Accelerating Contextual Convolutions using CUDA

Varun Menon VK2148@NYU.EDU
Department of Computer Science
New York University

Aishwarya Raman AR6381@NYU.EDU
Department of Computer Science
New York University

Atharv Bhat ARB881@NYU.EDU
Department of Computer Science
New York University

Abstract

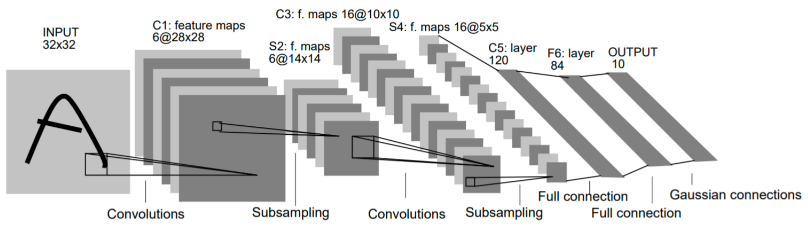
Convolutional Neural Networks form the backbone of all computer vision tasks and applications. Duta et al. (2021) proposes a new contextual convolution (CoConv) for visual recognition. CoConv is a direct replacement of the standard convolution, which is the core component of convolutional neural networks. CoConv is implicitly equipped with the capability of incorporating contextual information while maintaining a similar number of parameters and computational cost compared to the standard convolution using dilated kernels. The authors have implemented CoConv using PyTorch. Pytorch does not support multiple dilation factors for a single convolution layer. This forces the Authors to build CoConv as a Sequential model of 3-5 Convolution layers, each with it's appropriate dilation factor. This repeats 3 of the involved operations that is required to compute one "CoConv" operation, specifically, "Im2Col", "Kernel Unrolling" and "GEMM". The objective of this project is to parallelize the entire CoConv pipeline and compute the results in one set of "Im2Col", "Kernel Unrolling" and "GEMM" operations.

1. Introduction

When LeCun et al. (1989) proposed the use of CNNs for image recognition tasks, it revolutionized the field of computer vision. CNNs exploit the structure of an image and use their inherent property of locality to form a weight sharing network built on the convolution operation where the weights of a convolution filter are learned using backpropagation.

In state-of-the-art CNN architectures, the receptive field of the kernel is fixed since only one type of kernel with fixed size is used to perform convolutions. However, these architectures do not have the ability to integrate contextual information that is vital for visual perception as shown by Duta et al. (2021)

Figure 1: CNN architecture (LeNet) proposed by LeCun et al. (1989)



Duta et al. (2021) propose contextual convolution (CoConv), a direct replacement of the standard convolution that can be used at any stage in CNN architectures. CoConv helps the network learn features with different levels of context at each level and does this while preserving the parameter count. The authors show this greatly improves the network's ability to learn global features and report improved detection, recognition and generation capabilities over standard Convolution.

2. Convolution

The discrete 2D convolution is a fairly straightforward operation. It can be seen as a sliding window of size K×K performing element-wise multiplication at each possible location and finally summing up the results. Depending on the size of the kernels, only certain pixels that are spatially close enough in the input are responsible for corresponding output pixels. This means that the kernel size directly dictates how much context a convolution layer has. Increasing the kernel size to increase contextual information leads to increase computation. This is where dilated convolutions come into play.

Let $F : \mathbb{Z}^2 \rightarrow$ be a discrete function. Let $\Omega_r = [-r, r]^2 \cap \mathbb{Z}^2$ and let $k : \omega_r \rightarrow \mathbb{R}$ be a discrete filter of size $(2r+1)^2$. The discrete convolution operator $*$ can be defined as,

$$(F * k)(\mathbf{p}) = \sum_{\mathbf{s}+\mathbf{t}=\mathbf{p}} F(\mathbf{s})k(\mathbf{t})$$

