Making GPU Tensor Cores compute texture interpolation

Maxence Doktorcik

To cite this version:

Maxence Doktorcik. Making GPU Tensor Cores compute texture interpolation. Computer Science [cs]. 2025. ⟨hal-05185903⟩

HAL Id: hal-05185903

https://inria.hal.science/hal-05185903v1

Submitted on 25 Jul 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

Making GPU Tensor Cores compute texture interpolation

Maxence Doktorcik
Master 1, 13/06/2025
Université Grenobe Alpes, Maverick Team at INRIA / LJK
maxence.doktorcik@etu.univ-grenoble-alpes.fr

Supervised by: Antoine Richermoz and Fabrice Neyret.

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature: Maxence Doktorcik, 19/03/2025

Abstract

Texture interpolations is computationally heavy. This paper aims at seeing if tensor cores, which are extra hardware, could be used for computing, with a reasonable performance, texture interpolations that are not possible under the texture unit, as well as to see if an improvement can be achieved compared to the texture unit's already existing interpolations.

1 Introduction

Over the past years, we have seen video games taking an important place in our lives. Every year, new advances on GPUs (Graphics Processing Units) are made in order to make video games look better while increasing the performance. In order to allow quality along with performance, Graphics Cards are now equipped with a texture unit allowing the storage and fast computations on textures which are typically 2D maps filled with color values that are used to map onto 3D objects. The texture unit allows relatively fast computation of texture interpolation, for which the goal is to fill missing values when resizing an image or approximating a color based on several pixels that surrounds it [Gonzalez and Woods, 2002].

Even though NVIDIA provides a texture unit with functions to do bilinear interpolation on textures that are efficient [NVIDIA, 2025a], we do not have a built-in function for bicubic, higher dimensions, tiled textures or α-multiplied interpolations so that we aim to do those calculations without calling multiple bilinear functions[Briand and Davy, 2019] and relatively fast. Since interpolation is very heavy on matrix multiplications, the purpose of this study is to evaluate the use of tensor cores, which are made for fast matrix calculus of the form matrix = matrix $\times$ matrix $+$ matrix, in order to do all kind of interpolations that we previously mentioned and compare the performances.

Unfortunately, there is not many documentations about tensor cores aside from how to send a matrix to them, how to use them in libraries[Raihan *et al.*, 2019] or a try at demystifying the tensor cores [Yan *et al.*, 2020], so that we could discover features on top of using cores that are, as today, mostly used in deep learning [Burgess, 2020]. Such utilization could then transform an element of the hardware that is almost never active into a helper for heavy interpolation workload.

2 Interpolation Techniques

There exists several interpolation techniques and algorithms seen in this study. There are also plenty of other techniques and algorithms that this paper does not take into account, as they do not match with matrix multiplication.

2.1 Bilinear interpolation

Mathematical formulation

Consider a function $f(x, y)$ defined at four grid points forming a unit square:

$$(x_1, y_1), (x_1, y_2), (x_2, y_1), (x_2, y_2)$$

where $x_2 = x_1 + 1$ and $y_2 = y_1 + 1$, and the function values at these points are denoted as: $f_{11} = f(x_1, y_1)$, $f_{12} = f(x_1, y_2)$, $f_{21} = f(x_2, y_1)$, $f_{22} = f(x_2, y_2)$.

For an arbitrary point (x, y) within the square, bilinear interpolation estimates $f(x, y)$ based on these values by the following:

$$f(x,y) \approx \frac{1}{\alpha} M \begin{bmatrix} x_2y_2 & -y_2 & -x_2 & 1 \\ -x_2y_1 & y_1 & x_2 & -1 \\ -x_1y_2 & y_2 & x_1 & -1 \\ x_1y_1 & -y_1 & -x_1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ x \\ y \\ xy \end{bmatrix} \quad (1)$$

where $\alpha = (x_2 - x_1)(y_2 - y_1)$ and $M = [f_{11} \quad f_{12} \quad f_{21} \quad f_{22}]$

Note that the matrix format is used to correlate with the use of tensor cores.

