Tensor Unit Evaluation

Arienzo Giuseppe, Coppola Felice, Squitieri Lucio
Department of Computer Science
University of Salerno
Italy

***Abstract*—In this paper, we do Tensor Unit Evaluation on two different problems: Matrix Multiplication and 2D Convolution. The goal of this evaluation is to estimate the performance boost obtained using Tensor Cores rather than CPU or GPU. First, we consider execution time with CPU and GPU and later we run the same problems with the support of Tensor Cores to obtain the execution time. Lastly, we compare all of them.**

I. Introduction

The raising markets of **AI-based data analytic** and **deep-learning applications**, have pushed several companies to develop specialized hardware to boost the performance of large dense matrix (tensor) computation, essential to both training and inferencing of deep learning applications. Recently, **NVIDIA** released the **Volta micro-architecture** featuring specialized computing units called **Tensor Cores** [1].

This specialized hardware is capable of performing one **Matrix-Multiply-and-Accumulate (MMA)** operation on a **4×4 matrix** in a single **GPU** clock cycle, in **mixed-precision** mode. More in details **Tensor Cores** take input data in **half floating-point precision**, perform matrix-multiplication and the accumulation the result in **single precision**.

Motivations behind this work. While large deep neural network applications will likely benefit from the use of **Tensor Cores**, it is still unclear how traditional **HPC** applications can exploit this technology. For this reason in this work we have done some experiments to evaluate the performance of classical problems (detailed later) on mainstream architecture such as **CPU** and **GPU** and the newly introduced **GPU** with support of **Tensor Cores**.

In order to do that, we have two challenges to fight: First, is to quantify performance improvement from using **Tensor Cores** for various problem sizes and workloads. The second, is to try to use **Tensor Cores** to execute different types of problems that are slightly different from the classic **MMA** operation.

These challenges are important in **HPC** field, because the scientists might be able to execute and obtain a very impressive speedup for resolving problems that nowadays required days or months to be resolved.

As we said, we focus on two particular problems:

- **Large matrix multiplication**. We compare the performance obtained with **Tensor Cores** against the performance obtained with **CUDA Cores**, and **CPU**, to quantify the performance boost.
- **2D Convolution** and batched it. We compare the performance obtained with **Tensor Cores** against the performance obtained with **CUDA Cores**, and **CPU**, to quantify the performance boost.

Related work. In the article '***NVIDIA Tensor Core Programmability, Performance and Precision***' [1] the authors have tested the three different ways of programming **MMA** on **Tensor Cores**, provided by **NVIDIA**: the **CUDA Warp Matrix Multiply Accumulate API (WMMA)**, **CUTLASS** a templated library based on **WMMA**, and **cuBLAS GEMM**. They found out with their results that **HPC** applications using matrix multiplications can strongly benefit from the use of **Tensor Cores**.

The article '***GPU Tensor Cores for fast Arithmetic Reductions***' [2] proposes a **GPU Tensor Cores** approach that encodes the arithmetic reduction of **n** numbers as a set of chained **m * m MMA** operations executed in parallel by **Tensor Cores**. The results obtained show that **Tensor Cores** can indeed provide a significant performance improvement to **Non-Machine Learning** applications such as the **arithmetic reduction**, which is an integration tool for studying many scientific phenomena.

In the article '***Accelerating Convolution with Tensor Cores in CUTLASS*** [3] the authors focus on implementing **2D** and **3D** convolution kernels for **CUDA** and **Tensor Cores**. They use new **CUTLASS** components that form convolution matrices, and then compute their product using the highly optimized **CUTLASS GEMM** pipeline targeting **CUDA** and **Tensor Cores**.

II. Background

Graphics Processing Unit (GPU). Is a specialized processor originally designed to accelerate **graphics rendering**. **GPUs** can process many pieces of data simultaneously, making them useful for machine learning, video editing, and gaming applications.

NVIDIA Volta Architecture. In **2017**, **NVIDIA** released **Volta GV100 GPU** architecture (Fig. 1) and the **Tesla V100** accelerator to boost **AI** and **HPC** applications. **Volta GPU** includes 21.1 billion transistors on a die area of $815mm^2$ and a renewed **Streaming Multiprocessor (SM)** design each of these is partitioned into four processing blocks, each block consists of two **Tensor Cores**, **8 FP64 Cores**, **16 FP32 Cores**, **16 INT32 Cores** and one **Special Function Unit (SFU)**.