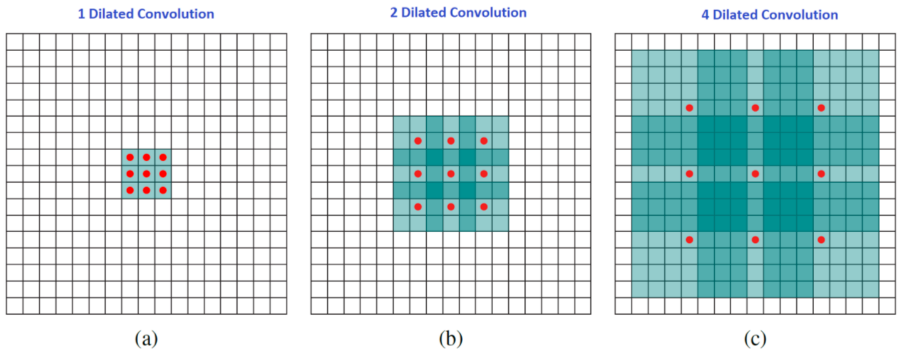
3. Dilated Convolution

Dilated convolution can be thought of as the standard discrete 2D convolution with padded kernels (pixel skipping). They are use to increase the effective receptive field of a convolution operation without increasing the number of parameters. Dilation=2 means the weights in the kernel matrix have a padding of (Dilation-1) around them, in this case, a padding of 1.

Let l be a dilation factor and let $*_l$ be defined as the dilated convolution operator. Dilated Convolution can then be defined as,

$$(F *_l k)(\mathbf{p}) = \sum_{\mathbf{s}+l\mathbf{t}=\mathbf{p}} F(\mathbf{s})k(\mathbf{t})$$

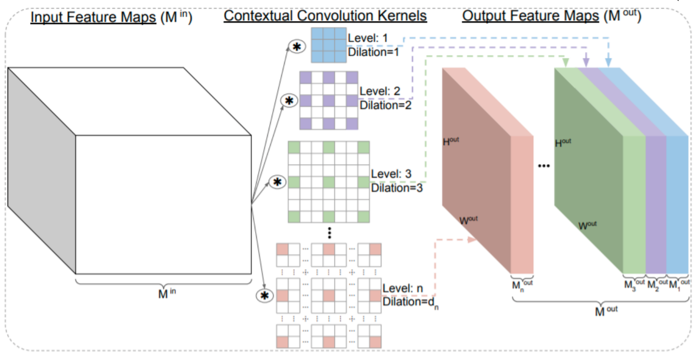
Figure 2: Dilated kernels with 1, 2 and 4 levels of dilation shown by Yu and Koltun (2016)



4. CoConv

CoConv (illustrated in Fig 3) proposed by Duta et al. (2021). The authors apply different dilation factors to sets of kernels within one convolution block: $D = \{d_1, d_2, d_3, \ldots, d_n\}$. At every convolution layer, each set of dilated kernels allows the network to build over different levels of context. Kernels with lower dilations help with understanding local features vs kernels with larger dilations which help build a global understanding of the input volumes.

Figure 3: CoConv architecture proposed by Duta et al. (2021)


Input Feature Maps (M^{in})
Contextual Convolution Kernels
Output Feature Maps (M^{out})
Level: 1
Dilation=1
Level: 2
Dilation=2
Level: 3
Dilation=3
Level: n
Dilation=d_n
M^{in}
H^{out}
W^{out}
M_n^{out}
M_3^{out} M_2^{out} M_1^{out}
M^{out}

5. Engineering Convolutions

5.1 Naive Convolution

The naive implementation is quite simple to understand, we simply traverse the input matrix and pull out "windows" that are equal to the shape of the kernel. For each window, we do simple element-wise multiplication with the kernel and sum up all the values. Finally, before returning the result we add the bias term to each element of the output. This results in looping over the number of kernels, input channels, output height, output width, kernel height and kernel width. As easy as the implementation is, it is extremely expensive to compute.

5.2 Convolution by Matrix Multiplication

The entire process of Naive Convolutions can be optimised by converting the problem into a single matrix multiplication (GEMM) operation. To facilitate using GEMM, we need to pre-process the input volume using "Im2col" and the Kernels using "Kernel Unrolling".

