Intelligent Architectures 5LIL0

Lab 2: Mapping and Accelerating a Convolutional Neural Network on a GPU

Eindhoven University of Technology
Version 2022.0

Contents

1 Introduction

Convolutional neural networks (CNNs) are one of the most popular algorithms used to perform classification tasks. They are composed of an input layer, an output layer, and many hidden layers in between. The total number of layers, the way they process data and the interconnection between them dictate the complexity of the network. When working with complex network architectures, GPUs can significantly accelerate the processing time needed to perform classification tasks. CNNs can be applied in a wide range of applications including classification of images, text, sound, and video. Moreover, they can be used in real-time applications, such as face detection and autonomous driving.

The goals of this assignment are:

1. To learn how to map a CNN to a Nvidia GPU using CUDA.
2. To learn how to implement convolutional layers on GPUs using the im2col transformation and to learn the trade-offs of the transformation.
3. To explore various optimization techniques that can be used to accelerate the execution time of a given application.

CUDA is an API that allows applications to use GPUs for general purpose processing. Furthermore, CUDA is designed to work with programming languages such as C to make it easier for specialists in parallel programming to accelerate their applications using a GPU. Since the operations in a CNN are massively parallel, they are well suited for acceleration using GPUs.

In this assignment, you will optimize a single layer from a pretrained CNN that performs image classification on a Nvidia GPU. In order to map a convolution layer to a GPU, the convolution is converted into a matrix-multiplication, using the im2col transformation (Section 3). The mandatory optimizations in this assignments include code transformations and quantization, but other optimizations can be done for bonus points.

This assignment is organized as follows, Section 2 provides a quick getting started. Section 3 introduces the im2col transformation. Having converted a convolution into a matrix-multiplication (MM), Section 4 provides several optimization strategies to accelerate MMs on a GPU. Section 5 is dedicated to mapping MMs on Tensor Cores. Section 6 presents details about the report that should be submitted. Finally, Section 7 outlines optional bonus exercises.

1.1 Application

The application you will be using for this assignment is a single convolutional layer (Fig. 1) with dimensions typically found in famous image classification networks, such as AlexNet, VGG, ResNet. These networks can classify images into one or more classes. Typically, these networks are trained on more than a million images from the ImageNet database. The input of the network is an image of 224-by-224 pixels. The output is a 1000-dimensional vector that contains the probabilities that each object class has of being present in the input image.

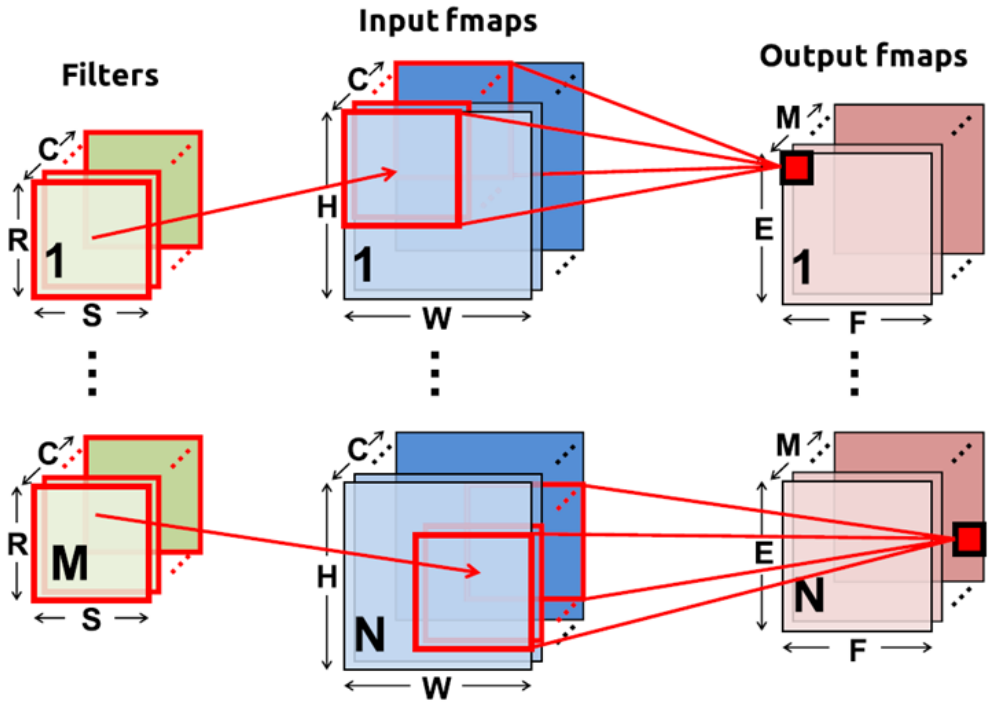


Figure 1: Example of a convolution layer. The convolution window shifts over the input image and thereby computes output pixels. Image from here.

As an example, suppose you feed an image of a panda holding an apple as input to the network. The output should be a 1000-dimensional vector with very low probabilities for every element except for those that correspond to a panda and an apple (which of course should be high).

1.2 Profiling CNNs

To explain why this assignment is focused on the optimization of a single convolutional layer, rather than a complete network, we present an analysis[1] of the time distribution over different layers in a network and over different computation architectures: GPU and CPU.

Fig. 2 shows where the time is going for a typical CNN doing image recognition. In this specific case, AlexNet. It can be observed that both computation architectures spend most of the computation time on convolution and fully-connected layers. Moreover, the GPU is more capable of handling layers such as pooling and normalization.

During the training of a neural network, it is usually the case that computation is carried out in minibatches. The stochastic gradient descent algorithm has been shown to work well with minibatches. During the inference of a neural network, it depends on the application whether single inputs or minibatches are used.

The size of a minibatch affects the overall computation time on GPUs. Fig. 3 illustrates the effect and shows the absolute time spent by each layer per image.

[1]The analysis is based on this thesis.