A new feature of **Volta SM** is **mixed-precision** operations with **Tensor Cores**, in each cycle, a **Tensor Core** can perform **64 floating-point Fused-Multiply-Add (FMA)** operations [4]. An **FMA** operation takes input values in half precision

while the output values can be either in **half (FP16)** or **full precision (FP32)** as illustrated in Fig. Figure 2. **FMA** has the advantage of using only one rounding operation instead of two, resulting in a more accurate output. As we said **NVIDIA** defining feature of the new **Volta GPU Architecture** is **Tensor Cores**, which give the **Tesla V100 accelerator** a peak throughput **12 times** the **32-bit floating point** throughput of the previous-generation **Tesla P100**.

Tensor Cores enable **AI** programmers to use **mixed-precision** to achieve higher throughput without sacrificing accuracy. **Tensor Cores** are already supported for **Deep Learning training** either in a main release or via pull requests in many **Deep Learning frameworks** (including **Tensorflow**, **PyTorch**, **MXNet**, and **Caffe2**) [5].

Fig. 1: VOLTA GV100 SM

Programming NVIDIA Tensor Cores. The **NVIDIA Tensor Core** basically performs only one kind of operation: **matrix-multiply-and-accumulate on 4×4 matrices**. Therefore, a programming interface for **NVIDIA Tensor Cores** can simply express the **BLAS GEMM** (GEneral Matrix to Matrix Multiplication) operation. A **GEMM** operation consists of the multiplication of two matrices **A** and **B** and accumulation of the result into a third matrix **C**, i.e. **C = AB + C**, as you can see in Fig. 2. Here we present one interface of **Tensor Cores** to illustrate their programmability. The lowest level interface to program **NVIDIA Tensor Cores** is **CUDA 9 Warp Matrix Multiply and Accumulation (WMMA) API**. **CUDA 9 WMMA** is a **CUDA** preview feature and We briefly present it as at the moment it is the only way to program **Tensor Cores** directly and future **APIs** might be developed upon **CUDA 9 WMMA**. **CUDA 9** allows us to program a basic **MMA** on **16×16** matrices.

Listing 1: CUDA Kernel with Tensor Cores

```
1  //Declare the fragments
2  wmma::fragment<wmma::A,M,N,K, half ,wmma::col_major
       >a_fg;
3
4  wmma::fragment<wmma::B,M,N,K, half ,wmma::col_major
       >b_fg;
5
6  wmma::fragment<wmma::ACC,M,N,K, float>acc_fg;
7
8  //Initialize the output to zero
9  wmma::fill_fragment(acc_frag, 0.0f);
10
11 //Load the inputs into the fragments
12 wmma::load_matrix_sync(a_fg ,A,M);
13 wmma::load_matrix_sync(b_fg ,B,K);
14
15 //Perform the matrix multiplication
16 wmma::mma_sync(acc_fg ,a_fg ,b_fg ,acc_fg);
17
18 // Store the output
19 wmma::store_matrix_sync(acc_fg ,M,wmma::
       mem_col_major);
```

Listing 1 shows **CUDA kernel** that performs a matrix multiplication of two **16×16 matrices** with one **CUDA Warp**, that consists of **32 threads executes** a **SIMT (Single Instruction Multi Thread)**. The kernel consists of five parts. First, the **WMMA fragments** (GPU register memory for storing the input matrices) **a_fg**, **b_fg**, **acc_fg** are declared.

Second, the **accumulator fragment**, **acc_fg**, for storing the result of the matrix multiply, is set to zero. Third, the input matrices are loaded into the fragments **a_fg**, **b_fg** using **wmma::load_matrix_sync()**. Fourth, the multiplication is performed by calling the **wmma::mma_sync()**. Finally, we move the results to **acc_fg** in the **GPU global memory**. Each matrix multiplication and accumulation should be executed by one **CUDA Warp (32 threads)**. If the kernel is launched with less than **32 threads**, the result of the matrix multiplication is undetermined. On the other hand, using more **threads** than a **Warp** will still result in the correct results.