5.2.1 IM2COL

We have to convert the 3D input volume into a 2D matrix to use matrix multiplication. Every possible 3D volume a kernel operates on is called a patch. As shown in Figure 4, each possible KxKxC patch in the input volume is flattened into a column in the im2col operation. Sequentially, applying im2col on an input volume of size H×W×C will result in (H×W×C)×(K×K) operations, where H is the height, W is the width and C is the number of channels of the input volume and K is the Kernel size. We parallelize this operation at the outermost level saving computations by a factor of H×W×C (averaging in the order on 10e5).

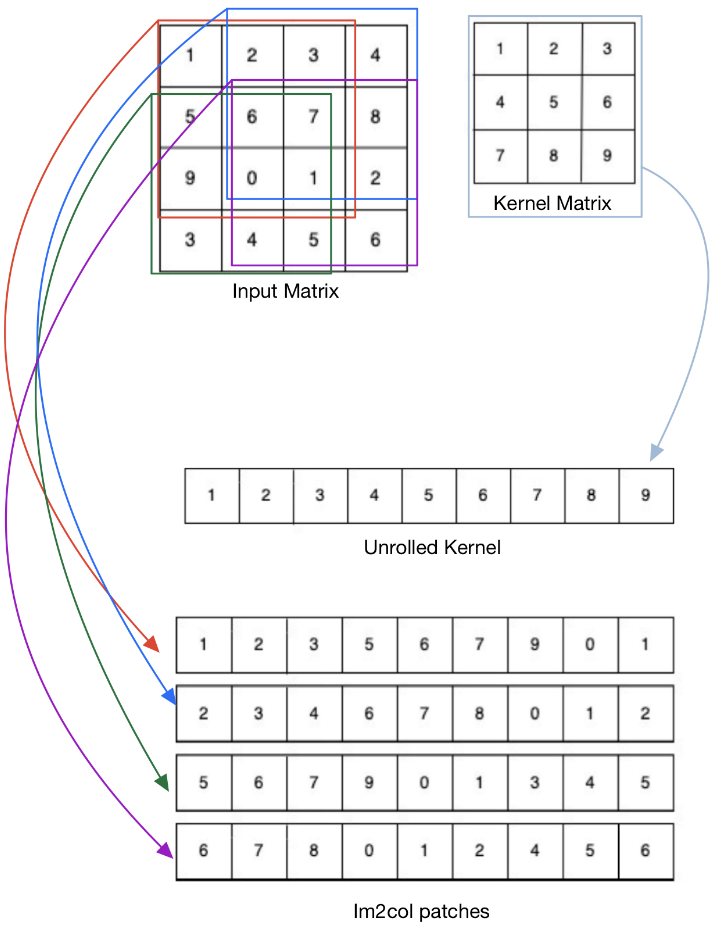
5.2.2 KERNEL UNROLLING

Similarly, we flatten the 3D Kernel volume to apply GEMM on the output of im2col and the flattened kernels. Since the kernel sizes are usually small, we do not see any improvement parallelising this part of the algorithm and leave it as sequential. The catch in this project is that we have to index it appropriately to be able to handle dilated convolutions. This entails the kernels being padded with zeroes to space them out for the correct element-wise multiplication.
To support kernels with different dilation factors, we need to pad the Kernel matrix with appropriate number of zeros to match the correct weight, input pair. Section 6 will have a detailed description on how this is achieved.

5.2.3 GEMM

To compute the result of convolution, we use GEMM (General Matrix to Matrix Multiplication) on the flattened kernel matrix and im2col matrix. GEMM takes both the input matrices in col major form and returns the output matrix of dimension $C_{out} \times H \times W$ where C_{out} is the output channels, H is the height of the input matrix and W is the width of the input matrix. The number of GEMM kernels called is equal to the dimension of the output matrix i.e $C_{out} * H * W$. The GEMM algorithm we use (https://github.com/aditisingh/GPU-

Figure 4: Im2col and Kernel Unrolling (Im2col patches are shown as rows only for the ease of illustration.)