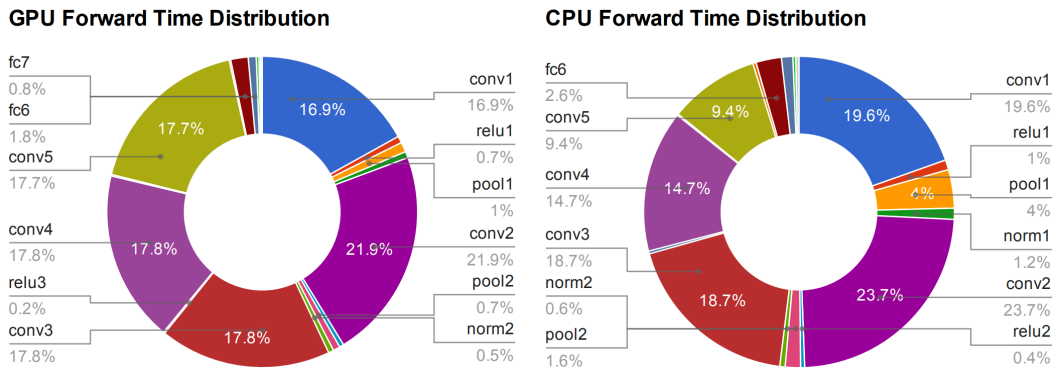


Figure 2: Computation time distribution of individual layers in AlexNet on both GPU and CPU, using a batch size of 256. Image from here.

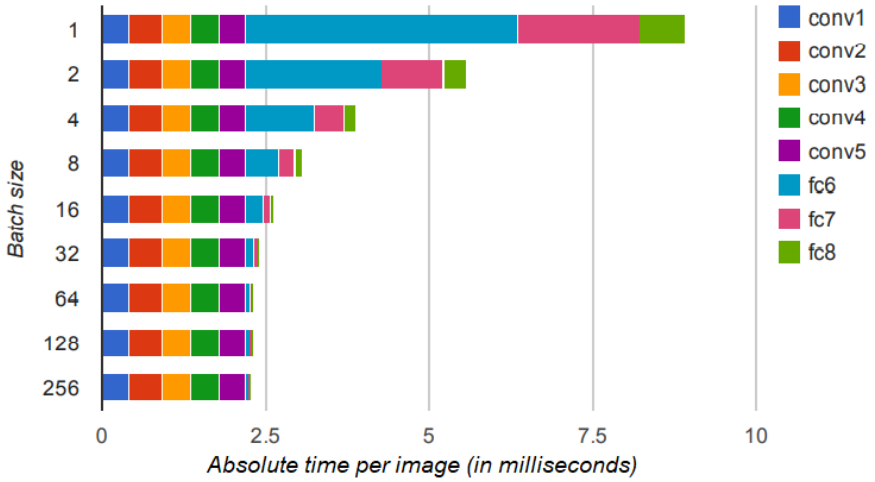


Figure 3: Absolute computation time spent on each layer per image, when the batch size varies from 1 to 256. Image from here.

Exercise 1.1: Batch size influence

Explain:

- Why the absolute computation time per image of the fully-connected layers in Fig. 3 decreases when the batch size increases.
- Why the absolute computation time per image of the convolution layers in Fig. 3 remains constant when the batch size increases.

1.3 GPU architecture

The GPU you will be using in this assignment is the Nvidia GeForce RTX 2080 Ti. This GPU is based on the Turing microarchitecture and is manufactured on TSMC's 12nm FinFET process. Most notable features are ray tracing cores, tensor cores and the ability of concurrent execution of integer and floating point operations.

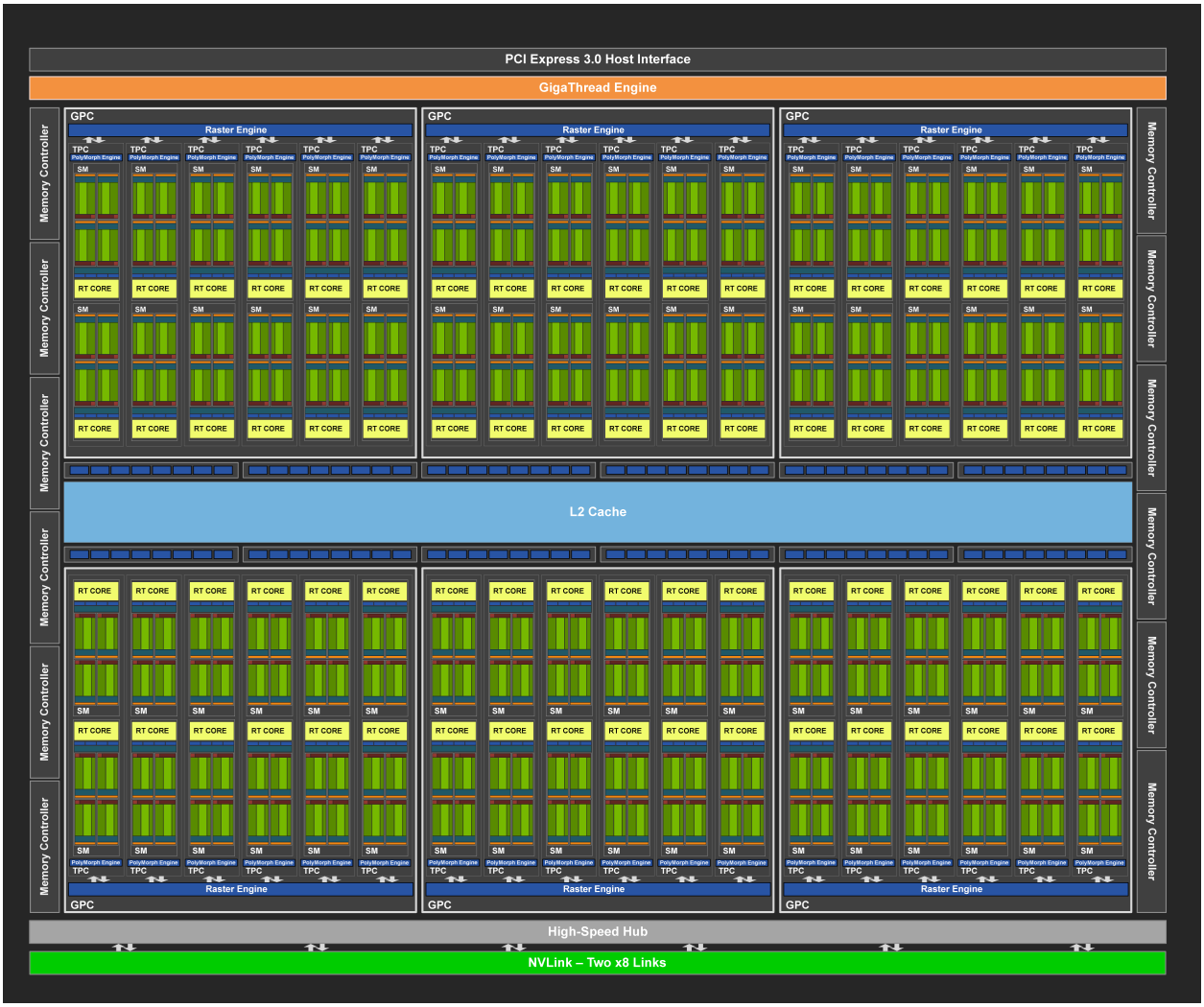


Figure 4: Turing GPU architecture overview. Image from here.