Algorithm

The algorithm for bilinear interpolation can be described as follows for a given fractional location (x,y):

1. Identify the four neighboring grid points (pixels in this case) (x_1, y_1), (x_1, y_2), (x_2, y_1), and (x_2, y_2).
2. Retrieve the corresponding function values $f_{11}, f_{12}, f_{21}, f_{22}$.

3. Compute α
4. Compute the interpolated value using Equation (1).

The computational complexity of this algorithm is up to the complexity of the matrix multiplication used for one position, as it requires a fixed number of arithmetic operations regardless of the grid / texture size.

2.2 Bi-cubic interpolation

Bi-cubic interpolation allows for a smoother interpolation while it is computationally heavier. It differs from bilinear interpolation by having 16 neighboring pixels instead of 4 in 2D.

Mathematical formulation

Consider a function f and the derivatives f_x, f_y, f_{xy} sampled at four grid points that form a square. Then bi-cubic interpolation consists of finding 16 coefficients α as follows:

$$A = M \begin{bmatrix} f(x_1,y_1) & f(x_1,y_2) & f_y(x_1,y_1) & f_y(x_1,y_2) \\ f(x_2,y_1) & f(x_2,y_2) & f_y(x_2,y_1) & f_y(x_2,y_2) \\ f_x(x_1,y_1) & f_x(x_1,y_2) & f_{xy}(x_1,y_1) & f_{xy}(x_1,y_2) \\ f_x(x_2,y_1) & f_x(x_2,y_2) & f_{xy}(x_2,y_1) & f_{xy}(x_2,y_2) \end{bmatrix} M^T \quad (2)$$

$$= \begin{bmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix} \quad \text{and}$$

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix}$$

Then the interpolated value $p(x,y)$ is:

$$p(x,y) = \begin{bmatrix} 1 & x & x^2 & x^3 \end{bmatrix} A \begin{bmatrix} 1 \\ y \\ y^2 \\ y^3 \end{bmatrix} \quad (3)$$

Algorithm

An algorithm for bi-cubic interpolation can then be described as follows:

1. Identify the four neighboring grid points (pixels in this case) (x_1, y_1), (x_1, y_2), (x_2, y_1), and (x_2, y_2).
2. Calculate the derivative for each corner pixel by finding the value of the slope between the two surrounding points in the appropriate axis x or y, and for xy, take the derivative in both axes, one at a time.
3. Retrieve the corresponding function values in f, f_x, f_y, f_{xy}.
4. Compute α using Equation (2)
5. Compute the interpolated value using Equation (3).

The computational complexity of this algorithm is also up to the complexity of the matrix multiplication used for one position, as it requires a fixed number of arithmetic operations regardless of the grid / texture size, but there are far more operations and matrix loads to do compared to the bilinear interpolation.

For each of those techniques, in order to get RGB values, the computation has to be done three times, once for each channel. They can be done in parallel as a channel value does not depend on another channel.

3 Experimental setup and Performance Evaluation

3.1 Hardware

The hardware used in this study is a NVIDIA RTX A4000 containing 6144 CUDA cores for a single precision performance of 19.2 TFLOPS as well as 192 third-generation tensor cores for a tensor performance of 153.4 TFLOPS (using the sparsity feature) according to the NVIDIA documentation. [NVIDIA, 2022]

3.2 Software

This study uses NSight, which is a software that shows performance and hardware usage for the current kernels on a NVIDIA graphics card, in order to evaluate the performances.

3.3 Test strategies

Textures

As far as this study as reached, four textures have been used in order to demonstrate the correctness of the implemented methods such as the unit square with a different color on each corner, a checkerboard in order to check the borders and sudden changes, a fluid-like pattern with iridescent colors texture and a texture with 4 channels.

Methods

For any of the following, the goal was to find a working implementation that works for our specific needs, validate it works by comparing the results with already supported versions of interpolation or known algorithms and finally compare the performances.

In order to test out different strategies, two or three steps are designed depending on the implementation and what is available on the NVIDIA documentation about textures [NVIDIA, 2025a]. In all cases, two approaches are tested. The first being an interpolation over a texture that is the same size of the current texture to see if anything changes and the second one evaluating completion for up/down-scaling which are used to show a texture that is either smaller or bigger than the actual window size, thus meaning that interpolation is needed to fill missing value or stay consistent within the image.