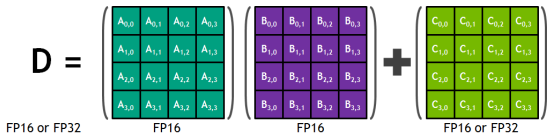

Fig. 2: Tensor Core 4x4x4 matrix multiply and accumulate.

OpenMP. Is an **application programming interface (API)** that supports multi-platform shared-memory multiprocessing programming in **C**, **C++**, and **Fortran**, on many platforms, instruction-set architectures and operating systems. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. The core elements of **OpenMP** are the constructs for **thread** creation, workload distribution (work sharing), data environment management, thread synchronization, user-level runtime routines and environment variables.

III. Proposed Method

In this paper we considered the implementation proposed by '***Programming Tensor Cores in CUDA 9***' [5] about the **MM Tensor Cores** in **CUDA 9.0**, and we perform the same operation, as we said before, with **CPU** and **GPU**. About **2D Convolution** we considered the implementation proposed by [6] for implementing this type of operation on CPU and GPU, and we tried to implement it with the Tensor Cores without the use of specialized library as showed in '***Accelerating Convolution with Tensor Cores in CUTLASS***' [3].

A. *Matrix Multiplication (MM)*

Matrix Multiplication on CPU. In Listing 2, is contained the code for **MM** on the **CPU**, with some directives from **OpenMP** to optimize the execution time.

Listing 2: Matrix Multiplication

```
#pragma omp parallel for
for (int colck=0;colck<s;colck+=16)
    #pragma omp parallel for
    for(int r=0;r<s;r++)
        for(int tile=0;tile<s;tile+=16)
            for(int t_row=0;t_row<16;t_row++)
                for(int idx=0;idx<16;idx++)
                  mr[r*s+colck+idx]+=ma[r*s+tile+
                      t_row]*mb[tile*s+t_row*s+
                      colck+idx];
```

More in details ***#pragma omp parallel for*** is a combined construct, composed by ***#pragma omp parallel*** that create a **parallel region** and ***#pragma omp for*** that divide the loop iteration space among the thread. Also, we use **loop tiling** that exploits **spatial and temporal locality** of data accesses in loop nests.

Matrix Multiplication on GPU. The code contained in Listing 3 represents the **kernel** for **MM** with **Cuda Cores**. We using the **loop tiling** technique to improve **data locality** preloading data into **local memory** before use them.

Listing 3: Kernel Matrix Multiplication

```
__shared__ float sA[B_D][B_D];
__shared__ float sB[B_D][B_D];

int tx=threadIdx.x, ty=threadIdx.y;

int Row = blockDim.y * blockIdx.y+ty;
int Col = blockDim.x * blockIdx.x+tx;

float Cvalue = 0.0;
sA[ty][tx]=0.0;
sB[ty][tx]=0.0;

for(int k=0;k<(((size - 1)/B_D)+1);k++) {
    if((Row<size)&&(tx+(k*B_D))<size)
    {
        sA[ty][tx]=mat_a[(Row*size)+tx+(k*B_D)];
    } else {
        sA[ty][tx] = 0.0;
    }
    if (Col<size&&(ty+k*B_D)<size) {
        sB[ty][tx]=mat_b[(ty+k*B_D)*size+Col];
    } else {
        sB[ty][tx]=0.0;
    }
    __syncthreads();

    for (int j=0;j<B_D;++j) {
        Cvalue+=sA[ty][j]*sB[j][tx];
    }
}
if(Row<size&&Col<size){
    res_mat[Row*size+Col]=Cvalue;
}
```

Matrix Multiplication on Tensor Core. The code in Listing 4 e Listing 5 represents the **kernel** for **MM** with **Tensor Cores**. In the Listing 4 we define **fragments** that will contain our data and in Listing 5 we perform the operation with the data declared and initialized.

Before the **MMA** operation is performed the operand matrices must be represented in the registers of the **GPU**. Since **MMA** is a **warp-wide** operation these registers are distributed amongst the **threads** of a **warp** with each **thread** holding a **fragment** of the overall matrix.