The most performant GPU in the RTX 20 series includes 72 Streaming Multiprocessors (SMs). See Fig. 4 for an illustration of the microarchitecture.

Each SM (Fig. 5) contains 64 CUDA Cores, 8 Tensor Cores, a 256 KB register file, 4 texture units, and 96 KB of L1/shared memory which can be configured for various capacities depending on the compute or graphics workloads. Ray tracing (RT) acceleration is performed by a RT processing engine within each SM.

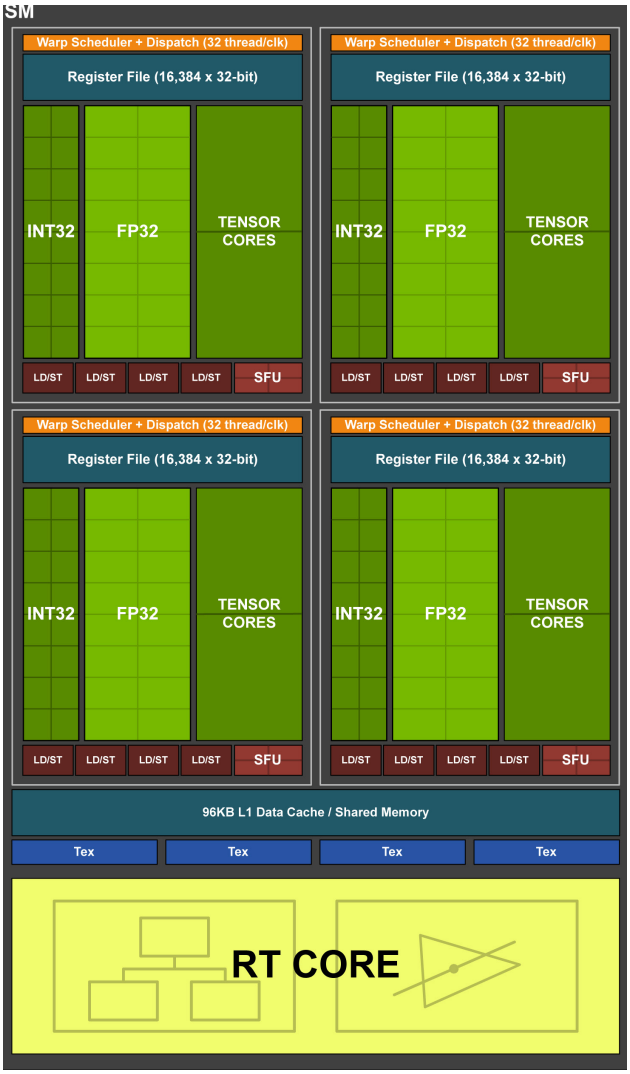


Figure 5: Turing SM architecture overview. Image from here.

The full implementation of the most performant in the RTX 20 series includes the following:

- 4,608 CUDA Cores
- 72 RT Cores
- 576 Tensor Cores
- 288 Texture units
- 12 32-bit GDDR6 memory controllers
- 6,144 kB of L2 cache.

2 Getting started

For this assignment, you will be working on the university server –HPC cluster– on which the tool-flow is already pre-installed. You will be connected to a specific node with 7 NVIDIA GPUs, namely GeForce RTX 2080 Ti (you will use only one GPU for this assignment). Follow this link to get more information about the cluster.

2.1 Accessing GPU cluster

To access the servers, accounts have been made for you on the server. Your username and password are your student number (e.g. 20177347) and your email password. In order to connect, you will need an ssh client. On Linux and OS-X you already have such a client (the ssh command on the terminal). On Windows, nevertheless, you need to download and install something like MobaXterm. This will allow you to connect to the server, edit files, and run commands. **Note** that if you are off-campus you need to enable TU/e VPN in order to be able to access the server.

```
$ ssh username@hpc.tue.nl
```

2.2 Running an application on the cluster

Now that you are connected to the server, you first need to download the required files. To do so, you should make an account on TU/e gitlab. Then, you should clone the files on your account:

```
$ git clone https://gitlab.tue.nl/20177018/IA-GPU
```

You will be asked for a username and a password. Use the username and the password of the account you just made.

The first step is to check if you have access to the GPU cluster:

```
$ sinfo
```

You should see a `PARTITION` (`5LIL0.elec.q`) with GPU node (`elec-gpuA001`).

The next step is to test a CPU implementation of the convolutional layer we have provided. By default, the application runs on the server's CPU only. Later on, we will see how to map its computations to GPU. Use the following script to: (i) build the application's pipeline, (ii) execute it, (iii) measure the execution time, and (iv) validate the result using the provided reference.

```
$ cd IA-GPU/LabAssignments
$ sbatch run_prog.run
```

To check the status of your job you can execute the following command.

```
$ squeue
```

When your job is finished you should be able to see two files in the directory, i.e. `slurm-<JOB ID>.out` and `slurm-<JOBID>.err`. If you open the `*.out` file you should see a message along the lines of "average time of CPU baseline" at the end of the file, confirming the correctness of your implementation result with respect to the reference. In later stages, after you have done modifications, if your program generates an error you can see that in the `*.err` file.

2.3 CUDA tutorial

For someone who is unfamiliar with CUDA and wants to learn it quickly we suggest reading the following resources. It can also be used by those who already know CUDA and simply want to brush-up on the concepts.

- CUDA C Programming Guide

3 Implementing Convolutions using im2col

A lot of research is going on into the acceleration of CNNs. Nvidia, for example, has created deep-learning libraries that contain many highly-optimized implementations of most operations that happen inside a CNN. But what exactly is the black magic that these libraries use? What exactly does one do to "optimize" or accelerate CNNs?

