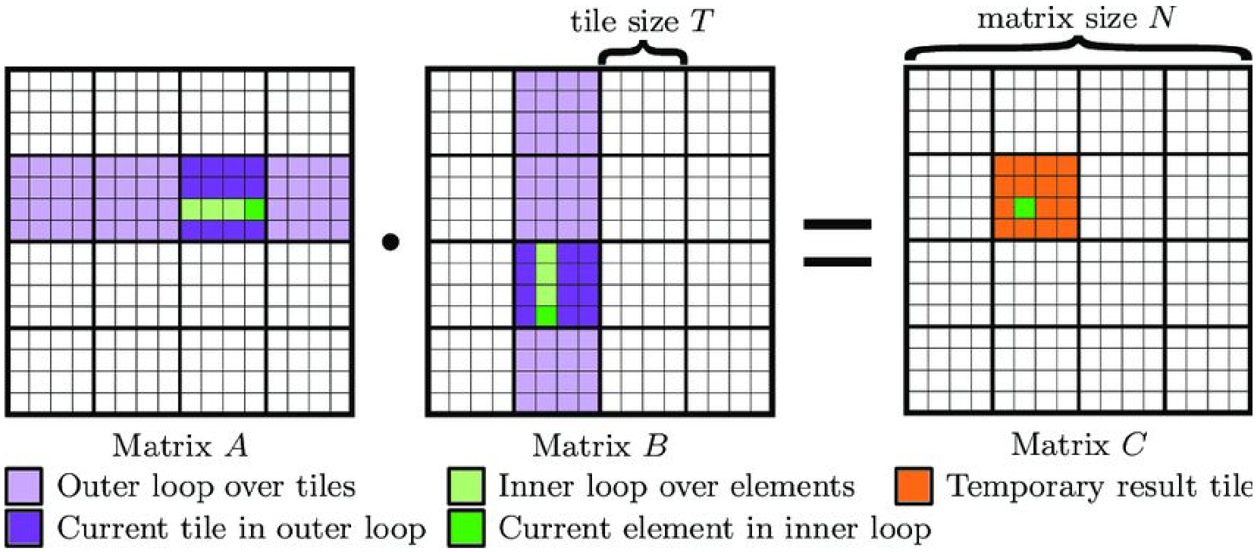
GEMM stands for General Matrix to Matrix Multiplication, and it essentially does exactly what it sounds like: multiplies two input matrices together to get an output matrix.
In the context of neural networks, GEMM is usually aimed at works where matrix sizes are often very big. For instance, a single layer in a typical network may require the multiplication of a 256 row, 1,152 column matrix by an 1,152 row, 192 column matrix to produce a 256 row, 192 column result. This amounts to about 57 million floating point operations, and a typical network would contain several such layers, all containing several such multiplications.

The inspiration of GEMM comes from NVidia, who use a particular implementation of GEMM for convolution operations on their Basic Linear Algebra Subroutines (BLAS) library. So how exactly does GEMM aid in improving the speed of our CNN?

There's two parts to the solution:

- It employs a unique “tiled” multiplication algorithm to speed up naive matrix multiplication
- It performs im2col and col2im techniques, which are mathematical operations used with matrix multiplication to produce an operation equivalent to direct convolution.