- For bilinear interpolation, since texture memory supports functions such as tex2D in CUDA, a program that simply puts into memory the original texture and does a simple call to the tex2D function when we want to interpolate was created and will be our checker for correctness of further bilinear interpolations, as tex2D allows for bilinear interpolation done on the texture unit in 2D.

- For bi-cubic interpolation, since it is not supported by the texture unit, a program to interpolate a texture with

”normalized” coordinates was implemented. This is useful in the case of up-scaling or down-scaling. Since we want to compare performances, a similar function for bilinear interpolation is implemented.

For both interpolations, instead of calculating the channels separately, we structured the matrix so that it holds the 3 channels directly so that they are independent.

Finally, we will compare the previous methods with the different proposed implementations of texture interpolation using tensor cores that follows.

4 Tensor Core interpolation limitations and strategies

4.1 Difficulties

A first thing to note when we are using CUDA is that we are working warp-wise, i.e, using 32 packed SIMD threads of a block that typically ranges between 256 and 1024 threads. This means that for a single matrix multiply, 32 threads are used and are waiting on loading and storing matrices to/from the tensor core. A second limitation are the precisions available on any WMMA (Warp Matrix Multiply and Accumulate) API call. A typical use of this API is to call the function load_matrix_sync() to load the matrix in the tensor core, then do the multiplication and addition by using mma_sync() and finally store the result with store_matrix_sync(). The limitation about precision is the fixed set of available matrix size available depending on the precisions that have been chosen for the input and output matrices [NVIDIA, 2025a] for which they are bigger than our needs and needs to be padded with 0. In this study, we will focus on half- and double-precision input matrices, as well as half-, single- and double-precision output matrices. Another difficulty mentioned in the CUDA programming guide is that the mapping of matrix elements to each thread in the warp is unspecified and subject to changes between architectures. Finally, the WMMA API expects inputs that are only in either global or shared memory, so that one cannot define a matrix in local memory.

4.2 Padding with 0

In order to prepare the experiment with tensor cores, a simple baseline CUDA code was created dealing with only 1 pixel per warp and using the previously defined matrices described in Section 2. This is done by filling the rest of the matrices with zeroes, knowing that matrices of half inputs and half or single output are of size 16×16×16 and for double-precision inputs and outputs it is 8×8×4. Using double-precision, there is no need to modify the matrix when using a 1024×1024 pixels output as it still allows for a precise output but there is a need to change the matrices for half-precision.

Mathematical reformulation

For bilinear interpolation, we need to transform the Equation (1) into the following normalized form :

$$f(x,y) \approx [f_{11} \quad f_{12} \quad f_{21} \quad f_{22}] \begin{bmatrix} 1 & -1 & -1 & 1 \\ 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} M^T \tag{4}$$

where $M = [1 \quad a \quad b \quad ab]$ and x’ = (x + 0.5) / size of the output window × size of the texture, y’ = (y + 0.5) / size of the output window × size of the texture, $a = x' - \lfloor x' \rfloor$, $b = y' - \lfloor y' \rfloor$

This switches to a unit square on the local 4 neighboring texels of the current pixel in a normalized form in order to allow half-precision range.

For bi-cubic interpolation, we need to transform the Equation (2) into the following normalized form :

The calculation for the A matrix is unchanged, but as for bilinear interpolation we modify the Equation (3) such that x is replaced by a and y is replaced by b on the right hand side of the equation.

Again, this works because we are locally computing the interpolation using the 16 neighboring texels, if it was not neighboring then the pixels would be outside the locally created unit square and would then give results that are bigger than 255.

