Utilizing Tensor Cores For Matmul

Global Memory Access Only

Luo, Pham, Saigal, Tangri

April 2025

An Extremely Brief History

- 1998: LeNet-5 introduces CNNs. It is limited by the hardware of its time.
- 2012: AlexNet brings GPUs to CNNs sparking a deep learning resurgence.
- 2017: NVIDIA creates tensor cores to further accelerate the matmul-heavy deep learning domain.

What are Tensor Cores?

Tensor cores are specialized **functional units** for matrix multiply and accumulate.

Functional units include CUDA cores, Load/Store Units, etc. and utilize their own pipeline.

Tensor Core Advantages

- **Massive Throughput** - theoretical 74.8 (TF32) TFLOPs vs. 37.4 TFLOPs (FP32) on A40[1]
- **Latency Hiding** - warp scheduler can assign work to free functional units.

[1] https://images.nvidia.com/content/Solutions/data-center/a40/nvidia-a40-datasheet.pdf

Tensor Advantages: Massive Throughput

$$\text{TFLOPs} = \frac{\text{Tensor Cores} \times (\text{FMA/Clock/Tensor Core}) \times 2 \times (\text{Clock/Second})}{10^{12}}$$

Tensor Advantages: Massive Throughput (cont.

A40 Specs:

- 336 Tensor Cores[2]
- 1740 MHz Boost Clock[3]
- 8x8 FMA Operations

Plugging back in, we get 74.8 theoretical TFLOPs, doubling non-tensor throughput AND matching the spec sheet!

Additionally, if we sacrifice precision, we can reach 149.7 TFLOPs!

[2] https://images.nvidia.com/content/Solutions/data-center/a40/nvidia-a40-datasheet.pdf

[3] https://www.techpowerup.com/gpu-specs/a40-pcie.c3700

Tensor Core Advantages: Latency Hiding

Warp schedulers are critical to GPU performance. Part of this responsbility is **latency hiding**.

Latency hiding - doing other work while an operation is executing.

By utilizing a separate functional unit, warp schedulers are able to schedule warps to unused units (such as CUDA cores).

How do we use Tensor Cores?

To utilize tensor cores, we can use wmma intrinsics. This is very similar to SIMD intrinsics in CPU programming.

```
// Data-type for Matrix Tile
wmma::fragment<...>;

// Load shared memory to fragment.
wmma::load_matrix_sync(...);

// Matrix Multiply AND Accumulate (C = A * B + C);
wmma::mma_sync(...);

// Store fragment to shared memory.
wmma::store_matrix_sync(...);
```

Alternatively, we can use inline PTX (but we will not today!).

Tensor Core Limitations

- Operations occur at the warp level
- Limited shapes/precision combinations by architecture [4]

[4] Read Ampere Whitepaper for relevant information on our GPUs.

Matrix Multiplication Subproblems

Unfortunately, our matrix multiplications are, oftentimes, not 16x16x16. However, we can utilize it as a subproblem in a generalized matmul.

Recall our shared tiled matrix multiplications from lecture:

```
...
for (int q = 0; q < Width/TILE_WIDTH; ++q) {
    // --- Load Logic ---
    for (int k = 0; k < TILE_WIDTH; ++k)
        Pvalue += subTileM[ty][k] * subTileN[k][tx];
    __syncthreads();
}
...
```

Matrix Multiplication Subproblems (cont.)

Let $Inc_{i,tx,ty}$ be the amount $Pvalue$ is incremented by the thread corresponding to (tx, ty) during iteration i.

$Inc_{i,tx,ty}$ maps to the dot product of the corresponding vectors (subTileM[ty][...], subTileN[...][tx]).

Thus, $Inc_{i,tx,ty}$ is equivalent to $(subTileM \cdot subTileN)_{tx,ty}$.

Mapping Loads and Outputs

Remember that tensor operations are warp granular. How can we map 32 threads to a 16x16 tile?

One potential strategy:

- Each thread is responsible for loading 4 elements per A tile.
- Each thread is responsible for loading 4 elements per B tile.
- Each thread is responsible for 4 output elements.

Sample code

```
__shared__ __half shared_input  [BLOCK_OUTPUT_SIZE][BLOCK_OUTPUT_SIZE];
__shared__ __half shared_weight [BLOCK_OUTPUT_SIZE][BLOCK_OUTPUT_SIZE];
__shared__ float  shared_result [BLOCK_OUTPUT_SIZE][BLOCK_OUTPUT_SIZE];
...
for (int i = 0; i < CEIL_DIV(C, BLOCK_OUTPUT_SIZE); i++) {
    // Load Input
    for (int j = 0; j < THREAD_WIDTH; j++) { // ROW REMAINS CONSTANT
        int loadColumn = (i * BLOCK_OUTPUT_SIZE) + j + threadWarpCol;
        shared_input[threadWarpRow][threadWarpCol + j] =
            __float2half((threadOutputRow < T && loadColumn < C) ?
                         inp[batchId * (T * C) +
                         threadOutputRow * C +
                         loadColumn
                         ] : 0
            );
    }
    ...
    A_fragment input_fragment;
    B_fragment weight_fragment;
    wmma::load_matrix_sync(input_fragment , &shared_input [0][0], 16);
    wmma::load_matrix_sync(weight_fragment, &shared_weight[0][0], 16);
    wmma::mma_sync(result_fragment, input_fragment,
        weight_fragment, result_fragment);
}
...
//Writeback to Global
```

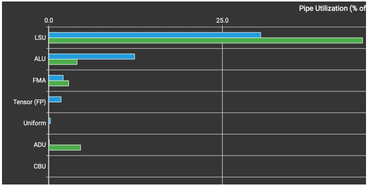
Performance Gains

We achieved $\approx$ **6.45x** gains in tokens/second over our shared memory implementation.

Despite issues with memory coalescing, we were able to achieve $\approx 77.8\%$ of cuBLAS tokens/second.

Profile Analysis: Pipeline

- Higher LSU & ADU occupancy, fewer FP-MMA operations
- Work offloaded to tensor cores, easing shared-memory pressure
- Memory bound, not compute bound: 99.55% memory throughput, 30.45% compute



Additional Notes

- **Full $\neq$ All** - Despite improved performance our, roofline profiles indicated otherwise. This is because NCU does not include tensor cores in FLOPs. Be sure to include the proper flags (beyond full).
- **Row vs. Col Major** - wmma::row_major and wmma::col_major have tradeoffs. Specifically, additional data shuffling may occur if formatted incorrectly.

Works Cited

- ECE408 Lecture Slides
- Volta White Paper
- AlexNet
- A40 Datasheet