Speeding up matrix multiplication

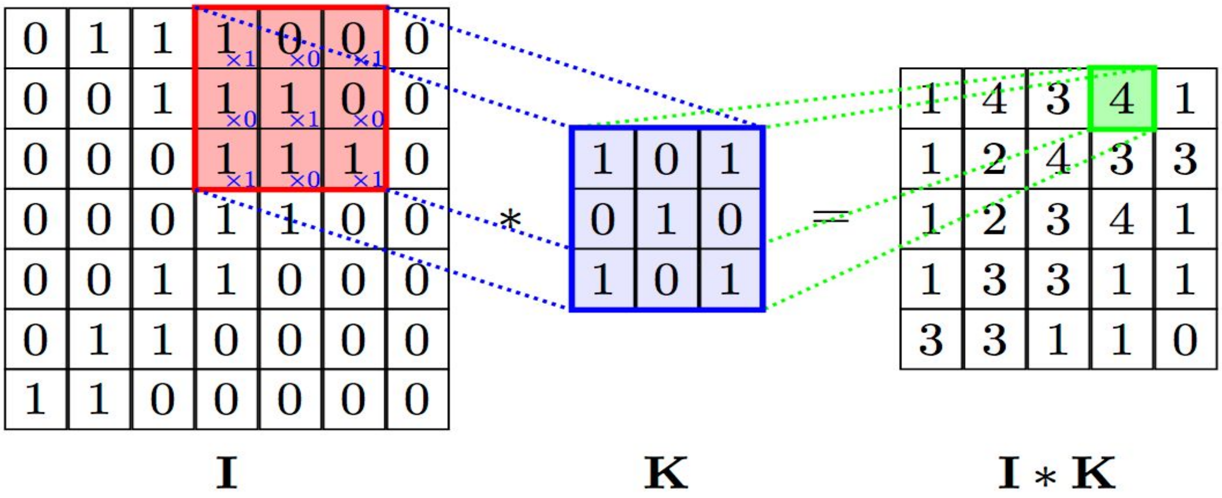


Reference https://www.researchgate.net/

While a naive matrix multiplication loops over the matrices index by index, and perform multiplications and additions to attain the product matrix, one index item each time; our multiplication algorithm breaks the input matrices down into tiles, loops over each tile at a time, and accumulates the correct results for those. While it might not be as fast on its own, it carries huge potential for parallelisation. Not only can each tile be a separate block, evaluated on its own concurrently with all other tiles, but it also cuts down the number of times the memory has to be accessed since the entire tile is cached at a time. These two advantages are particularly effective if terms of FPGA programing, since parallelization is the most effective method to minimise the latency of a program.

Im2Col and Col2Im

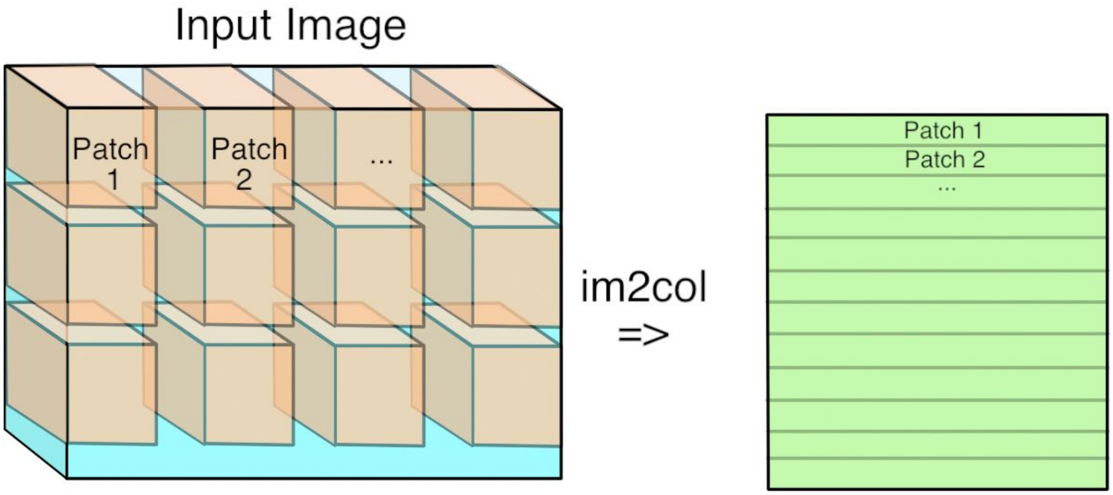
One of the most important, and repeatedly employed operation is convolution. This is where a “kernel” Matrix is “convolved” over an input matrix, one representing an image in our case. The smaller kernel matrix is operated on input matrix one by one on each “patch”. This patch can be seen highlighted in red in the following image. The value of each index from the patch is multiplied by the corresponding value in the kernel matrix, and added together to give the corresponding result highlighted in green. This operation allows us to extract “features” from a given input image.



