Evaluating tensor cores

John Carlsson, Cyprien Toulon de Courtex
Department of Computer Science
University of Salerno
Italy

***Abstract*— This project was aimed at evaluating tensor cores in cuda through experiemnts to get a better understanding off high performance computing through different methods. With the ultimate goal to compare the time to calculate a matrix multiplication addition. At the start of the project, we were fairly certain that the fastest algorithm would be the tensor cores, followed by Cuda with shared memory,Cuda global memory, and at the bottom, an optimized CPU version. However, this was not the case. The results show that Cuda shared memory is the fastest with global memory just behind. For small matrices tensors appear superior.**

I. Introduction

Over the past decade, the increasing demand for deep learning and AI applications has fueled the need for highly efficient tensor operations. Traditional GPU architectures are powerful but are often limited in their ability to fully exploit the potential of tensor computations due to the reliance on higher-precision floating-point arithmetic. Tensor cores address this limitation by leveraging mixed-precision arithmetic, combining high throughput with reduced precision.

Tensor cores, introduced in NVIDIA's Volta architecture and further enhanced in subsequent architectures such as Turing and Ampere, have revolutionized the performance of tensor computations on GPUs. These specialized hardware units offer significant speedups by providing native support for low-precision tensor operations, specifically matrix multiplications and convolutions. The purpose of this project report is to present an evaluation of tensor cores, exploring their capabilities compared to CPU and GPU based computation.

In this project, we aim to evaluate tensor cores using the Matrix Multiplication Accumulate (MMA) with various workloads and compare the results to the CPU and GPU performances.

By conducting this comprehensive evaluation, we aim to provide insights into the capabilities and limitations of tensor cores. Additionally, the findings from this study will contribute to the broader understanding of GPU acceleration techniques for tensor operations and their potential impact on high-performance computing.

A. Motivation

The MMA operation plays a critical role in accelerating various computational tasks. It involves three matrices and is fundamental within many computational domains, such as deep learning, scientific simulations, and image processing. However, performing MMA efficiently and at scale is difficult due to the intensity of the operation.

The difficulties that arise when trying to implement a fast version of MMA are due to a combination of factors including memory access patterns, data dependencies, computational intensity, matrix sizes, hardware limitations, and optimization trade-offs.

The objective of this paper is to provide valuable insights for optimizing MMA computations and facilitate the adoption of tensor cores as a powerful tool for accelerating MMA operations in comparison to CPU and GPU. Our contributions will be as follows:

- We will measure the speedup achieved by tensor cores measured in actual seconds and explore how performance scales. with the size of the matrices.
- We will compare the performance of tensor cores with CPU and GPU implementations of MMA using different matrix sizes and precision settings.
- We will analyze the impact of data precision on the performance of tensor cores and investigate the trade-offs between accuracy and computation time.
- We will explore optimization techniques for maximizing the utilization of tensor cores and achieving optimal performance.
- We will provide insights into the limitations and challenges of tensor cores, including memory constraints, data dependencies, and scalability issues.

II. Background

A. Tensor cores

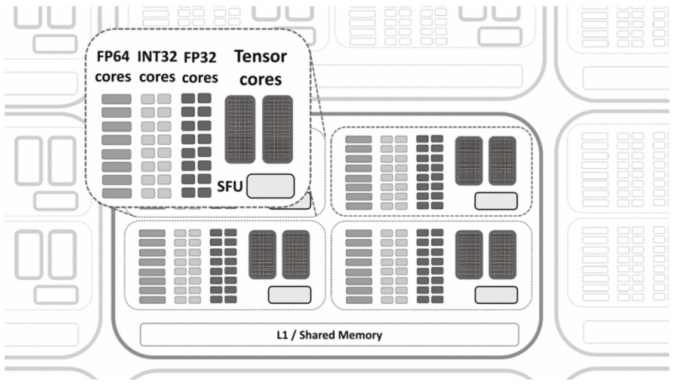

Fig. 1. SM architecture.[1]

tensor cores are specialized hardware units introduced in NVIDIA GPUs that are designed to accelerate matrix com-

putations, specifically matrix multiplications and convolutions. They leverage mixed-precision arithmetic, combining high throughput with reduced precision to achieve significant speedups in tensor operations.