In this assignment, we aim to cover how a single convolution layer is implemented in deep-learning libraries and what it takes to achieve high performance. The convolution operation is particularly representative of high-performance implementations. It encompasses algorithmic cleverness, careful tuning and proper exploitation of the low-level hardware architecture.

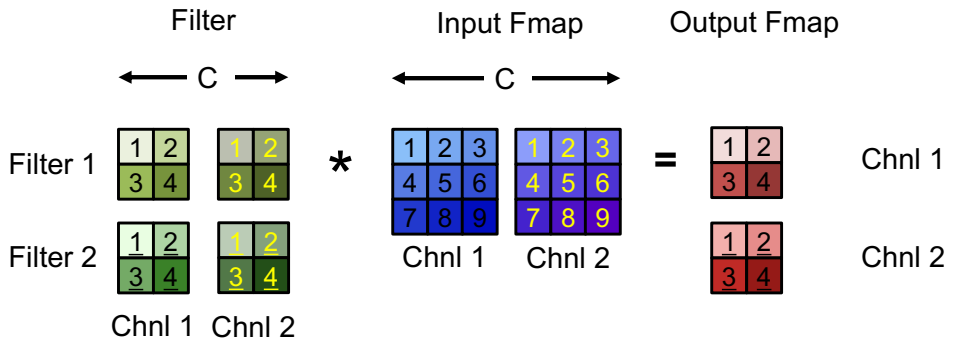


Figure 6: Example of convolution with 2 input and output channels. The window size is 2-by-2. Image from here.

As a recap, a convolution process is shown in Fig. 6. When you implement this in software, you get a 7-layer deep loopnest. Applying code transformations to optimize this problem is difficult and hampers the programmer from achieving high performance. Fortunately, we can transform the convolution operation into a problem that's easier to solve: matrix multiplication.

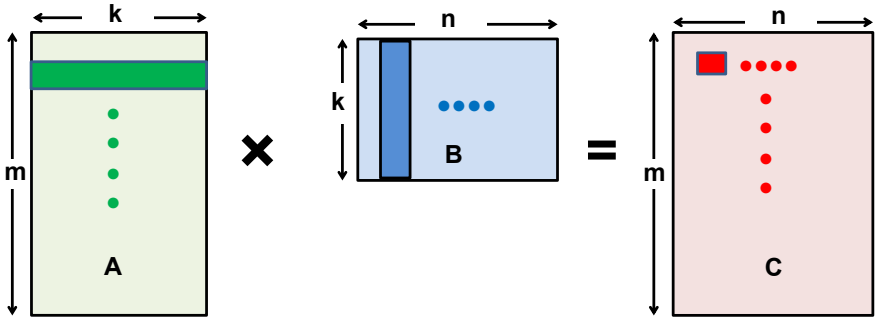


Figure 7: Example of GEMM ($A * B = C$). Image adapted from here.

An example of Matrix multiplication or **GE**neralized **M**atrix **M**ultiplication (**GEMM**) is shown in Fig. 7. GEMM is widely used to accelerate CNNs as it can be used to for both fully-connected and convolution layers. However, how can you transform the 7-dimensional problem of Fig. 6 into a 3-dimensional GEMM problem (Fig. 7)?

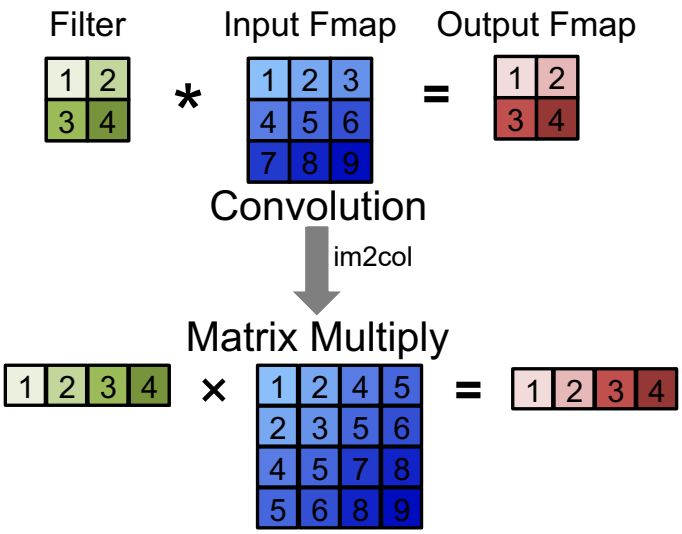


Figure 8: Example of a convolution implemented using GEMM via the im2col transformation. Image from here.

The transformation that allows converting a convolution into GEMM is called im2col (image to column) and shown in Fig. 8. For a 4D filter (weight), it comes down to flattening a single filter into a 1D array. Transforming the input is however not so straightforward. The careful reader might have noticed in Fig. 8 that some of the input data appears multiple times. This happens because pixels are included again and again due to overlapping filter sites.

Exercise 3.1: Visualizing im2col memory overhead

Produce a plot that shows the additional storage (in kB) required for the im2col transformation for varying filter sizes. Vary R and S (as in Fig. 1) from 1 to 11 with step size 2. Assume a convolutional layer that has dimension M=N=64, H=W=32.

Note: For getting memory overhead, you do not need to run program, just do some math calculation.

Exercise 3.2: Visualizing im2col runtime overhead

Produce a plot that illustrates the runtime of the im2col transformation. Vary R and S (as in Fig. 1) from 1 to 11 with step size 2. Assume a convolutional layer that has dimension M=N=64, H=W=32.

Note: For visualizing run-time overhead, you need to use some timing functions, as a reference. You can study the codes in `IA-GPU/LabAssignments/helper.h` (the function `measure_CPU_runtime()`). Im2col code is provided in `IA-GPU/Im2col/im2col.h`.

Now, you might be shocked after quantifying the overhead in terms of memory and runtime after having done the im2col transformation. However, you will notice that this wastage is outweighed by its simplified optimization advantages.