$$\begin{array}{ccccccc} 0 & 1 & 1 & 1_{\times 1} & 0_{\times 0} & 0_{\times 1} & 0 \\ 0 & 0 & 1 & 1_{\times 0} & 1_{\times 1} & 0_{\times 0} & 0 \\ 0 & 0 & 0 & 1_{\times 1} & 1_{\times 0} & 1_{\times 1} & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{array} * \begin{array}{ccc} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{array} = \begin{array}{ccccc} 1 & 4 & 3 & 4 & 1 \\ 1 & 2 & 4 & 3 & 3 \\ 1 & 2 & 3 & 4 & 1 \\ 1 & 3 & 3 & 1 & 1 \\ 3 & 3 & 1 & 1 & 0 \end{array}$$

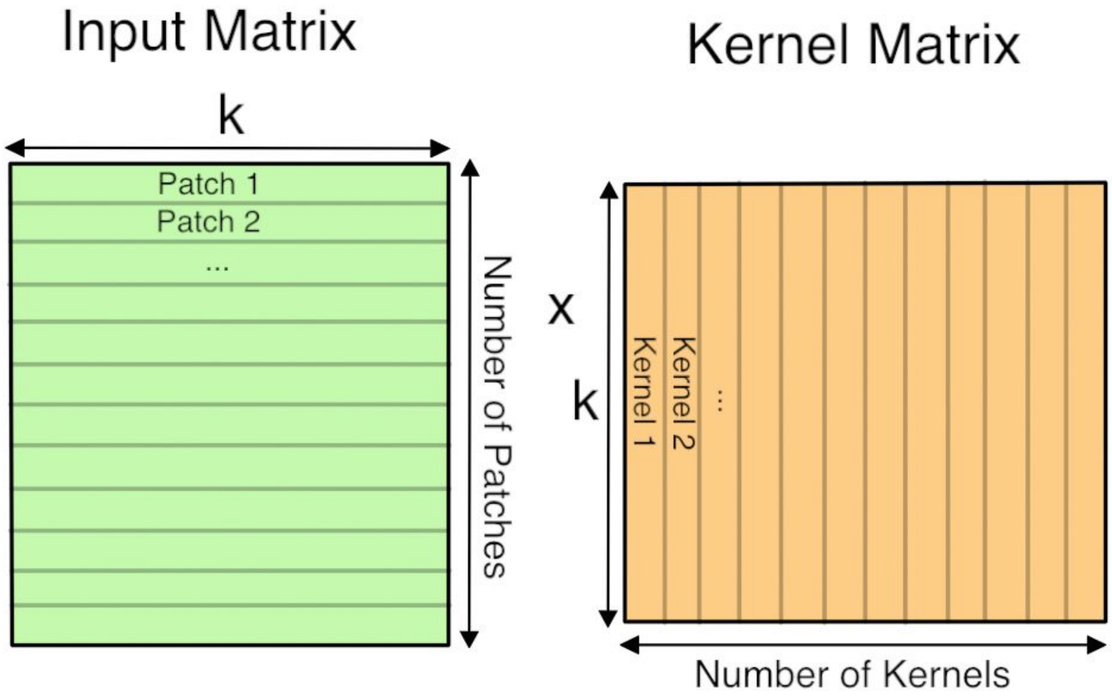
Reference https://www.oreilly.com/library/view/learn-unity-ml-agents/

The same operation however can be performed in a different manner, which can speed up the entire process at the cost of utilisation of more resources. We first convert input image into a single large matrix, where each row represents one linearised patch. This process is called Image to Column or “im2col”.