We define the **fragments a_fg** and **b_fg** for the two input matrices and **acc_fg** that will contain the result. Is important to note that **acc_fg** is set to zero with **fill_fragment**, this is how **Tensor Cores** perform a **MMA** operation.

Listing 4: Fragment definition on Tensor Cores

```
int tx=threadIdx.x,ty=threadIdx.y;
// Tile using a 2D grid
int t_row=(blockIdx.x*blockDim.x+tx)/warpSz;
int t_col=(blockIdx.y*blockDim.y+ty);

//Declare the fragments
fragment<wmma::A,M,N,K, half ,wmma::row_major>a_fg;
fragment<wmma::B,M,N,K, half ,wmma::col_major>b_fg;
fragment<wmma::RES,M,N,K, float>acc_fg;

fill_fragment(acc_fg, 0.0f);
```

In the computation phase, each **thread** of the **warp compute** a specific part of two matrices. The way how these data are mapped in each **thread** of the **warp** is explained in III-B.

Listing 5: Tensor Multiplication

```
for(int i=0;i<size;i+=K) {
    int aCol=i,aRow=t_row*M;
    int bCol=t_col*N,bRow=i;

    //Bounds checking
    if(aRow<size&&aCol<size&&bRow<size&&bCol<size
        ){
        //Load the inputs
        load_matrix_sync(a_fg,ma+(aRow*size)+aCol
            ,size);
        load_matrix_sync(b_fg,mb+(bRow*size)+bCol
            ,size);

        //Perform the matrix multiplication
        mma_sync(acc_fg, a_fg, b_fg, acc_fg);
    }
}

int cCol=t_row*N;
int cRow=t_col*M;

//Store the output
store_matrix_sync(mc+(cRow*size)+cCol,ac_fg,size,
mem_row_major);
}
```

B. 2D Convolution

Convolution [6] is an array operation where each output data element is a **weighted sum** of a collection of **neighboring** input elements.

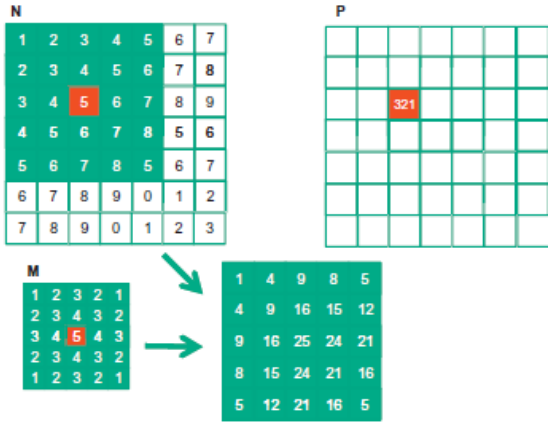

Fig. 3: 2D Convolution Example.

The **weights** used in the sum calculation are defined by an input **mask array**, commonly referred to as the **convolution kernel**. Since there is an unfortunate name conflict between the **CUDA kernel** functions and convolution kernels, we will refer to these **mask array** as **convolution masks** to avoid confusion. The same convolution mask is typically used for all elements of the array. In a **2D Convolution**, the mask is a **2D array**, its **x** and **y** dimensions determine the range of **neighbors** to be included in the weighted sum calculation.

Convolution with CPU. The code in Listing 6 represents the core part of the **Convolution** on the **CPU**, as we can see, we use **#pragma omp parallel for** that enables us to improve the performance of the algorithm. In particular, we calculate **rw** and **rc**, that represent the start coordinate in order to move inside the matrices, and after the bound checking (line 9) we can store the result calculated as showed in the Fig.3

Listing 6: CPU Convolution

```
1  #pragma omp parallel for
2  for (int row = 0; row < size; row++)
3      #pragma omp parallel for
4      for (int col = 0; col < size; col++)
5          for (int kr = 0; kr < mks; kr++)
6              for (int kc = 0;kc < mks;kc++) {
7                  int rw = row + (kr-mask_center);
8                  int rc = col + (kc-mask_center);
9                  if (rw >= 0 && rw < size &&
10                     rc >= 0 && rc < size)
11                     res[(row * size) + col]
12                     +=
13                     start[(rw * size) + rc]
14                     *
15                     mask[(kr * mks) + kc];
16             }
```

Convolution with GPU. The code in Listing 7 represents the kernel for calculation of **Convolution** on **GPU**, even in this case we used the **shared memory**, loaded in cooperative way to all thread composing the block. Also, during the loading of the elements we check the boundary (line 16) and if they are inside the bounds we load the element from starting matrix in the shared memory otherwise we load **0.0f**. After that for each element of the starting matrix it is multiplied with the elements of the mask matrix (as you can see in lines 27-29), the result obtained is then inserted in the cell of the **resulting matrix**.

