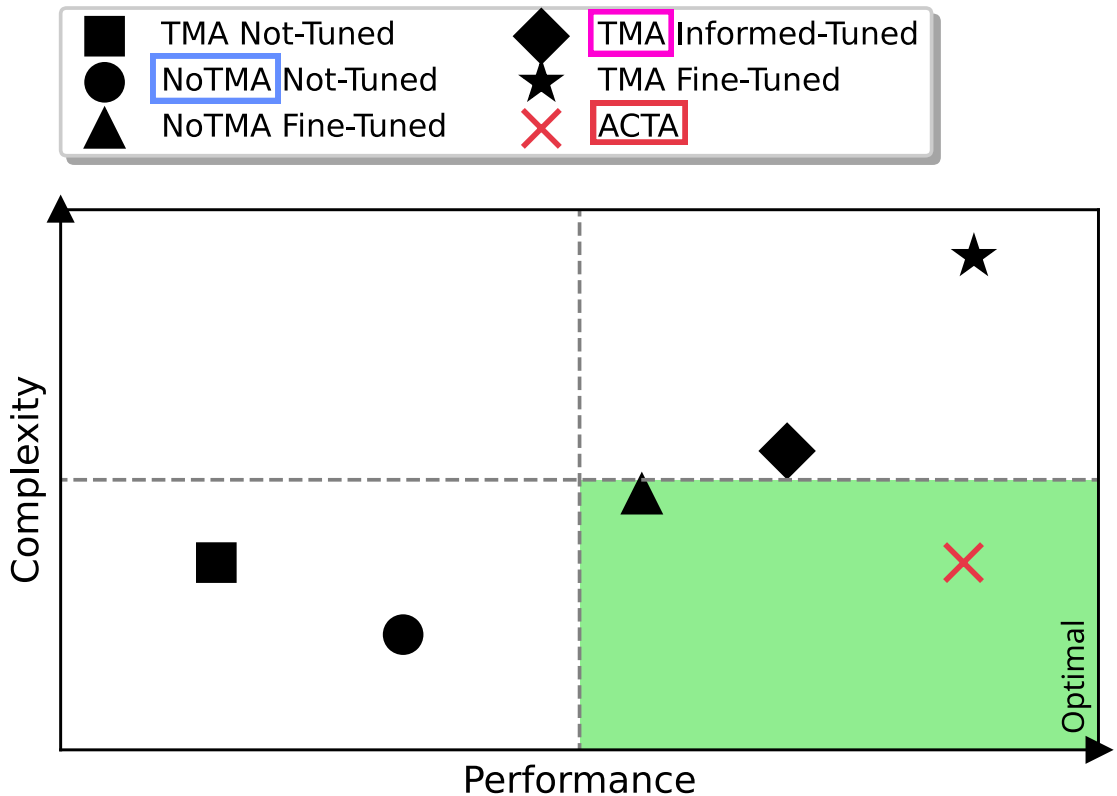


Motivation

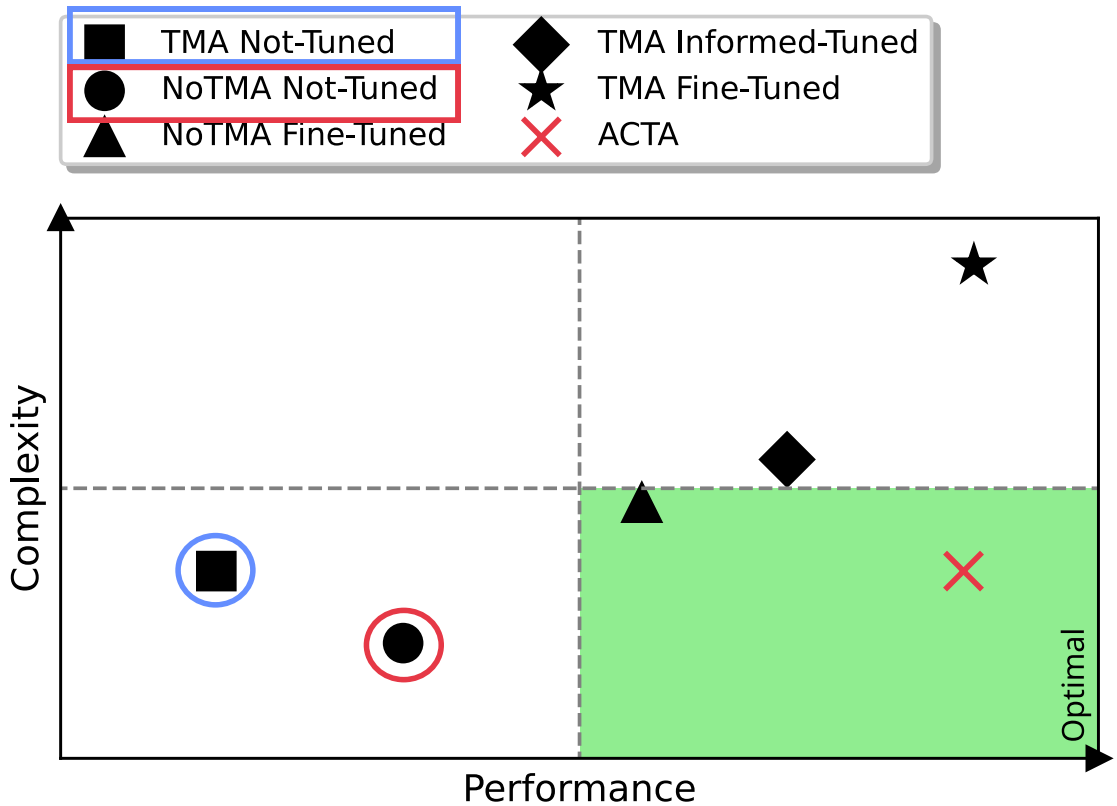
- New problem arises, **the TMA is very complex** to use.
- Queue mechanisms (OperandQueues in CUDA or Pipes in OpenCL), as a way to reduce complexity.
- Helps the programmer, but still, there are