Algorithm

Both algorithms are unchanged up to the fourth step where we need to load the matrices, call the WMMA kernel containing multiple WMMA API calls to do the matrix multiplication, and retrieve the results. The difference is that the matrix is filled with 3 channels R, G, B, so that for bi-cubic for example, the M and M^T matrices are repeated 3 times inside the 16x16x16 matrix as well as there are 3 different f matrices inside the middle matrix. A WMMA kernel is as follows:

```
__device__ void wmmaNaiveKernel(const half *__restrict__ A,
                                const half *__restrict__ B,
                                half *C) {
    wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N,
        WMMA_K, half, wmma::row_major> A_frag;
    wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N,
      WMMA_K, half, wmma::row_major> B_frag;
    wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N,
      WMMA_K, half> C_frag;

    wmma::load_matrix_sync(A_frag, A, WMMA_K);
    wmma::load_matrix_sync(B_frag, B, WMMA_N);
    wmma::fill_fragment(C_frag, 0.0f);
    wmma::mma_sync(C_frag, A_frag, B_frag, C_frag);
    wmma::store_matrix_sync(C, C_frag, WMMA_N,
                            wmma::mem_row_major);
}
```

Figure 1: *Naive WMMA function implemented with CUDA half precision fragments*

Where WMMA_M, WMMA_N, and WMMA_K are all equal to 16 because this is using half-precision in this case.

4.3 Making the matrix denser

Using the previous baseline implementation, one can see that the matrix containing the pixel color values is very sparse

because of padding and can be easily improved by doing 4 pixels at once. In the bicubic case, the matrix was filled such that each group of four columns of the matrix corresponded to the $f(x, y)$s and their derivatives of each channel while adding zeroes at the last 4 rows of 1 pixel. That way, since it is a 16×16 matrix, we have 16/4 = 4 pixels that can fit in only 1 matrix that is now filled with 192 potentially non-zero elements and 64 zeroes. Those 4 pixels are always picked such that they form a square of 2x2 pixels allowing for a potential faster memory access. Of course, the other matrices had to be changed accordingly, notably duplicating the M and M^T matrices along the diagonals of the overall matrices. This was only done for the bi-cubic interpolation and for 4 pixels implementation.

4.4 Four channels rather than 3

Sometimes, textures have an arbitrary amount of channels such as a fourth channel that can be added to describe the transparency of the current pixel called Alpha (RGBA). This is also something to compare with to see if adding channels is less expensive with the tensor core implementation compared to adding channels in a CUDA classic approach or even using the texture unit, which allows for 4 channel bilinear interpolation. In order to add such a channel for the bilinear interpolation, the matrix is simply added the 4 new data points right below the other 3 "vectors" of 4 data points in the matrix as we already had 3 channels. For the bi-cubic part, setting the values of the fourth channel instead of zeroes at the last 4 rows of each group of 4 columns is enough in the 4 pixels per warp case described before and now fills up the whole matrix.

4.5 Taking advantage of half precision intrinsics

In CUDA, using the FP16 (half-precision) type, the + or * operators are not recognized whereas it is in float type (single-precision). A first implementation was then to let the compiler do the conversion from float to half by itself after doing the operations in float, then a second implementation was to use the __float2half() intrinsics (functions) in order to explicitly "cast" any float after their operations into a half. A last implementation was to use the provided add and multiply functions available in the FP16 library such as __hmul so that we can directly work with half values without the need for a transformation from float to half[NVIDIA, 2025b]. Such a change was made on all the previous implementations but the 4 channel implementation.

4.6 Changing the block sizes

As previously said, sets of 32 threads, called warps, lie in blocks which sizes usually ranges from 256 to 1024 threads. All the previous implementations used 8 warps per block, hence using 256 threads. Performance could be improved by varying the block size depending on the workload, memory constraints, and other factors and can only be checked through experimentation [Ikram *et al.*, 2022]. Changing the block size can be beneficial in terms of resource utilization, improved occupancy and better load balancing but can complicate tuning and not have the same result on another GPU. In the current context, it is not sufficient to only change the block size but some of the memory layouts were also changed along with the size of the block, as our implementation uses shared memory which is shared across a block rather than a warp, which is directly affected by the block size. Multiple block sizes are tested to identify the ideal block size that is then used in the following implementation.

4.7 "Keeping" the results in between tensor calls

As seen in Figure 1, the WMMA kernel relies on API calls that are synchronous across the warp, meaning that all 32 threads have to synchronize on loading the first matrix then the second, etc. This is costly when consecutive calls of this function are made as the kernel waits on loading and storing for most of the time and implies a memory bottleneck.