The key feature of tensor cores is their ability to perform mixed-precision operations using half-precision floating-point (FP16) data types. By using FP16, tensor cores can process a larger number of elements simultaneously, leading to improved throughput at the cost of accuracy.[2]

Tensor cores operate on 4x4 matrix tiles, performing matrix multiplication and accumulation (MMA) operations. These operations are highly parallel and can be efficiently executed on tensor cores. By exploiting the parallelism of tensor cores and their ability to process multiple elements simultaneously, significant performance gains can be achieved compared to traditional GPU implementations.

The tensor operations are warp wize, this means that each tensor instructions must work on the same data within the same warp.

B. Matrix Multiplication Accumulate (MMA)

Matrix Multiplication Accumulate (MMA) is a key operation in many computational tasks, including deep learning, scientific simulations, and image processing. It involves three matrices: two input matrices (A and B) and an output matrix (C). The operation computes the matrix product of A and B and accumulates the result into C.

Mathematically, the MMA operation can be defined as follows:

$$C \leftarrow \alpha \cdot A \cdot B + \beta \cdot C$$

where A is an m x k matrix, B is a k x N matrix, C is an m x N matrix, and α and β are scalar coefficients.

The MMA operation is computationally intensive and can benefit greatly from hardware acceleration. tensor cores, with their ability to efficiently perform MMA operations, offer a promising solution for accelerating this operation.

III. Code

In this section we present the main part of the code. The code shown is the computational part. Since this is deemed most important. We show here the Cuda code and the tensor code.

A. CPU code

We implement a CPU version of the MMA operation to have a reference to verify our computation results. The CPU code is an implement of the MMA operation that uses the following optimizations:

1) Tiling
2) Tarallelisation using OpenMP
3) Compiler vectorisation for AVX512 using OpenMP
4) Compiler optimization flags

B. Cuda code

We implemented multiple versions of the matrix multiplication and addition with cuda to test out the effect of multiple optimizations techniques. For each implementation, the code is heavily simplified to keep the essence. For full code, see appendix.

1) Basic MMA in Cuda: The MMA implement in cuda is very basic as it uses the base formula of a matrix multiplication and addition and assign one thread per each element of the output matrix.

2) MMA with shared Memory: This MMA implement uses shared memory to optimize global memory acces count. The multiplication operation is done using tiling with the size of the tile matching the size of the block. For each tile, within a same block, each threads load a single element of the tile to the shared memory.

```
__global__ void MMACudaShared(float *out,
    float *A, float *B, float *c, int n)
{
   sum = c[...];
   for (tileNum = 0; tileNum <
       n/TILE_SIZE; tileNum++){
      a_shared[...] = A[...];
      b_shared[...] = B[...];
      __syncthreads();
      for (int k = 0; k < TILE_SIZE; k++)
      {sum += a_shared[...] *
          b_shared[...]; }
   }
   out = sum;
}
```

3) Mixed precision MMA: The Cuda API provides the posibility to work with floating point numbers of half the size of regular 32 bits floats. Mixed computation works with A and B being half precision floats matrices and C and The output are regular float matrices. We made two versions of this MMA Mixed precision operation based on the MMA shared memory implementation. One version convert halfs to float before doing the computation to see the effect on bandwidth and the other does the conversion after.

```
__global__ void MMACudaMixedA(float *out,
    half *A, half *B, float *c, int n)
{
   ...
      for (int k = 0; k < TILE_SIZE; k++)
      sum += (float)a_shared[...] *
          (float)b_shared[...]; }
   ...
}
__global__ void MMACudaMixedB(float *out,
    half *A, half *B, float *c, int n)
{
   ...
```

```
        for (int k = 0; k < TILE_SIZE; k++)
        sum += (float)(a_shared[...] *
            b_shared[...]); }
    ...
}
```