4 Accelerating Matrix-Multiplication

This section will show several techniques to optimize matrix multiplication on GPU. Most of them are generic, as they can be applied to many other applications. These techniques are:

- Tiling
- Memory coalescing
- Avoiding memory bank conflicts
- Quantization

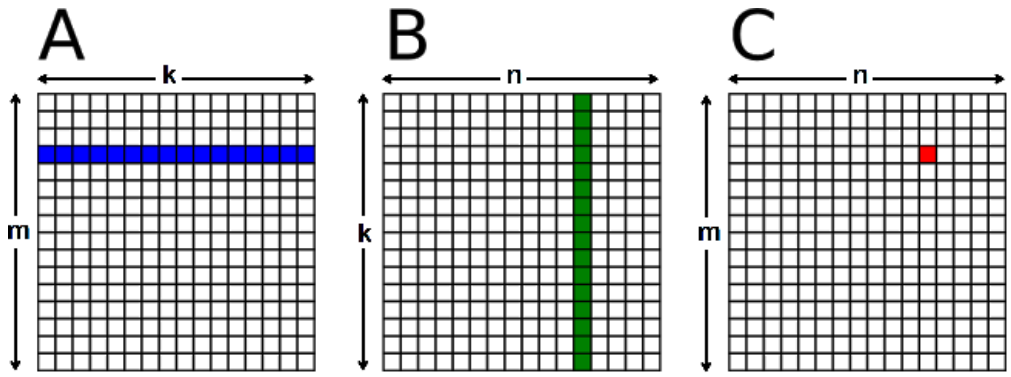


Figure 9: Matrix multiplication. Computation of a single element $C_{3,11}$.

To simplify the explanation of these optimizations, square matrices ($m = n = k$) are used in the illustrations. Fig. 9 shows the memory footprint to compute an element $C_{3,11}$ (in red). This can be viewed as the inner product of one row of A (in blue) and one column of B (in green).

4.1 Naive GPU implementation

```
/* Pseudocode of naive kernel running on GPU */
__global__ void matrixMul(A_gpu, B_gpu, C_gpu, K){
    i <= blockIdx.y * blockDim.y + threadIdx.y    // Row i of matrix C
    j <= blockIdx.x * blockDim.x + threadIdx.x    // Column j of matrix C

    for k = 0 to K-1 do
        accu <= accu + A_gpu(i,k) * B_gpu(k,j)
    end

    C_gpu(i,j) <= accu
}
```

A naive implementation on GPUs assigns one thread to compute one element of matrix C. Each thread loads one row of matrix A and one column of matrix B from global memory, do the inner product, and store the result back to matrix C in the global memory.

Exercise 4.1: Establishing GPU baseline

Implement the naive implementation in `IA-GPU/LabAssignments/baselineCUDAcore.h`, assume $m = n = k = 4096$. A template function has been setup for you already (`simple_fpu_gemm`). Note that the template function is more extensive than the pseudocode examples in this section. This is done to keep your code aligned throughout the assignment.

For the basic CUDA GEMM implementation report the runtime of:

- Memory copy from host to device
- Execution time on GPU
- Total runtime of the CNN layer

Reflect on the CPU vs GPU speedup, if any.

Note: Un-comment the corresponding part in `IA-GPU/LabAssignments/main.cu`

In the naive implementation, the amount of computation is $2 \cdot M \cdot N \cdot K$ FLOPs, while the amount of global memory access is $2 \cdot M \cdot N \cdot K$ words. The "computation-to-memory ratio" is approximately $\frac{1}{4}$ FLOP/byte. Therefore, the naive implementation is bandwidth bounded.

4.2 Increasing "Computation-to-Memory Ratio" by Tiling

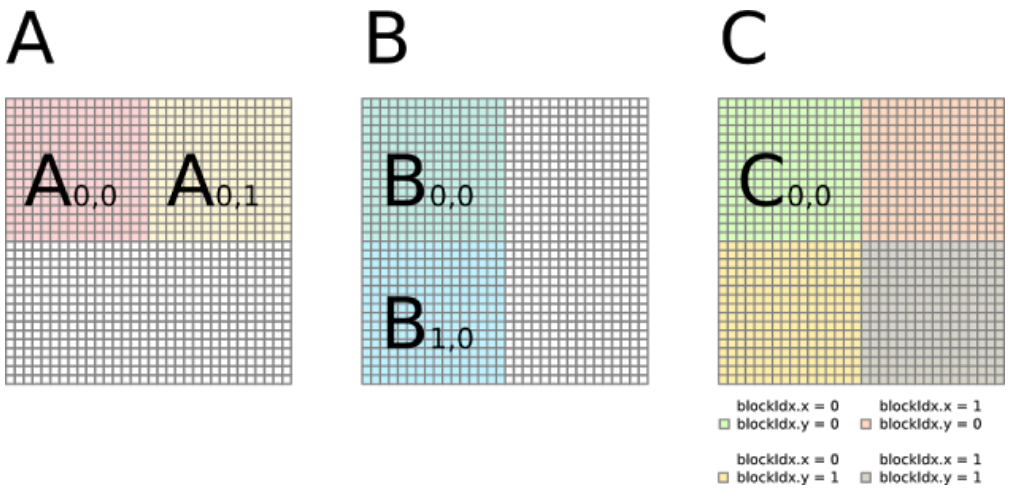


Figure 10: Example of tiling on GPU with four thread blocks.

To increase the "computation-to-memory ratio", the tiled matrix multiplication can be applied. One thread block computes one tile of matrix C. One thread in the thread block computes one element of the tile. Fig. 10 shows a $32 \cdot 32$ matrix divided into four $16 \cdot 16$ tiles. To compute this, four thread blocks each with $16 \cdot 16$ threads can be created.