reference :https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/

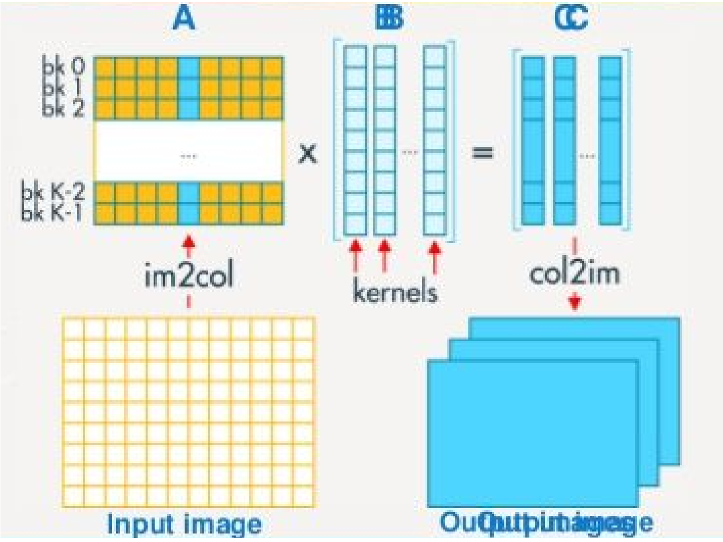
A similar process is done for the kernel matrices, where each kernel matrix is linearised into one column of a larger “kernel matrix”. Notice how Im2Col places each path row-wise, while kernel matrix contains all kernels column-wise. Using the faster GEMM multiplication algorithm discussed in the first part, these two large matrices are multiplied together as shown by the following figure to produce a product matrix.



Reference https://www.systems.ethz.ch/sites/default/files/parallel-distributed-deep-learning.pdf

This product matrix does not contain the dimensions we need however, the values correspond by index. For instance, in normal convolution, the first value would be result of multiplying the first kernel with the first patch,and so on. This is also exactly the same in this case.
We simply read of values from the product matrix in order it and resize it into our desired dimensions. This process is called Col2im.

The entire process in summarized into the image below.



Reference

SlideShare.net/embeddedvisionusing-sgemm-and-ffts-to-accelerate-deep-learning-a-presentation-from-arm

- What improvements could be made to the system if you had more time?
    - Improve the speed-accuracy tradeoff - figure out ways to make the CNN run faster so that we could accommodate a larger network (for higher accuracy and to support a higher number of output classes)
        - Finish Winograd
        - Learn more techniques in Vivado HLS to figure out how we can optimize the system better
        - Use automated neural architecture search through python packages like AutoKeras to find a better architecture that uses less weights or requires a lower number of operations but still has the same accuracy
        - Data augmentation - enlarge the training set to make the network more accurate - make it work with something drawn on paper / variety of backgrounds, stroke colors, stroke widths, stroke textures, etc.
        - See whether performing pre-processing and post-processing in hardware rather than in software improves latency
    - Explore other uses of our CNN - the code we wrote could be easily used for running other CNNs trained on other datasets (not just the QuickDraw dataset)

In terms of future improvements, our main area of focus would be exploring a better speed-accuracy trade-off. Our original network was 93% accurate, with 887000 parameters, which we later cut down to 29,000 parameters while retaining 89% accuracy. Such a drastic difference in parameters with little loss in accuracy indicates that there is a lot that could be improved within the structure of the layer, since it can be inferred that a considerable part of our network parameters did not contribute much. Streamiling input parameters means we have more resources to spare, which can be used to accomodate for more than just twenty classes. This would allow our network to detect more than just twenty objects.

Given more time, GEMM is a potential tool that can significantly improve our implementation. While we had managed to implement GEMM, we did not allocate enough time and effort to optimise it on the FPGA. The effectiveness of GEMM relies on parallel computation, several tools and directives that focus on parallel computation, like loop pipelining, unrolling and flattening are highly likely to be very effective. Time constraints prevented us from integrating GEMM into our direct convolution, but in the future it can be a useful addition.

We also noticed a significant drop in frame-rate due to the python processing scripts, bringing the overall network to 18.71 fps even though the hardware implementation of the CNN produces around 71 fps. Implementing the frame processing within the vivado hardware implementation along with the network might help improve the overall frame rate.

It is important to realise that while our network only works for recognising hand-drawn doodles, it is capable of running CNN's that have been trained of other datasets, not just the Google Quickdraw dataset we have used in our current implementation. In other words, we could reuse the code to train our network to recognise electrical components (e.g resistors, and transistors) or even food.

Google Docs is so much better than Microsoft Word