4) Thread coarsening Mixed precision MMA: The space gained in the shared memory is not enough to double the tile size of the shared memory. We wanted to maximize the usage of the shared memory to reduce the amount of time each thread has to wait for the memory to arrive to the shared memory between each operation. This operation is based on Mixed precision implementation.

```
__global__ void MMACoarsening(float *out,
    half *A, half *B, float *c, int n)
{
   Out accs[NUMTHREADS];
   for (subthread = 0; subthread <
       NUMTHREADS; subthread++)
      accs[subthread] = c[(row + TILE_SIZE
          * subthread)*n + col];
   for (tileNum = 0; tileNum <
       n/TILE_SIZE; tileNum++){
      for (subthread = 0; subthread <
          NUMTHREADS; subthread++)
         a_shared[subthread][...] = A[...];
      b_shared[...] = B[...];
      __syncthreads();
      for (int k = 0; k < TILE_SIZE; k++)
         for (subthread = 0; subthread <
             NUMTHREADS; subthread++)
         {suaccs[subthread]m +=
             (float)a_shared[subthread][...]
             * (float)b_shared[...]; }
   }
   for (subthread = 0; subthread <
       NUMTHREADS; subthread++)
     out[...] = accs[subthread];
}
```

C. Tensor code

The tensor code is implemented using the WMMA, Warped matrix multiplication addition, library. We are working in our case with 16*16*16 fragments that are for row major matrices.

```
using namespace wmma
__global__ void mat_mul_add_tensor(half *a,
    half *b, float *C, float *d, int N){
// Declare the fragments
fragment<matrix_a, ...> a_frag;
fragment<matrix_b, ...> b_frag;
fragment<wmma::accumulator, ..., float>
    c_frag;
load_matrix_sync(c_frag, C, N,
    mem_row_major);
// Loop over k
for (int i = 0; i < N; i += WMMA_K){
  load_matrix_sync(a_frag, a, N);
  load_matrix_sync(b_frag, b, N);
  // Perform the matrix multiplication
  mma_sync(c_frag, a_frag, b_frag, c_frag);
}
store_matrix_sync(d, c_frag, N,
    mem_row_major);
}
```

We tried to shared memory and an implementaion with the cublass library but we had terrible accuracy results and did not had time to find a solution to fix them.

IV. Experimental Setup

In this section, we describe the experimental setup used to evaluate tensor cores and compare their performance with CPU and GPU implementations of MMA. We did a multitude of tests with different parameters to see how performance scale over matrix size and how conducting many runs consequently affects performance over time.

We used square matrices with the sizes of 32, 64, 128, 256, 512, 1024, 2048, 4096 and 8192.

we conducted single comparison runs and then multiple runs to get a better average for time and error.

To understand the effect of each optimization methods, we produced different implementations, exploiting Cuda, shared memory, tensor cores, etc.

A. Hardware Configuration

The experiments were conducted on a system equipped with an NVIDIA GPU that supports tensor cores. The specific GPU model is seen below. The GPU belongs to a group of compute capability 7.5.

TABLE I
GPU Specifications[3]

GPU Model	NVIDIA Quadro 4000 RTX
Cuda cores	2304
Memory interface	256-bit
Compute capability	7.5

B. Software Configuration

The experiments were conducted using the following software tools and frameworks:

- CUDA Toolkit version 11.8
- WMMA
- C++17 compiled with g++ 11.3.0

We implemented the MMA operation in different ways: CPU, GPU (without tensor cores), and GPU (with tensor cores). The CPU implementation was written in C++, while the GPU implementations were developed using CUDA.