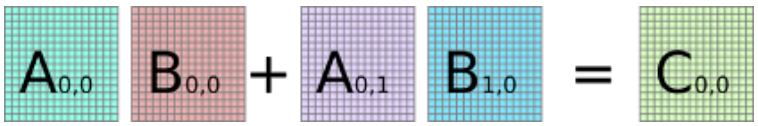
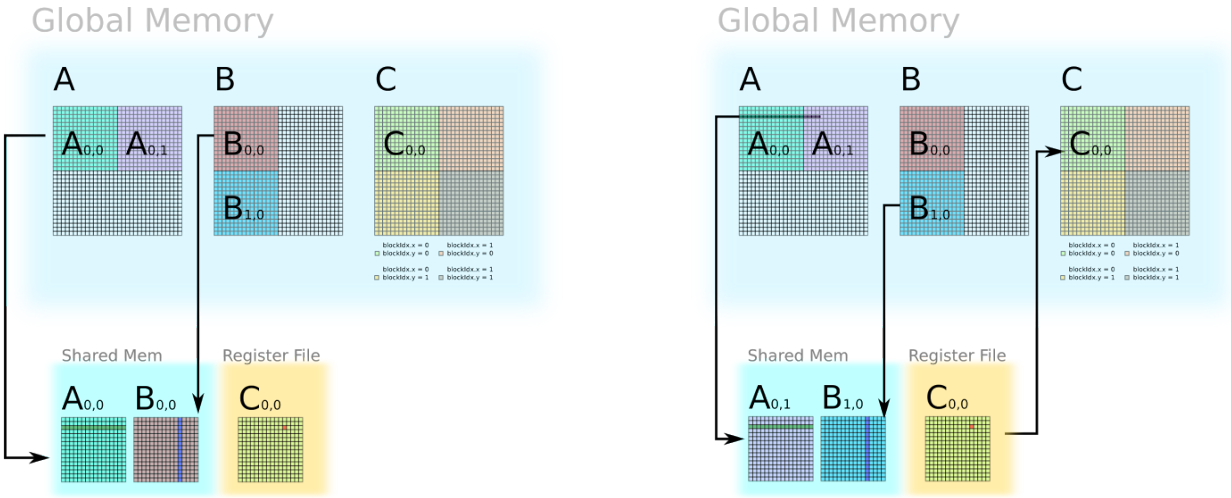


Figure 11: One thread block may compute $C_{0,0}$ in two iterations.

The GPU kernel computes C in multiple iterations. In each iteration, one thread block loads one tile of A and one tile of B from global memory to shared memory, performs computation, and stores temporal result of C in register. After all the iteration is done, the thread block stores one tile of C into global memory. For example, a thread block can compute $C_{0,0}$ in two iterations: $C_{0,0} = A_{0,0}B_{0,0} + A_{0,1}B_{1,0}$. This is exemplified in Fig. 11



(a) Tiling iteration 1.

(b) Tiling iteration 2.

Figure 12: Visualization of two tiling iterations.

In the first iteration, shown in Fig. 12a, the thread block loads tile $A_{0,0}$ and tile $B_{0,0}$ from global memory into shared memory. Each thread performs inner product to produce one element of C. This element of C is stored in the register, which will be accumulated in the next iteration. In the second iteration, shown in Fig. 12b, the thread block loads tile $A_{0,1}$ and tile $B_{1,0}$ from global memory into shared memory. Each thread performs the inner product to produce one element of C, which is accumulated with previous value. If this is the final iteration, then the element of C in the register file will be stored back into global memory.

```
/* Pseudocode of kernel running on GPU */
__global__ void matrixMul(A_gpu,B_gpu,C_gpu,K){
    __shared__ float A_tile(blockDim.y, blockDim.x)
    __shared__ float B_tile(blockDim.x, blockDim.y)

    accu <= 0

    /* Accumulate C tile by tile. */
    for tileIdx = 0 to (K/blockDim.x - 1) do
        /* Load one tile of A and one tile of B into shared mem */
```

```
        // Row i of matrix A
        i <= blockIdx.y * blockDim.y + threadIdx.y
        // Column j of matrix A
        j <= tileIdx * blockDim.x + threadIdx.x
        // Load A(i,j) to shared mem
        A_tile(threadIdx.y, threadIdx.x) <= A_gpu(i,j)
        // Load B(j,i) to shared mem
        B_tile(threadIdx.x, threadIdx.y) <= B_gpu(j,i) // Global Mem Not
   coalesced
        // Synchronize before computation
        __sync()

        /* Accumulate one tile of C from tiles of A and B in shared mem */
        for k = 0 to threadDim.x do
            // Accumulate for matrix C
            accu <= accu + A_tile(threadIdx.y,k) * B_tile(k,threadIdx.x)
        end
        // Synchronize
        __sync()

    end

    // Row i of matrix C
    i <= blockIdx.y * blockDim.y + threadIdx.y
    // Column j of matrix C
    j <= blockIdx.x * blockDim.x + threadIdx.x
    // Store accumulated value to C(i,j)
    C_gpu(i,j) <= accu
}
```

In the tiled implementation, the amount of computation is still $2 \cdot M \cdot N \cdot K$ FLOPs. However, using tile size of B, the amount of global memory access is $2 \cdot M \cdot N \cdot K/B$ words. The "computation-to-memory ratio" is approximately $\frac{B}{4}$ FLOP/byte. We now can tune the "computation-to-memory" ratio by changing the tile size B.

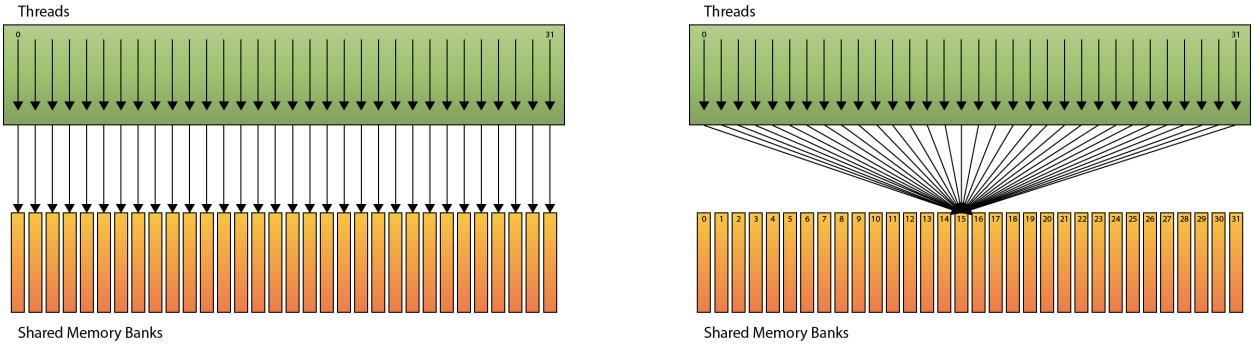
Exercise 4.2: Using tiling & shared memory

Modifying `IA-GPU/LabAssignments/optimizedCUDAcore.h`, create a plot of runtime versus tiling factor. Assume square matrices with size $m = n = k = 4096$. Use tiling factors that are a power of two.

Reflect on why certain tiling factors are good and others are not.