A way to reduce the amount of loads and stores would be to be able, for consecutive matrix multiplications, to keep the result matrix in the tensor core or the layout of it in each thread to get rid of a store operation, whose only purpose is to put the whole matrix in shared memory as well as a load.

Even though the part of the matrix that each thread in the warp contains is unknown and is subject to change, we can only try to assume that the layout between a row major matrix result is the same as the one for a row major matrix_a input matrix.

Such an implementation does not require one to know the layout of the matrix but only to correctly copy from one call to another, within the same thread of the current warp, the current matrix elements it holds from the result.

The change made to Figure 1 was to move the accumulator out of the function, pass it as a pointer before the call to the function and remove the store operation in order to only have the result in the accumulator. Then, a second function was needed to have the input as the previously saved accumulator. Unfortunately, the WMMA API does not provide a way to transform a wmma::accumulator matrix to a wmma::matrix_a matrix so that the following workaround was found by checking the members available in the code editor for each of those elements.

```
__device__ void wmmaNaiveKernelCol(
    wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K,
    half> *A, const half *__restrict__ B, half * C) {

    wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K,
                    half, wmma::row_major> A_frag;
    wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K,
                    half, wmma::col_major> B_frag;
    wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K,
                    half> C_frag;
    wmma::load_matrix_sync(B_frag, B, WMMA_N);

    #pragma unroll
    for (int i = 0; i < A_frag.num_elements; i++) {
        A_frag.x[i] = (*A).x[i];
    }
    wmma::fill_fragment(C_frag, 0.0f);

    wmma::mma_sync(C_frag, A_frag, B_frag, C_frag);
    wmma::store_matrix_sync(C, C_frag, WMMA_N,
                            wmma::mem_row_major);
}
```

Figure 2: *Second WMMA function implemented for allowing results immediate usage*

This uses the previously made assumption that the mapping

of thread elements are the same between calls and between an accumulator and an input matrix a.

5 Results

For every test conducted in this study, the results were compared to the already provided texture interpolation function from NVIDIA in the bilinear case and to the naive full CUDA implementation for bi-cubic interpolation. The following images, set as textures, were used and all outputs are of size 1024x1024 pixels.

Each of those textures allows one to check whether there is a loss in needed sharpness in the checkerboard, if the colors and curves are respected in Figure 3b and if transparency is respected when using 4 channels in Figure 3c.

All tensor core based implementations correctly interpolated each texture. Visual inspection and pixel comparison shows that they are correct up to precision differences when using half-precision that make up a slight error on usually 1 channel by 1 value on some pixels. For example, a pixel which value would be (173,56,89) after being interpolated might end up being (173,57,89) for which the error cannot be seen by a human eye. Such error does not appear when using double-precision. The point of this study is mainly about performances which are on the following sections beside validating the tensor core implementations.

In all of the figures 4, 5, 6, 7, A corresponds to Bilinear CUDA naive 3 channels, B to Texture unit 3 channels, C to texture unit 4 channels, D to Bi-cubic CUDA naive 3 channels and E to Bi-cubic CUDA 4 channels. Everything that does not specify CUDA or texture unit is using tensor cores.

In Figure 4, a corresponds to Simple pad half-precision, b to Simple pad double-precision and c to 4 channels.

In Figure 5, a corresponds to Bi-cubic Simple pad half-precision, b to Dense matrix half-precision and c to Bi-cubic 4 channels.

In Figure 6, a corresponds to Double block size, b to Half intrinsics and c to Half intrinsic and keeping results.

In Figure 7, a corresponds to Double block size, b to Double block size with denser matrix, c to Half intrinsics, d to Half intrinsic, denser matrix and double block size, e to Half intrinsic and keeping results and f to Denser matrix, half intrinsic, keeping results and double block size

5.1 Modifying the core algorithm performances

Figure 4 and 5 present the average time spent in the kernel from before the interpolation up to after interpolating the whole textures using different approaches. It appears that using the double-precision version is significantly longer than when the half-precision version is used in the same domain. In any case, at this stage, both the texture and CUDA naive implementations seems faster than the tensor core versions. Adding a channel is more expensive when using the tensor cores approach than in the texture unit in the bilinear case and it is way more expensive in the bi-cubic case compared to its naive CUDA implementation.