V. Benchmarking Methodology

To evaluate the performance of tensor cores, we performed a series of experiments using different matrix sizes and precision settings. For each experiment, we measured the execution time of the MMA operation for CPU, GPU (without tensor cores), and GPU (with tensor cores) implementations.

We varied the matrix sizes from small to large to analyze the impact of matrix dimensions on performance. Additionally, we tested different precision settings, including full precision (FP32) and mixed precision (FP16).

To ensure accurate measurements, we repeated each experiment multiple times and calculated the average execution time. We also recorded the peak memory usage during each experiment to analyze the memory requirements of tensor cores.

VI. Results and Analysis

In this section, we present the results of our experiments and analyze the performance of tensor cores compared to CPU and GPU implementations of MMA.

A. Performance Comparison

First we look at the computational time for defferent kinds of matrices, the results are plotted on a log scale on both axises. And the result show clearly that CPU is inferior to the other implementations even with optimzations. The tensor code which is not optimized is among the cuda implementaions with optimzations.

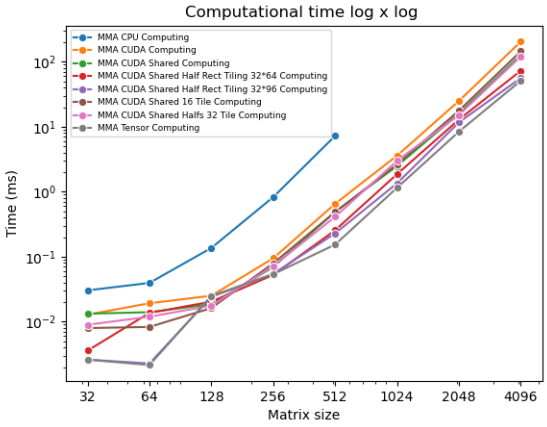

Fig. 2. Performance Comparison

The second graph shows the total time to run a instruction. Here we take into account the movement of memory. The time is divided into three parts: allocation, computation, an deallocation. Here we see that memory movement has the biggest impact on the smaller matrix sizes.

Now we look at the relative computational speedup compared to a Cuda implementaion that uses global memory. We chose to compare to the Cuda with global memory since the performance is closer to the other implementaions and a comparison to CPU would be irrelevent since all of the

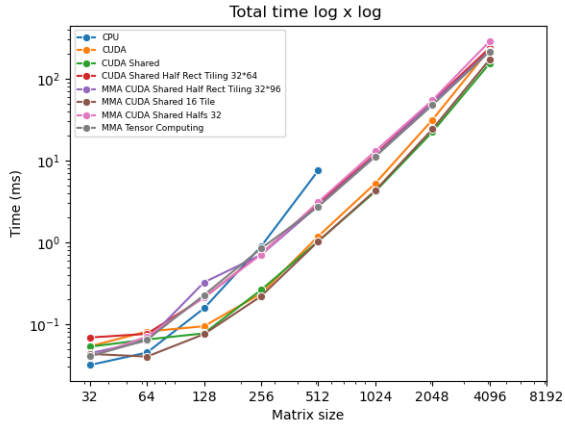

Fig. 3. Performance Comparison total time with deallocation

implementaions outscale the CPU by large. We see here that tensor cores without any speific optimzation is the fastest implementaion for all matrix sizes except for 32x32. However the Cuda implementaion with 32*96 tiling is not far behind, however as we will see in figure 5, the accuracy for smaller matrices is not good.