Gemm), utilizes Tiling to efficiently fetch matrices from Global Memory. We divide the two input matrices into tiles of size 32x32. At a given time an instance of shared memory contains two tiles, one from kernel matrix and another from im2col matrix. The threads then access the content of the shared memory to calculate the result matrix. We use syncthreads() to ensure all threads have completed computing result for the current tiles before moving on to the next one.

6. Implementation

We have to pad the input to preserve the output shape since dilated convolutions reduces the output resolution. This only affects how we pad the kernel weights, so we will ignore it's effects in the Im2col section.

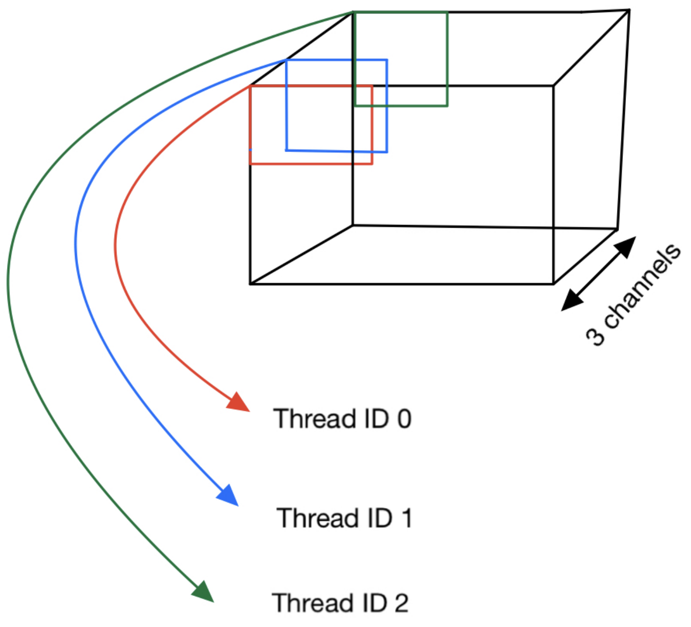
6.1 Im2col indexing

We parallelize Im2col such that each thread is responsible for flattening a 2D KxK window in the input volume, as shown in Figure 5. Equation 1 shows how the input volume is indexed to the 2D output Matrix. MatAc is the output 2D matrix which will be used for GEMM and matA is the input volume.

$$\begin{aligned} &matAc[((idx/C) * (K * K * C)) + (r * K * K) \\ &\quad + (x * K + y)] = matA[(r * (H^p * W^p)) + (w * W^p + h)] \end{aligned} \tag{1}$$

where idx is the ThreadID, C is the number of input channels K is the size of maximum dilated kernel, r is the ChannelID, H^p is the padded height of the input volume, W^p is the padded width of the input volume and h and w are temporary iterators to index elements within the current KxK 2D window.

Figure 5: Parallelizing Im2Col



6.1.1 Breakdown of the indexing

LHS:

((idx / C)*(K * K * C)) → skips (K * K * C) indices depending on which patch is being indexed.
(r * K * K) → skips (K * K) indices depending on which channel is being filled.

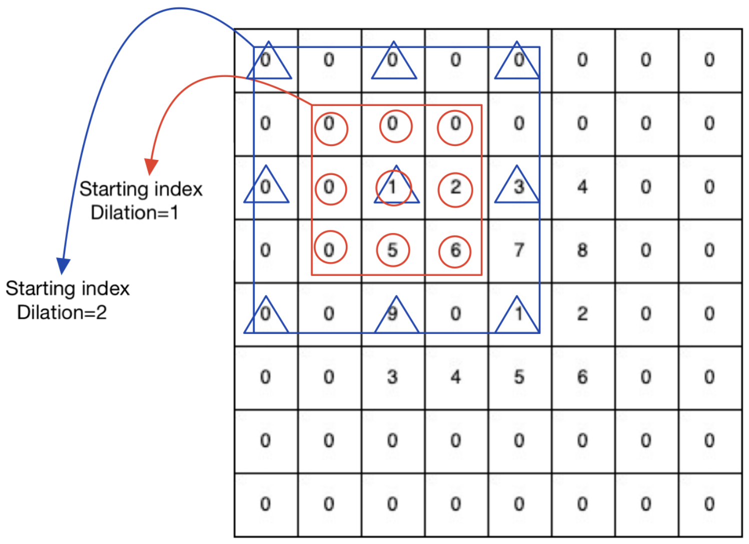
(x * K + y) $\rightarrow$ gives the index of the element within the K * K window.