so many details left out (number of queues, SMEM addresses, managment of the queues).
- Cell BE processor was a failure due to the complexity of the DMA engine.

Motivation

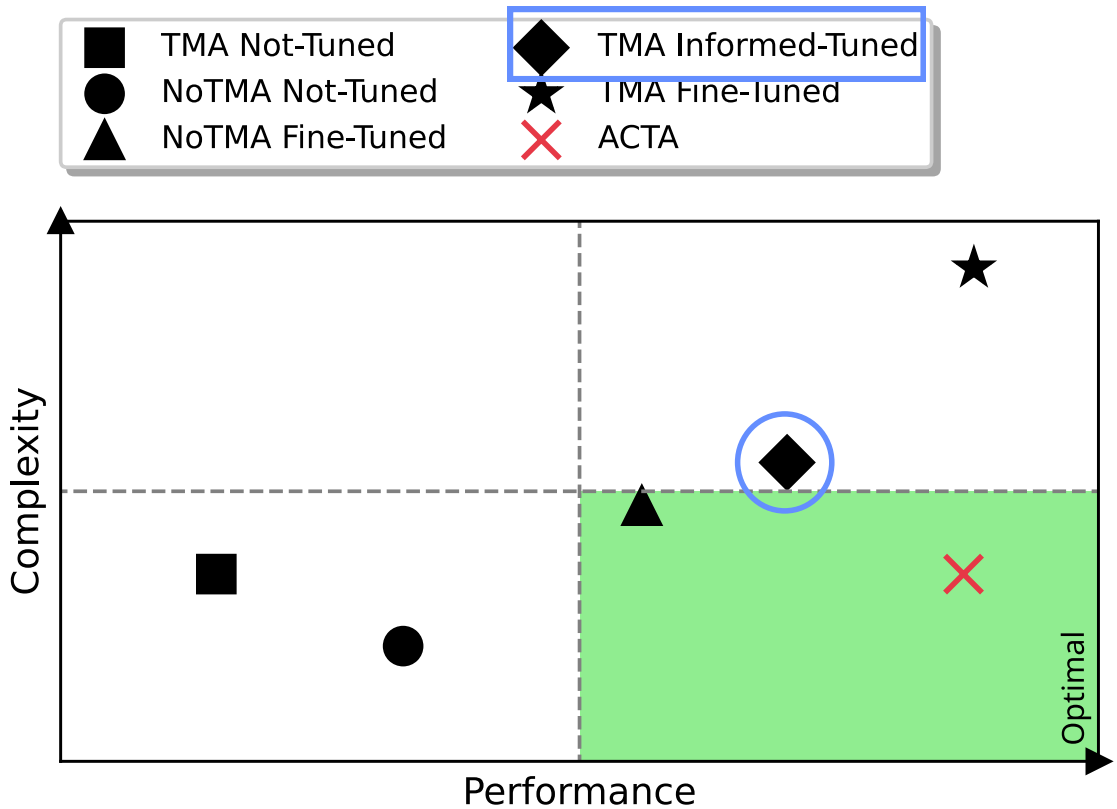
- Our goal: use the TMA with the lowest complexity and highest performance.



