

ACTA: AUTOMATIC CONFIGURATION OF THE TENSOR MEMORY ACCELERATOR FOR HIGH-END GPUS

Nicolas Meseguer¹, Yifan Sun², Michael Pellauer³,
Jose L. Abellan¹, Manuel E. Acacio¹

¹University of Murcia, Spain

²William & Mary, USA

³NVIDIA, USA

OUTLINE

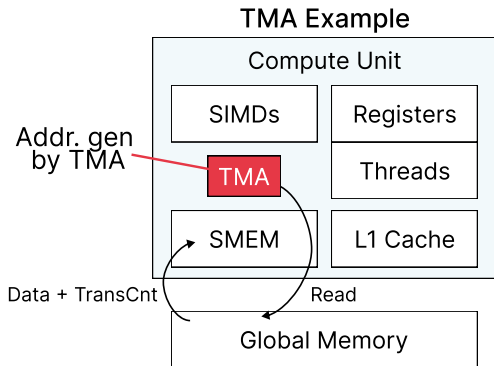
- 1 INTRODUCTION
- 2 MOTIVATION
- 3 ACTA
- 4 EVALUATION METHODOLOGY
- 5 RESULTS
- 6 CONCLUSIONS

- GPUs are the fundamental compute platform in data centers (HPC, DL, Big-Data, etc.).
- 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.

- 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

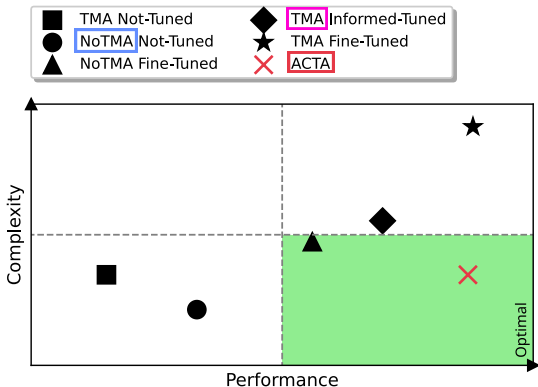
- New NVIDIA accelerator, Tensor Memory Accelerator (TMA), can transfer large blocks of data asynchronously.
 - TMA Descriptor a new data structure located in the SMEM to store different parameters: memory addresses, data length, offsets...



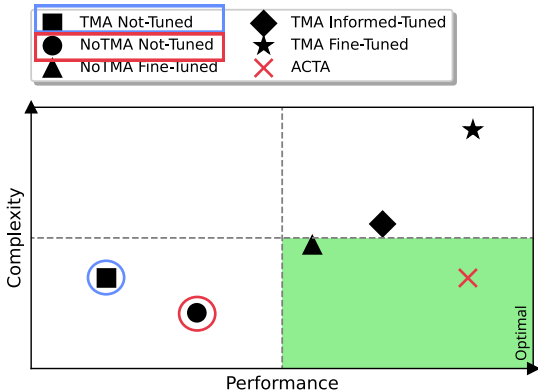
- New problem arises, **the TMA is very complex** to use.
- Queue mechanisms (OperandQueues in CUDA or Pipes in OpenCL), as a way to reduce complexity.
- Helps the programmer, but still, there are so many details left out (number of queues, SMEM addresses, management of the queues).
- Cell BE processor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

MOTIVATION

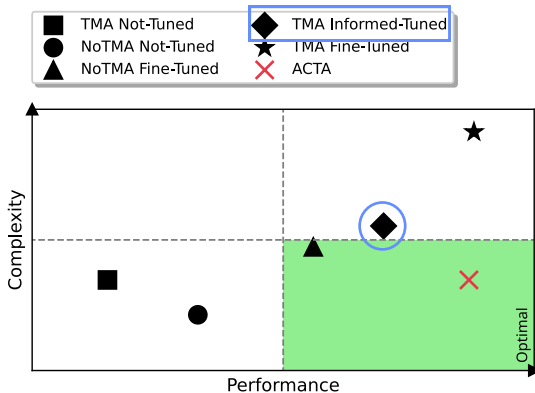
- Our goal: use the TMA with the lowest complexity and highest performance.