Note: Un-comment the corresponding part in `IA-GPU/LabAssignments/main.cu`

4.3 Memory coalescing and avoiding bank conflicts



(a) No bank conflict for matrix A.

(b) Bank conflict for matrix B.

Figure 13: Illustration of threads loading data without bank conflicts and with bank conflicts. Image from here.

Two dimensional arrays in C/C++ are row-major. In the tiled implementation above, neighbouring threads have coalesced access to matrix A (Fig. 13a), but do **not** have coalesced access to matrix B (Fig. 13b). In column-major languages, such as Fortran, the problem is the other way around. An obvious solution is to transpose matrix B by CPU before offloading it to GPU memory.

Exercise 4.3: Memory coalescing and avoiding bank conflicts

Implement matrix transpose operation on CPU (`IA-GPU/LabAssignments/helper.h`). In addition, implement coalesced access without bank conflicts to matrix B on GPU (`IA-GPU/LabAssignments/optimizedCUDAcore.h`)
Report:

1. Runtime of the transpose operation on CPU
2. Runtime (and speedup, if any) of the complete matrix multiplication including the transpose operation

Reflect on the extra operations introduced: is transposing matrix B worth it?

Hint:

```
Load B(i,j) to shared mem without bank conflicts

B_tile(threadIdx.y, threadIdx.x) <= B_gpu(i,j) //Note indexing changed
// Accumulate for matrix C without Shared Mem Bank conflict
accu <= accu + A_tile(threadIdx.y,k) * B_tile(k,threadIdx.x)
```

4.4 Quantization

Deep neural network architectures have a natural resilience to errors due to the backpropagation algorithm used in training them, and some have argued that 16-bit floating point (half precision, or float16) is sufficient for training neural networks.

Storing float16 (half precision) data compared to higher precision float32 reduces memory usage of the neural network, allowing training and deployment of larger networks, and float16 data transfers take less time than float32 transfers. Moreover, for many networks deep learning inference can be performed using 8-bit integer computations without significant impact on accuracy.

Exercise 4.4: Quantizing float32 to float16

Implement matrix-multiplication kernel using float16 matrix A and B.
Hint: for switching datatype look at `configs.h`
Hint: memory coalescing and avoiding bank conflicts!

Report:

- Mean Squared Error between unquantized and quantized kernel's output.
- Runtime and speedup, if any.

Reflect on speedup, if any. What changes did you expect?
Hint: check Nvidia Turing architecture.

5 Acceleration using Tensor Cores

In order to boost performance for deep learning applications, Nvidia introduced the Tensor Core unit in the Volta GPU architecture. The Tensor Core unit can be seen as a domain-specific accelerator for GEMM but is integrated in the GPU. As a result, users can get both the advantages of a domain-specific accelerator (i.e. higher performance efficiency for the dedicated application) and of using a general purpose platform (i.e. better programmability).

In this assignment, you will learn how to use the Tensor Core unit to accelerate your application.

5.1 WMMA API [2]

Tensor Cores are exposed in CUDA9.0 via a set of functions and types in the `nvcuda::wmma` `namespace`. These allow you to load or initialize values into the special format required by the tensor cores, perform matrix multiply-accumulate (MMA) steps, and store values back out to memory. During program execution multiple Tensor Cores are used concurrently by a full warp. This allows the warp to perform a $16 \cdot 16 \cdot 16$ MMA at very high throughput (Fig. 14).

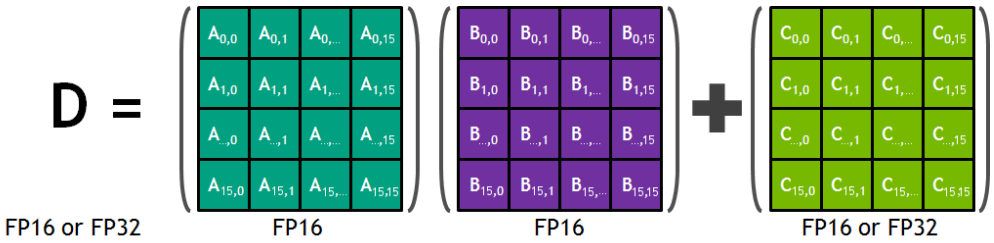


Figure 14: A warp performs $D = A \cdot B + C$ where A, B, C and D are $16 \cdot 16$ matrices.

Let's go through a simple example that shows how you can use the WMMA (Warp Matrix Multiply Accumulate) API to perform a complete matrix multiplication.

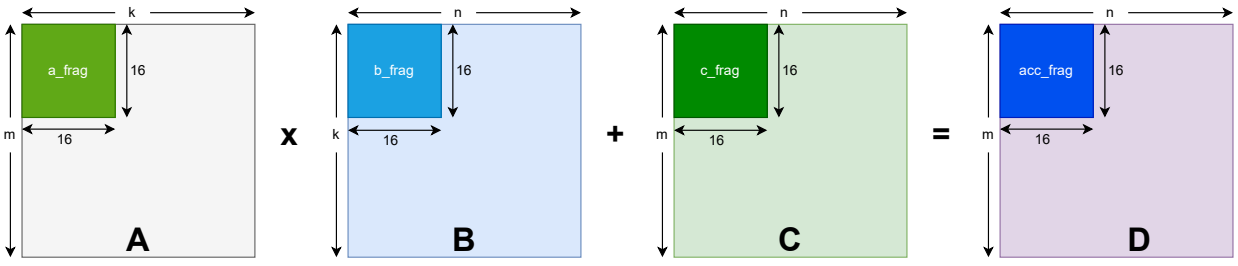


Figure 15: Illustration of complete MM using the WMMA naming scheme for matrices, variables and dimensions.

Fig. 15 acts as a reference for the naming of matrices, variables and dimensions.