RHS:

(r * (H^p * W^p)) $\rightarrow$ skips to the correct channel.
(w * W^p + h) $\rightarrow$ index of the element within the K * K window.

6.2 Kernel unrolling indexing

Figure 6: Kernel Unrolling for dilated convolutions, Triangles and circles show weight location for respective kernels. Red$\rightarrow$Dilation = 1, Blue$\rightarrow$Dilation = 2



$$itr = kid * ((K * K) * C) + (d - dilation) * K + (d - dilation) \tag{2}$$

Equation 2 gives the starting point of each padded kernel depending on what the dilation of the current kernel is, e.g., the red kernel in Figure 6 has dilation = 1 which is less than the maximum dilation (d=2) and therefore is offset by a certain value, which is 1 in this case. The blue kernel, on the other hand, has the same dilation as max dilation and therefore is not offset.

$$\begin{gathered} itr = kernelid * (K * K) * C + (dilation - 1) * (Cout/4) * (K * K) * C + \\ ((weightid + 1)/(k * k)) * (K * K) + (d - dilation) * K + (d - dilation) \end{gathered} \tag{3}$$

where Cout is the number of output channels. Equation 3 gives the index to skip to, for the next starting point after filling in on K*K window size. This also includes the same offset as discussed before.

$$\begin{aligned} itr = ((weightid\%k) * (dilation - 1)) + (weightid\%(k * k)/k) * (K * K) \\ * (dilation - 1) + (weightid\%k) + (weightid\%(k * k)/k) * k \end{aligned} \tag{4}$$

Equation 4 gives the index of each weight depending on the dilation of the current kernel and incorporates how many zeros to pad before setting the current weight.