Listing 7: GPU Convolution

```
1  int tx = threadIdx.x;
2  int ty = threadIdx.y;
3
4  //Coordinate result
5  int row_o = blockIdx.y * T_W + ty;
6  int col_o = blockIdx.x * T_W + tx;
7
8  // Coordinate start
9  int row_i = row_o - M_C;
10 int col_i = col_o - M_C;
11
12 // Tile in shared memory
13 __shared__ float n_ds[T_W + M_S * M_S - 1][T_W +
       M_S * M_S - 1];
14
15 // Tile cooperative upload
16 if ((row_i >= 0) && (row_i < s) && (col_i >= 0)
       && (col_i < s))
17     n_ds[ty][tx] = m_st[(row_i * size) + col_i];
18 else
19     n_ds[ty][tx] = 0.0f;
20
21 __syncthreads();
22
23 if (tx < T_W && ty < T_W) {
24     float output = 0.0f;
25     for (int i = 0;i < M_S;i++)
26         for (int j = 0;j < M_S; j++)
27             output += mask[i * M_S + j] * n_ds[i
                   + ty][j + tx];
28     if (row_o < mat_size && col_o < mat_size) {
29         mat_res[row_o * mat_size + col_o] =
               output;
30     }
31 }
```

Convolution with Tensor Core. To do a convolution that takes advantage of the **NVIDIA Tensor Cores**, we need to find a way to do the convolution as matrix multiplication; to do that we considered two strategies: The first one consists to transform the **input matrix** in a **Toeplitz matrix**, as showed in Fig. 4. After that, we multiply it with a **mask** represented as a **vector**. The main problem of this approach is the additional space and overhead needed to create the **Toeplitz matrix**, another problematic to consider is that the **Tensor Cores** only works on **square matrix**, so we need to add the necessary space and computation overhead to add a **padding** to the **mask** (to transform a generic matrix into a square one).

The second strategy consists to calculate the transpose of the **mask**, and multiply it with for each **sub matrix** of the **original matrix**. More in details, if we have a **starting matrix** of size **M * N** and a mask of size **K * K** where **k** is an odd number; We multiply each element of **starting matrix** for the **mask matrix** with the center in that element (for example if we're calculating the element in the ***position [2,2]*** of **starting matrix**, we considered the **mask matrix** obtained from that coordinate as showed in Fig. 3). After that we sum the **major**

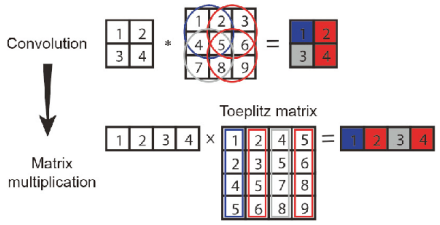

Fig. 4: Convolution as Toeplitz Matrix Multiplication

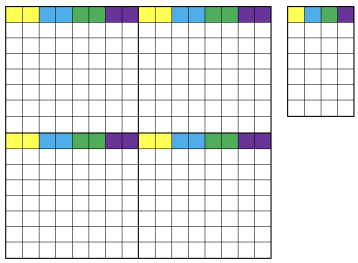

Fig. 5: Data Mapping on Tensor Cores

diagonal of **resulting matrix** and this value will be the result of **convolution** for the specific initial value of **starting matrix** (in our example was the element in position ***[2,2]***).