[2] Source: https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/

5.1.1 Headers and Namespaces

The WMMA API is contained within the `mma.h` header file. The complete namespace is `nvcuda::wmma::*`, but it's useful to keep wmma explicit throughout the code, so we'll just be using the nvcuda namespace.

```
#include <mma.h>
using namespace nvcuda;
```

5.1.2 Declarations and Initialization

For simplicity let's assume that neither A nor B are transposed, and that the leading dimensions of the memory and the matrix are the same. The strategy we'll employ is to have a single warp responsible for a single $16 \cdot 16$ section of the output matrix. By using a two-dimensional grid and thread blocks, we can effectively tile the warps across the two dimensional output matrix.

```
__global__ void simple_wmma_gemm(half *A, half *B, float *C, float *D, int
    gemmM,int gemmN, int gemmK, float alpha, float beta){
    // Tile using a 2D grid
    // compute global warp id
    int warpM = (blockIdx.x * blockDim.x + threadIdx.x) / 32; //  warpSize
    = 32
    int warpN = (blockIdx.y * blockDim.y + threadIdx.y);
```

Before the MMA operation is performed the operand matrices must be represented in the registers of the GPU. As an 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. The mapping between individual matrix parameters to their fragment(s) is opaque, so your program should not make assumptions about it.

In CUDA a fragment is a templated type with template parameters describing which matrix the fragment holds (A, B or accumulator), the shape of the overall WMMA operation, the data type and, for A and B matrices, whether the data is row or column major. This last argument can be used to perform transposition of either A or B matrices. This example does no transposition so both matrices are column major, which is standard for GEMM.

```
    // Declare the fragments
    wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::col_major>
        a_frag;
    wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::col_major>
        b_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, float> acc_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;
```

The final part of the initialization step is to fill the accumulator fragment with zeros.

```
   wmma::fill_fragment(acc_frag, 0.0f);
```

5.1.3 The Inner Loop

The strategy we're using for the GEMM is to compute one tile of the output matrix per warp. To do this we need to loop over rows of the A matrix, and columns of the B matrix. This is along the k-dimension of both matrices and produces an $m \cdot n$ output tile. The load matrix function takes data from memory (global memory in this example, although it could be any memory space) and places it into a fragment. The third argument to the load is the "leading dimension" in memory of the matrix; the $16 \cdot 16$ tile we're loading is discontinuous in memory so the function needs to know the stride between consecutive columns (or rows, if these were row-major fragments).

The MMA call accumulates in place, so both the first and last arguments are the accumulator fragment we previously initialized to zero.

```
    // Loop over k
    for (int i = 0; i < gemmK; i += 16) {
        int aCol = i;
        int aRow = warpM * 16; // offset
        int bCol = warpN * 16; //offset
        int bRow = i;

        // Bounds checking
        if (aRow < gemmM && aCol < gemmK && bRow < gemmK && bCol < gemmN){
           // Load the inputs
           wmma::load_matrix_sync(a_frag, a + aRow + aCol * gemmM, gemmM);
           wmma::load_matrix_sync(b_frag, b + bRow + bCol * gemmK, gemmK);

           // Perform the matrix multiplication
           wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
        }
    }
```

5.1.4 Finishing Up

`acc_frag` now holds the result for this warp's output tile based on the multiplication of A and B. The complete GEMM specification allows this result to be scaled, and for it to be accumulated on top of a matrix in place. One way to do this scaling is to perform element-wise operations on the fragment. Although the mapping from matrix coordinates to threads is not defined, element-wise operations do not need to know this mapping so can still be performed using fragments. As such, it is legal to perform scaling operations on fragments or to add the contents of one fragment to another, so long as those two fragments have the same template parameters. If the fragments have different template parameters the results are undefined. Using this feature, we load in the existing data in C and, with the correct scaling, accumulate the result of the computation so far with it.

```
    // Load in the current value of C, scale it by beta, and add this our
   result
    // scaled by alpha
    int cCol = warpN * 16;
    int cRow = warpM * 16;
```

```
5
6       if (cRow < gemmM && cCol < gemmN) {
7           wmma::load_matrix_sync(c_frag, C + cCol + cRow * gemmN, gemmN,
    wmma::mem_row_major);
8
9           for (int i = 0; i < c_frag.num_elements; i++) {
10              c_frag.x[i] = alpha * acc_frag.x[i] + beta * c_frag.x[i];
11          }
```

Finally, we store the data out to memory. Once again the target pointer can be any memory space visible to the GPU and the leading dimension in memory must be specified. There is also an option to specify whether the output is to be written row or column major.

```
1           // Store the output
2           wmma::store_matrix_sync(D + cCol + cRow * gemmN, c_frag, gemmN,
    wmma::mem_row_major);
3       }
4   }
```

With that, the matrix multiplication is complete.

5.2 Tensor core exercises

Exercise 5.1: Naive tensor core mapping

Run the matrix-multiplication on the tensor cores using the code above. Basically, copy and paste to `IA-GPU/LabAssignments/baselineTensorCore.h`

Report runtime and speedup, if any.
Can you explain the magnitude of speedup between the tensor core and float16 mapping?

Exercise 5.2: Optimized mapping

Optimize your tensor core mapping using the optimizations you have seen in the previous section. Implement your optimized kernel in `IA-GPU/LabAssignments/optimizedTensorCore.h`

Report and reflect on the runtime and speedup, if any, of each optimization tried.

Exercise 5.3: Overview of all mappings

Plot the runtime of the best CPU, GPU (float32 / float16) and tensor core mappings acquired.

6 Report

This is an individual assignment. The report should consist of at most 3 pages (1 side) of A4 paper in IEEE double column format. Please also submit a zip file which includes all written source code. Any code that was used as a base for projects must be referenced and cited in the body of the paper. This includes example code, open-source, or Github implementations. You can use a footnote or full reference/bibliography entry.

Some other guidelines for the reports are:

- Do not just present the numbers, but also reflect on them. Were you expecting the obtained figures? Include all the optimization techniques you have deployed, how the outcomes relate to your expectations.
- Explain results in a concise and clear way. Your understanding of the results is much more important than the results themselves.
- It is important to understand the fundamental limits. E.g. when it does not make sense to optimize a certain aspect further.
- Tables and graphs will help you explain results without long sections of text.
- Itemize the optimizations so they are easily traceable.
- If you use ideas or code from someone else (or work together) include the proper acknowledgements in your report. Failing to do so will be considered fraud.

7 Optional acceleration exercises for bonus points

For maximum 1 bonus point, instead of the bonus presentation, you may do optional exercises to further accelerate convolutional layers on GPUs. As a starting point one could think of the following optimizations:

- Perform the im2col transformation during the execution of the GPU kernel.
- Mapping an entire (small) CNN to GPU, e.g. LeNet-5.
- Convolution implementation on tensor core using 8-bit weights and activations.