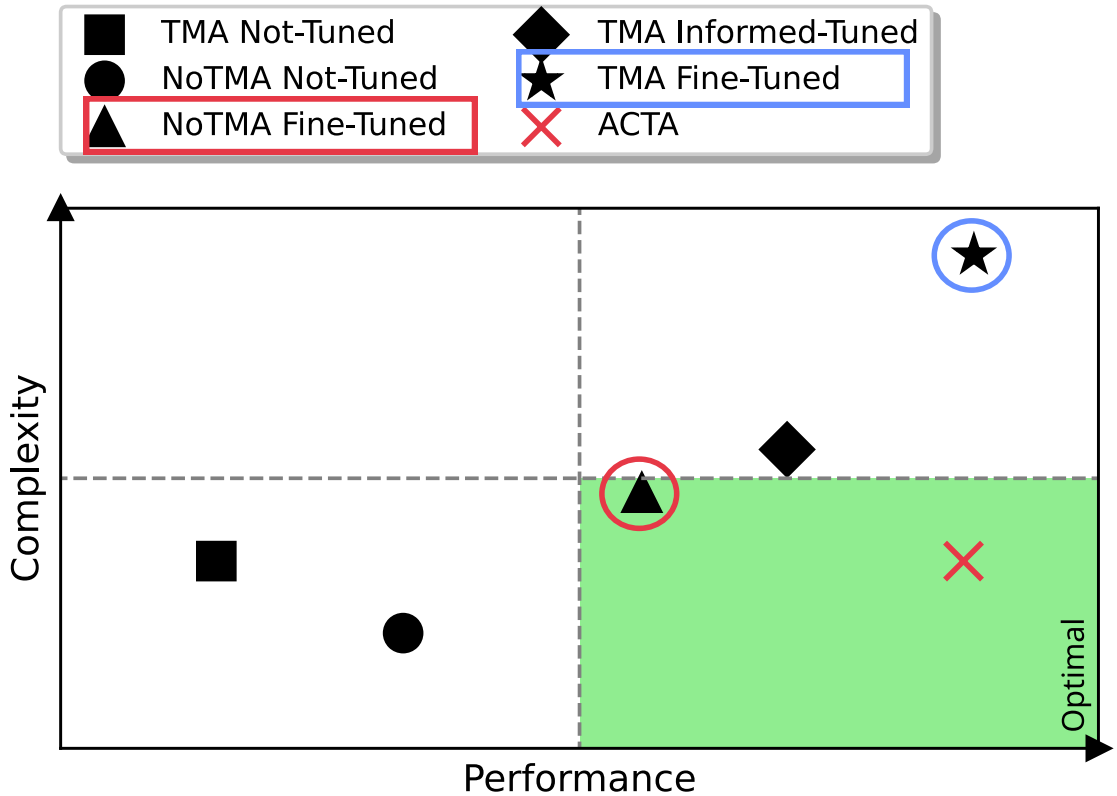
Motivation


TMA Not-Tuned
NoTMA Not-Tuned
NoTMA Fine-Tuned
TMA Informed-Tuned
TMA Fine-Tuned
ACTA
Complexity
Performance
Optimal