5.2 Tensor code modifications performances

The results in Figure 6 and 7 present enhanced tensor cores strategies. It shows that taking advantage of half-precision intrinsics, along with modifying the block sizes and keeping the results between WMMA API calls seems to improve performances in some cases but it appears that, in any way, it is faster than the results in Figure 4 and 5 for both bilinear and bi-cubic interpolation. It appears that the best version achieved in this study for bi-cubic interpolation is 4.34x slower than a naive CUDA interpolation.

6 Discussion

Although tensor cores provide extremely efficient matrix operations, actual performance depends heavily on the warp-level synchronization as loading and storing is what takes most of the time along with the computation of parameters to be put in the matrices. Another performance might also be due to an issue with memory bandwidth as the tensor core API require matrices to be put in either shared or global memory which can significantly create bandwidth pressure.

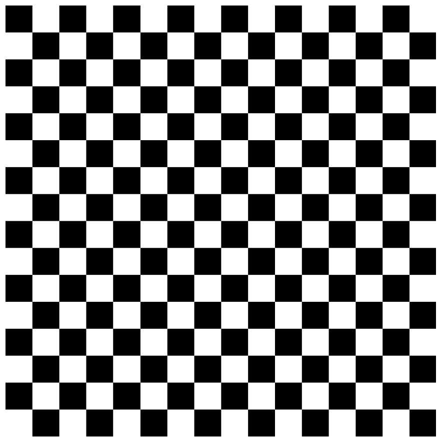
An observation that was made is that without waiting for the load, and thus having wrong results in this study, it ”improved” the performances by a matter of 3-4ms in the last implementation of the bi-cubic interpolation, which would be almost equivalent to the naive CUDA counterpart.

This is something that could be later studied, as it is possible to copy data directly as this study does in the case of keeping the results between tensor calls without explicitly knowing the matrix layout in each thread. This might also be one of the reasons why it is faster in the case of keeping the results between calls. The lack of a native, efficient way to pass accumulator fragments to input fragments might be a factor why the performance on keeping the result matrix between calls is not great enough to make a difference even though synchronization and memory bottleneck are avoided.

As expected, the first implementation have many too sparse matrices computing only 1 pixel compared to 4 in the last implementations. The tensor cores sparsity feature developed by NVIDIA could be a way to further optimize performances in such implementations.

Performance improves only slightly when using the packing of multiple pixels per warp and such cause might be that calculating the parameters for 3 more pixels would almost cancel out the speedup achieved by multiplying an almost full matrix with tensor cores along with loading such parameters.

Some other limitation to note is that WMMA internal thread-to-matrix mapping is undocumented and architecture dependent. An example found in this study is the undocumented behavior of calling the WMMA API with double-precisions matrices with GA104 chips (or other possible chips) used in this study, rather than GA100, which supposedly supports it, that leads to a larger compute time than it should have been. This result is explained by looking on NSight as it shows that it did not use tensor cores, thus meaning that the API is capable of doing a matrix multiply without tensor cores or when the current GPU does not support tensor usage with such precision without warning the user.