Even if the second strategy is more elegant than the first one, it poses a problem from the point of view of **data mapping**. In fact, we have discovered that data are mapped in a quite particular way inside the **CUDA cores** when you use **WMMA API** and **Tensor Cores**, as showed in ***Cuda Sample*** [7]. Each block of our **block grid** must be composed on the **x-axis** of a number of thread multiple of **32** (one warp) because each warp is able to compute a **MMA** for a matrix of size **16 * 16**.

Now, we don't know exactly how many **Tensor Cores** are mapped in a warp, because **NVIDIA** doesn't provide this information in the documentation. After some test we have discovered that thread that compose a **warp** share a single **Tensor Cores**. The proof of this is contained in how data are loaded in a **WMMA fragment** that are distributed among the threads that compose a **warp**.

In fact, if we assume to have a matrix of size **16 * 16** and a block composed of **32 threads** on the **x-axis** and **1 thread** on the **y-axis** (a grid of 1 * 1 wrap) data are mapped as showed in Fig. 5

Practically the start matrix is splitted in **4 submatrix** each of **8 * 8**, after that each **thread** of the **warp** obtains a couple of element. More in details the **threads** in position **0** of **warp** obtains first two elements (yellow cells) of **submatrix** in position **[0,0]**, first two elements of **submatrix [0,1]**, **[1,0]** and **[1,1]** and so on. The same for all other **thread** that compose the **wrap**.

This mapping strategy is very tricky and makes very complex apply the second strategy, because we don't have control on how the data is mapped to the **threads**, it's really difficult to develop a **kernel** capable to iterate on each element of the **starting matrix**. For this reason, it was decided not to present any complete algorithm for the realization of the **convolution** on **Tensor Cores** which is postponed to future and more accurate developments. We would like to add that the creation of a **kernel** to do a **2D convolution** is not easy to do without the use of specific libraries developed by **NVIDIA** itself [3], indeed all of the solutions found on internet always use the library.

IV. Experimental Results

In this section, we go in depth about the results of our experiments and analyze the machine and tools that we utilized:

Experimental setup.

- **CPU**
  - **Model name**: AMD Ryzen 5 3600
  - **CPU(s)**: 6
  - **Socket(s)**: 1
  - **Core(s) per socket**: 6
  - **Thread(s) per core**: 2
  - **Cache L1d**: 384 KB
  - **Cache L2**: 3 MiB
  - **Cache L3**: 32 MiB
- **Memory RAM**
  - **Size**: 16 GB
- **Operating System**
  - **Ubuntu Server**: 20.04
- **GNU compiler for host code**: 4.8.5
- **NVCC compiler flag**: -O3 -Xptxas -v -std=c++11- gencode arch=compute_75, code=sm_75 -gencode arch=compute_75, code=compute_75
- **GPU**
  - **Model name**: GeForce RTX 2060
  - **Shading Units**: 1920
  - **TMUs**: 120
  - **ROPs**: 48
  - **SM Count**: 30
  - **Tensor Cores**: 240
  - **RT Core(s)**: 30
  - **Thread(s) per core**: 2
  - **Cache L1**: 64 KB per SM
  - **Cache L2**: 3 MiB
  - **Memory**
    - **Size**: 6 GB
    - **Type**: GDDR6
    - **Bus**: 192 bit
    - **Bandwidth**: 336.0 GB/s

We used a **C++** script to automatize our work. For all our benchmarks, we used **square matrices** with these sizes: **1024**, **2048**, **4096**, **8192** and **16834**. For each size it runs **5 tests** with **CPU**, **GPU** and **Tensor Cores** for **MM** and with **CPU** and **GPU** for **2D Convolution**. At the end of the **5 executions**, it produces a **Report.txt** file with the mean of the execution time. Furthermore, for what concern **convolution** we use a square matrix of **size 5** that represents the **convolution mask**.

Results. Here the following execution timings are represented for every experiment that has been done, we report a mean after **5** iterations for each experiment.

Matrix Multiplication (MM) experiments.

MM on CPU execution time	
Input Size	Mean (ms)
1024	40,72
2048	545,93
4096	20896,34
8192	203777,52
16384	170707903,63

TABLE I: Results for the **CPU** execution for **MM** of the previous benchmark. As you can see execution times increase as the size increases, and for **16384** we get the worst execution time.