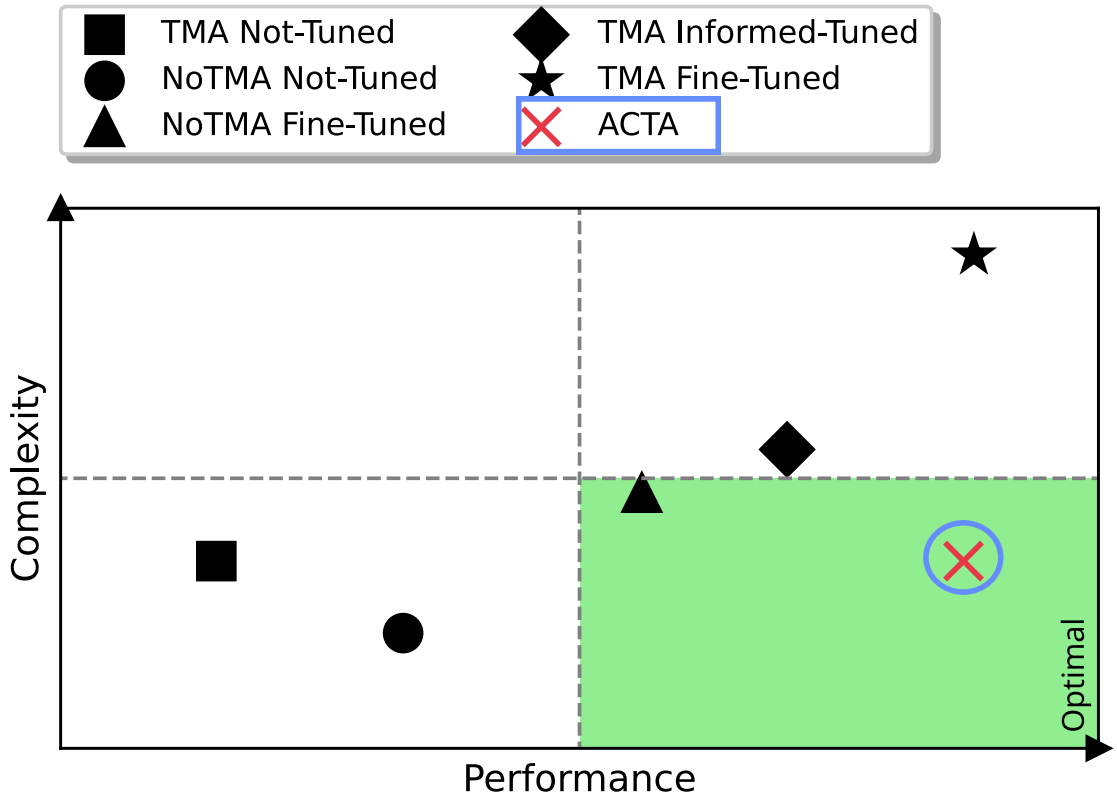
Motivation


TMA Not-Tuned
NoTMA Not-Tuned
NoTMA Fine-Tuned
TMA Informed-Tuned
TMA Fine-Tuned
ACTA
Complexity
Performance
Optimal