(a) *3 channel 1024×1024 checkerboard*

(b) *3 channels 256×256 fluid-like texture*

(c) *4 channels 1500×1102 splash of paint*

Figure 3: *Textures used for testing*

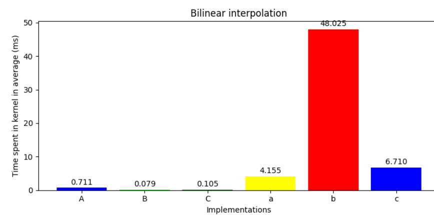


Figure 4: *Average time spent in kernel for some bilinear implementations*

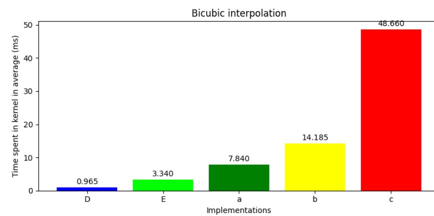


Figure 5: *Average time spent in kernel for some bi-cubic implementations*

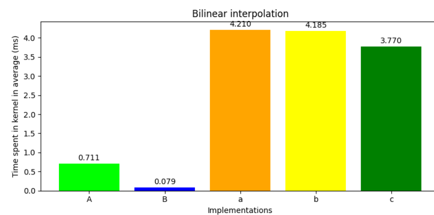


Figure 6: *Average time spent in kernel for some tensor-core modified bilinear implementations*

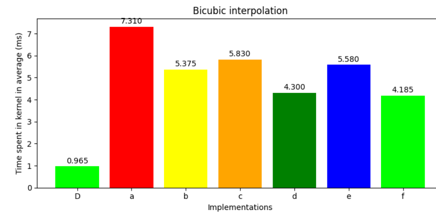


Figure 7: *Average time spent in kernel for some tensor-core modified bi-cubic implementations*

Overall, this study is to investigate whether NVIDIA Tensor Cores, typically used for deep learning workloads with bigger matrices, could be repurposed to accelerate texture interpolations on top of putting work into a hardware part that is mostly inactive. Our preliminary results indicate that tensor cores, in their current form and usage in this study, do not offer the same or better performance than the already existing implementation and the use of texture units. The primary bottleneck lies in the rigid programming model of WMMA that can be further investigated, especially for synchronization and load and store of matrices within each thread of a warp. Another bottleneck previously explained is memory bandwidth but is a similar issue. Other untapped potential is by using more complex or higher-order interpolations with more matrix multiplications such as 3D textures when needed by some applications for a better quality and where performance is yet to be improved. The improvements made within this study suggest that performance can be substantially enhanced with a better comprehension of the hardware and software abstractions, which this study unveiled some of these aspects. Tensor cores are, thus, promising candidates for future optimizations. More research is needed to explore alternative memory layouts, cooperative kernel designs, possible architecture workarounds or even using mixed-precision such as TF32 that is specific to tensor cores to see if a speedup can be gained using such precision that is available on tensor cores.

References

[Briand and Davy, 2019] Thibaud Briand and Axel Davy. Optimization of Image B-spline Interpolation for GPU Architectures. *Image Processing On Line*, 9:183–204, August 2019.

[Burgess, 2020] John Burgess. RTX on—The NVIDIA Turing GPU. *IEEE Micro*, 40(2):36–44, March 2020.

[Gonzalez and Woods, 2002] Rafael C. Gonzalez and Richard E. Woods. *Digital image processing*. Prentice-Hall, Upper Saddle River, NJ, 2. ed., internat. ed edition, 2002.

[Ikram *et al.*, 2022] Muhammad Jawad Ikram, Mostafa Elsayed Saleh, Muhammad Abdulhamid Al-Hashimi, and Osama Ahmed Abulnaja. Investigating the effect of varying block size on power and energy consumption of GPU kernels. *The Journal of Supercomputing*, 78(13):14919–14939, September 2022.

[NVIDIA, 2022] NVIDIA. NVIDIA RTX A4000 Datasheet, 2022.

[NVIDIA, 2025a] NVIDIA. 1. Introduction — CUDA C++ Programming Guide https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=precision%25203A%25203Atf32wmma-description, 2025.

[NVIDIA, 2025b] NVIDIA. 4.2. Half Arithmetic Functions — CUDA Math API Reference Manual 12.9 documentation https://docs.nvidia.com/cuda/cuda-math-api/cuda_math_api/group__CUDA__MATH____HALF__ARITHMETIC.html, 2025.

[Raihan *et al.*, 2019] Md Aamir Raihan, Negar Goli, and Tor M. Aamodt. Modeling Deep Learning Accelerator Enabled GPUs. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 79–92, March 2019.

[Yan *et al.*, 2020] Da Yan, Wei Wang, and Xiaowen Chu. Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 634–643, May 2020. ISSN: 1530-2075.