- **MM computation time GPU and TCores**:

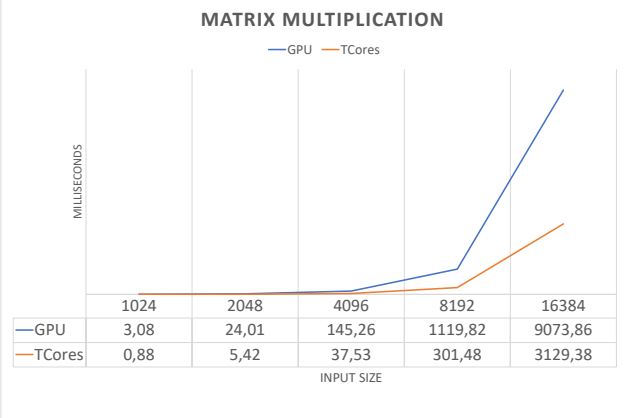

Fig. 6: Matrix Multiplication timings

In Fig. 6, the execution time on **GPU** and **Tensor Cores** it increases but it is still an acceptable time and an indication of good performance, because we need about **9073 ms (9 secs)** and **3138 ms (3 secs)** to execute a **MM** on **16384 * 16384 matrix** while on **CPU** we need **170707903,63 ms (1707 secs)**, it is an huge speedup. As expected with the use of **Tensor Cores** to execute **MM** we get the best results and this means that they can be used for problems that can take advantage of using this type of computation.

Convolution experiments.

- **Convolution computation time**
  In Fig. 7, execution times increase as the size increases but still with acceptable performance on both **CPU** and **GPU**. Although the **GPU** is more performing than the **CPU** as expected.

V. Conclusions

The results obtained for the problems seen in this experiment, shows us that they works better on **Tensor Cores** and

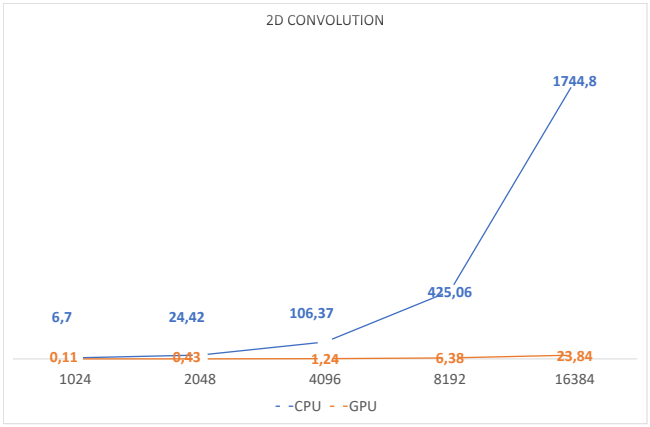

Fig. 7: Convolution timings

GPU than **CPU** (even though we used **#pragma omp parallel for** and **tiling** for increasing the performance). Even though we weren't able to do **2D Convolution** with **Tensor Cores** (for the reasons explained previously), we have seen the improvement of execution time and performance with their use for **Matrix Multiplication**. This means that there are some opportunities to utilize **Tensor Cores** in the **HPC** field, that was unexplored at nowadays.

References

[1] Erwin Laure Stefano Markidis Steven Wei Der Chien. *NVIDIA Tensor Core Programmability, Performance and Precision*. 2018. URL: https://arxiv.org/pdf/1803.04014.pdf.

[2] Cristóbal A. Navarro et al. *GPU Tensor Cores for Fast Arithmetic Reductions*. 2021. DOI: 10.1109/TPDS.2020.3011893.

[3] Manis Gupta NVIDIA. *Accelerating Convolution with Tensor Cores in CUTLASS*. 2021. URL: https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s31883/.

[4] NVIDIA. *NVIDIA TESLA V100 GPU ARCHITECTURE*. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[5] Jeremy Appleyard Scott Yakim. *Programming Tensor Cores in CUDA 9*. 2017. URL: https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/.

[6] Wen-mei W. Hwu David B.Kirk. *Programming Massively Parallel Processors*. 2017.

[7] Rutwik Choughule. Andy Dick Mahesh Doijade. *Cuda Sample*. 2020. URL: https://github.com/NVIDIA/cuda-samples.git.