Motivation

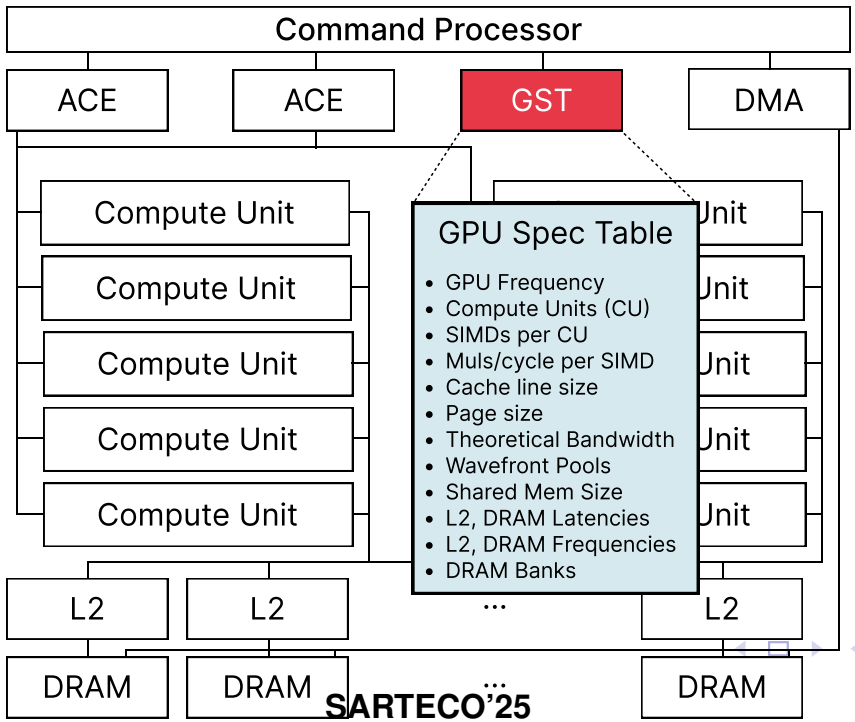

TMA Not-Tuned
NoTMA Not-Tuned
NoTMA Fine-Tuned
TMA Informed-Tuned
TMA Fine-Tuned
ACTA
Complexity
Performance
Optimal

Motivation


TMA Not-Tuned
NoTMA Not-Tuned
NoTMA Fine-Tuned
TMA Informed-Tuned
TMA Fine-Tuned
ACTA
Complexity
Performance
Optimal

ACTA

- Queue Configuration is highly kernel dependent and architecture dependent.
- Novel software library to infer optimal tile sizes and queue slots configurations for the TMA based on the kernel and gpu's architecture.
- Our new hardware unit, the GPU Specification Table (GST).



ACTA

- ACTA extends the GPU SDK by providing an API for dynamically configuring kernel queues.

```
driver.MemCopyH2D(b.device_A, b.MatrixA)
driver.MemCopyH2D(b.device_B, b.MatrixB)
driver.CreateCommandQueue()

// Init ACTA for configuring the Queues
driver.InitACTA(MEDIUM, 8, 64)

// Register the Queues
driver.RegisterQueue(K, 4, TYPE_STREAMING)
driver.RegisterQueue(K, 4, TYPE_STATIONARY)

// Obtain the Queues sized in FIFO order
a_queue = driver.SizeQueue()
b_queue = driver.SizeQueue()

// Load kernel arguments using the QuCo
kernArg := KernelArgs{
    b.device_A, b.device_B, b.device_Z, M, K, N,
    K0, a_queue.TileSize, b_queue.TileSize, K2, M0, M1,
     M2,
    a_queue.QueueTiles, b_queue.QueueTiles, ConsumerWfs
}

driver.EnqueueLaunchKernel(binary, kernArg)
```