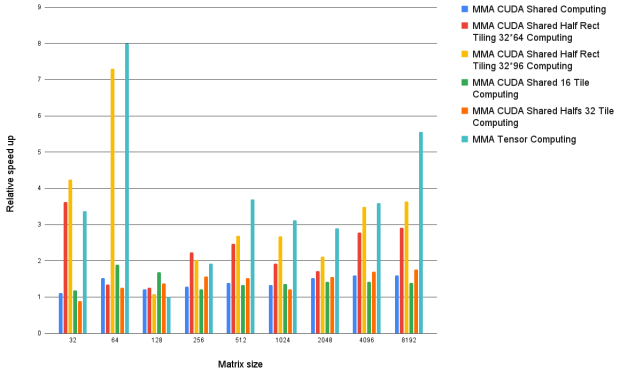

Fig. 4. Relative speedup to CUDA with global memory

B. Impact of Precision

To look at the precision of our implementations we compare to the CPU how much each value differ in percentage. We used this formula:

$$error = abs(OUT - CPU)/CPU$$

We see as previously mentioned that the Cuda with 32*96 tiling has an exceptionally high error compared to the other implementations. For smaller matrices the tiling is just not effective an will overlap and hence cause a big error. We also see that the error decreases with size. This may be attributed tho the number used in our testing, which is random floating point numbers between 0 and 1.

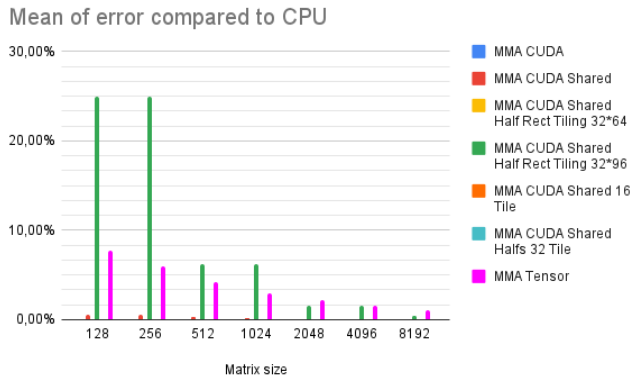

Fig. 5. Impact of Precision on tensor cores Performance

The results show that using mixed precision (FP16) with tensor cores can provide significant performance improvements compared to full precision (FP32). This is because tensor cores can process a larger number of elements simultaneously in FP16, leading to higher throughput. This accuracy also improves with time for the matrix sizes. Using FP16 then introduces a degree of numerical approximation, affecting the accuracy of the computation. Therefore, the choice of computing should be carefully considered based on the specific requirements of the application.

VII. Limitations and Challenges

The main limitations faced in this project was the limited information regarding how to program tensor cores. The cublass library, which is an optimized version of MMA with tensor cores, developed by NVIDIA is not open source, so there is no real way of obtaining it. This led to experimenting with different setups to achieve the fastest copmutation possible. Another challenge to achieve fast computation for larger matrices with tensors are the fact that the GPU only has 64Kb of memory for each work group. With the tensors, the sum of shared memory is then N*N *(16+16). Which corresponds to the matrix size, and the byte size of the elements in the A and B matrix. This entails that the largest matrices that can fit this is of 32 x 32 elements of half precision.

VIII. Conclusion

In this paper, we have explored the performance of NVIDIA tensor cores for matrix multiplication and accumulation operations. We conducted experiments to compare the performance of tensor cores with CPU and GPU implementations, considering different matrix sizes and precision settings.

From our result we can see that tensor cores without any optimization have significantly better computational power than the other, optimized approaches. It is then our belief that to optimize the tensor core version would decrease the computational time further. The total time comparison shows that allocation of memory has a significant impact on the time from input to output. Therefore using tensor cores is optimal in problen where the same data is handled continously, for example in convulational neural network and image processing. The error seems to decrease with matrix size which can be attributed to what values we are using when testing. Using only integers, the error is 0.

References

[1] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance & precision," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 522–531.

[2] P. Blanchard, N. J. Higham, F. Lopez, T. Mary, and S. Pranesh, "Mixed precision block fused multiply-add: Error analysis and application to gpu tensor cores," *SIAM Journal on Scientific Computing*, vol. 42, no. 3, 2020.

[3] Nvidia, "Volta Tuning guide," https://docs.nvidia.com/cuda/volta-tuning-guide/index.html, 2017, (Accessed: 31 May 2023).