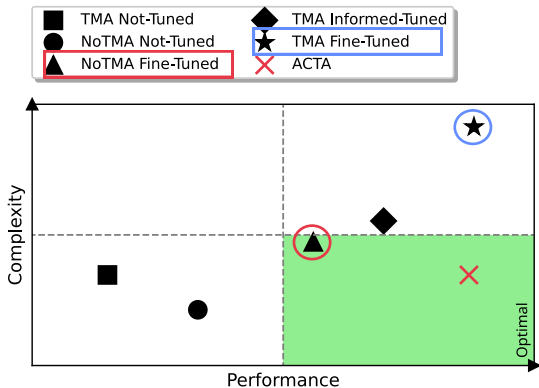
MOTIVATION



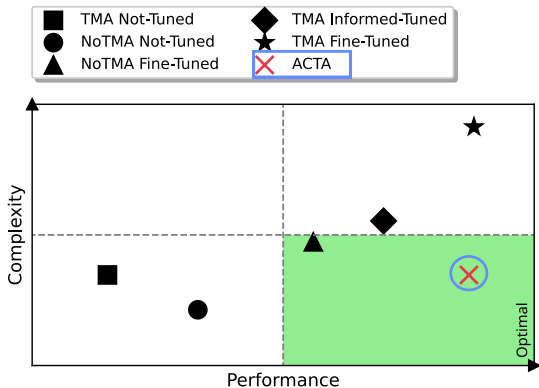
MOTIVATION



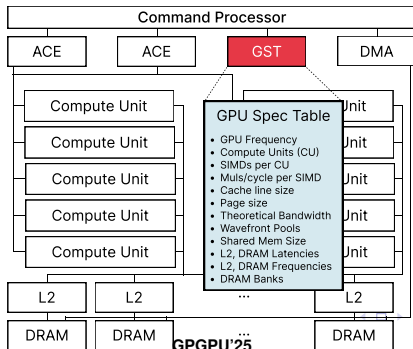
MOTIVATION



MOTIVATION



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

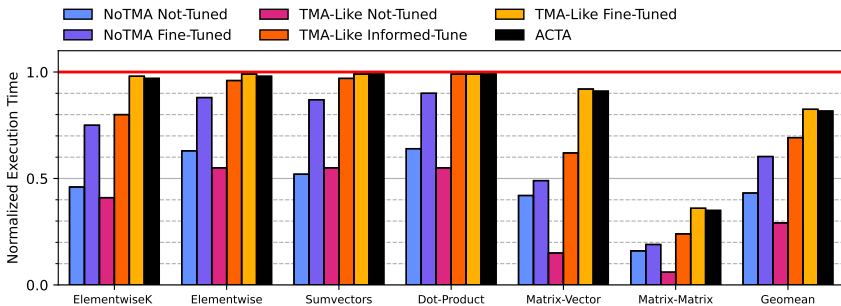
```
1 driver.MemCopyH2D(b.device_A, b.MatrixA)
2 driver.MemCopyH2D(b.device_B, b.MatrixB)
3 driver.CreateCommandQueue()
4
5 // Init ACTA for configuring the Queues
6 driver.InitACTA(MEDIUM, 8, 64)
7
8 // Register the Queues
9 driver.RegisterQueue(K, 4, TYPE_STREAMING)
10 driver.RegisterQueue(K, 4, TYPE_STATIONARY)
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,
20     M2,
21     a_queue.QueueTiles, b_queue.QueueTiles, ConsumerWfs
22 }
23 driver.EnqueueLaunchKernel(binary, kernArg)
```

- 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



- 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 Meseguer¹, Yifan Sun², Michael Pellauer³,
Jose L. Abellan¹, Manuel E. Acacio¹

n.mesegueriborra@um.es

Thank you for your attention!



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
```

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

$bestScheduling \leftarrow \frac{TileSize}{SIMDMulsPerCycle \times \min(ConsumerWfs, 4)}$

// Step 2: Calculate processing time, including scheduling
roundtrip overhead

$procTime \leftarrow bestScheduling + (bestScheduling - 1) \times$
 $\min(ConsumerWfs - 1, WfPools)$

// Step 3: Compute memory transfer latency and times

$latencyTotal \leftarrow TMACycles + DRAMLatency + L2Latency$

$memTransferTime \leftarrow \frac{TileSize \times ElementSize}{Bandwidth}$

$cacheTransferTime \leftarrow 2 \times \frac{TileSize \times ElementSize}{CacheLineSize}$

// 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 $\frac{procTime}{memTime}$

end

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 ← useLittleLaw();
    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
```