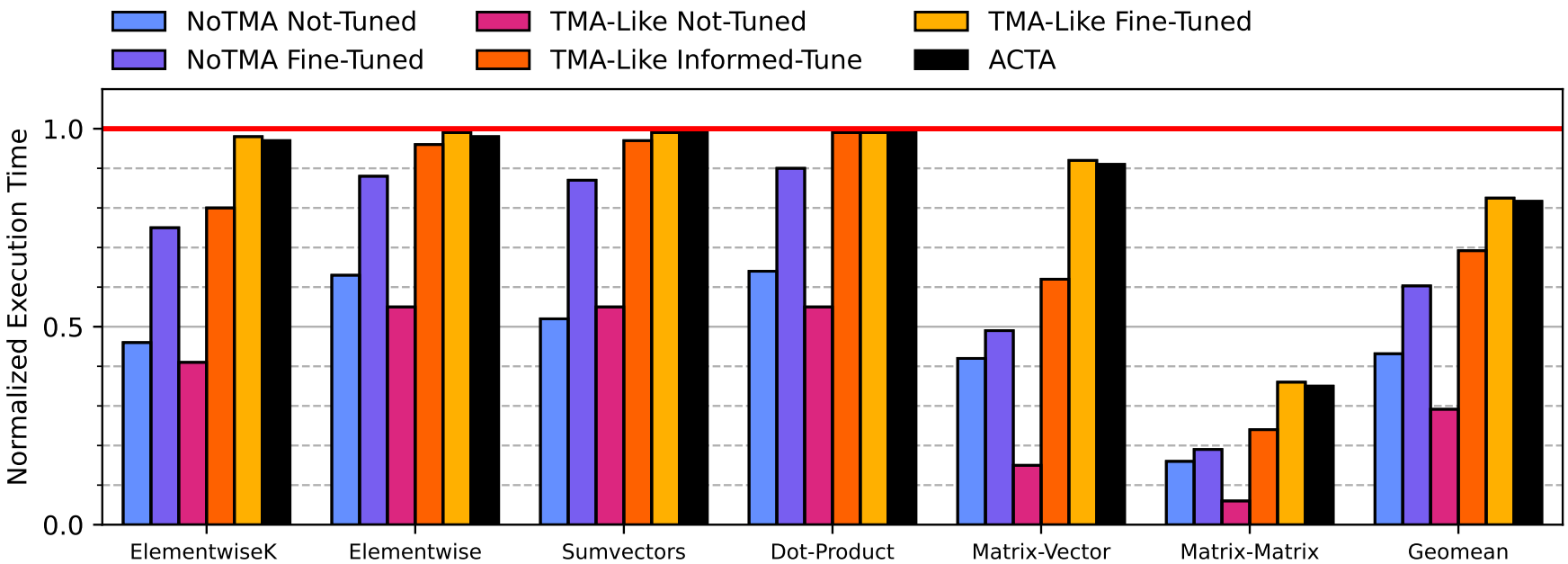
ACTA

- Optimal Tile Size Calculation
  - Using the arithmetic intensity, number of consumer wavefronts, GST, and a tile size range (i.e. 64 to 2048).
  - Merit factor to balance processing time and memory time for each tile.
  - For the given range, we calculate the most suitable tile based on the merit factor.
  - Further adjustments based on a scaling factor and the number of CUs.
- Optimal Number of Tiles Calculation
  - Differentiate streaming vs stationary to assign more or less SMEM space.
  - Little's Law fundamental relation in queuing systems, linking the average number of items in a system, their arrival rate (memory time), and their residence time (processing time).
  - Based on the arithmetic intensity, higher arithmetic intensity, more slots, lower arithmetic intensity, less slots.

Evaluation Methodology

- Implemented on MGPUSim, a microarchitectural cycle-level simulator that accurately models the AMD R9 Nano GPU, a solid baseline.
- We extended the simulator with a TMA model inspired by the functionality of the NVIDIA Hopper's TMA, we refer to ours as TMA-Like.
- Linear algebra kernels implemented (elementwiseK, elementwise, dot-product, sumvectors, matrix-vector and matrix-matrix).
- Using ACTA for matrix-matrix operations reduces the total iterations required for complete design space exploration from 2.6×10^{14} to just 1.

RESULTS


NoTMA Not-Tuned
NoTMA Fine-Tuned
TMA-Like Not-Tuned
TMA-Like Informed-Tune
TMA-Like Fine-Tuned
ACTA
Normalized Execution Time
1.0
0.5
0.0
ElementwiseK
Elementwise
Sumvectors
Dot-Product
Matrix-Vector
Matrix-Matrix
Geomean

Conclusions

- Release the programmer from the low-level details of the TMA.
- Achieve near-optimal performance (within **2.78%** compared to exhaustive tuning), with one single execution.
- Suitable across multiple GPU architectures.

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

Nicolas Meseguer[1], Yifan Sun[2], Michael Pellauer[3],
Jose L. Abellan[1], Manuel E. Acacio[1]

n.meseguerіborra@um.es

Thank you for your attention!

MINISTERIO DE CIENCIA, INNOVACIÓN Y UNIVERSIDADES
Financiado por la Unión Europea NextGenerationEU
Plan de Recuperación, Transformación y Resiliencia
AGENCIA ESTATAL DE INVESTIGACIÓN
UNIÓN EUROPEA
FONDO EUROPEO DE DESARROLLO REGIONAL
"Una manera de hacer Europa"
f SéNeCa (+)
Agencia de Ciencia y Tecnología Región de Murcia

Optimal Tile Size Calculation

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 ← evaluate(processing vs memory efficiency for
      tile);
    costFunction ← estimate(memory usage for tile);
    weightedMerit ← combine(meritFactor, costFunction to compute
      final score);
    if tile is better than the best then
        update best;
    end
end
best ← adjust(based on scaling factor and arithmetic intensity);
```

end

Merit Factor

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

Optimal Num Tiles Calculation

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
```