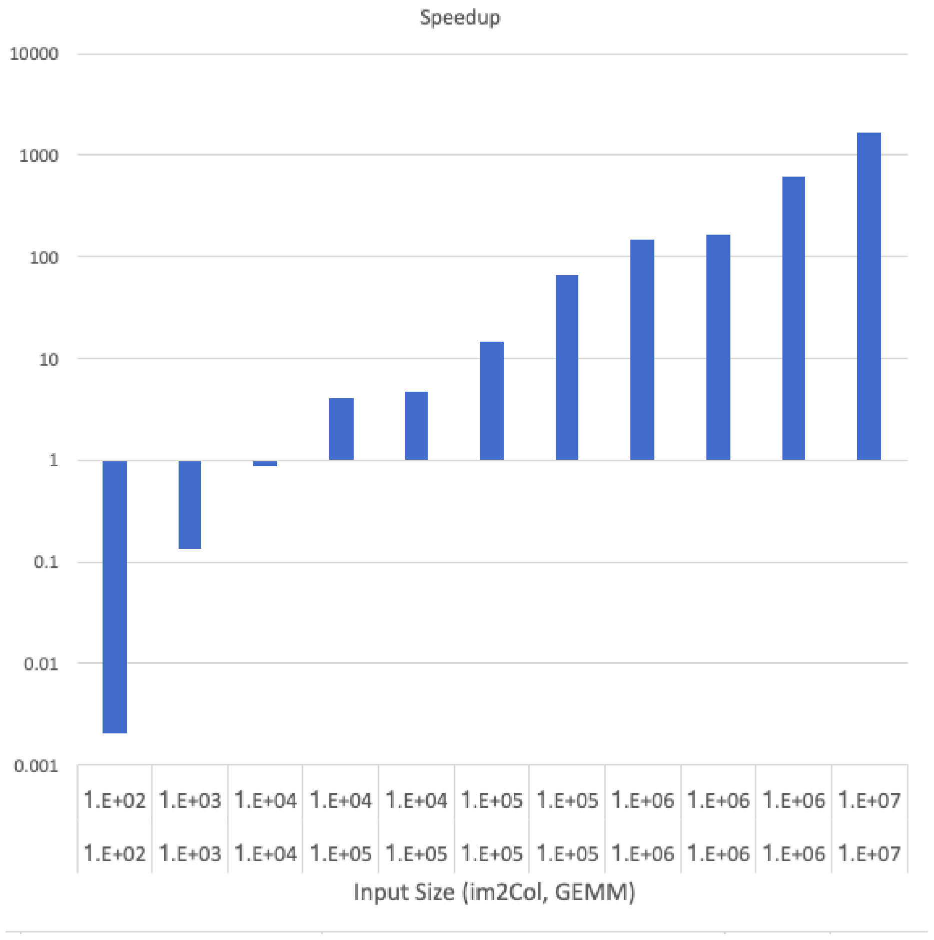
7. Results

We compared our parallel implementation of CoConv algorithm in CUDA against sequential version(CPU) and pytorch. We used Mean Square Error (MSE) to check correctness of our parallel and sequential algorithms with pytorch. We got an MSE of 0.0 for both versions of the algorithm. We used speedup to compare the performance of sequential and CUDA version. As can be seen from Figure 7, the speedup is low for small input sizes but increases significantly as input sizes increase. We achieved a maximum speedup of **1600x** against the CPU version. We used execution time as a measure to compare the performance of pytorch with our CUDA version. From the results in Table 2, we can see that CUDA version performs better than pytorch for all input sizes. We ran all three implementations(Sequential, CUDA and Pytorch) on CUDA clusters 2,3 and 5. We found CUDA 2 to be the slowest and CUDA 4 to be the fastest. We used the average time taken which was on CUDA 3 to compare performances.

Input Size (im2col, GEMM)	CPU Time	GPU Time	Speedup
(75, 100)	0.425	208	0.00204
(3072, 2048)	24.299	179.481	0.135
(32768, 12288)	165.912	189.25	0.876
(131072, 49152)	551.926	135.94	4.06
(131072, 65536)	781.265	167.64	4.66
(262144, 131072)	3306.033	226.037	14.626
(409600, 409600)	14911.199	222.084	67.142
(1000000, 1000000)	34492.828	233.251	147.878
(1638400, 1638400)	60498.508	369.057	163.928
(6553600, 6553600)	242430.468	396.469	611.474
(25000000, 25000000)	1054068.75	641.7	1642.616

Table 1: A comparison of parallel algorithm(GPU) and sequential algorithm(CPU) by calculating Speedup (CPU time/GPU time)

Figure 7: Plot of Speedup (CPU vs GPU Time)



CoConv CUDA Time (ms)	PyTorch Time (ms)	Input Dimension (H x W x C)	Output Dimension (H x W x C)
208	4511	5 x 5 x 3	5 x 5 x 4
179.481	3733	16 x 16 x 12	16 x 16 x 8
189.25	3863	32 x 32 x 32	32 x 32 x 8
135.94	3796	64 x 64 x 32	64 x 64 x 12
167.64	3876	64 x 64 x 32	64 x 64 x 16
226.037	3890	64 x 64 x 64	64 x 64 x 64
222.084	3980	64 x 64 x 100	64 x 64 x 32
233.251	4890	100 x 100 x 100	100 x 100 x 100
369.057	4150	128 x 128 x 100	128 x 128 x 100
396.469	3737	256 x 256 x 100	256 x 256 x 100
641.7	4080	500 x 500 x 100	500 x 500 x 100

Table 2: A comparison of CUDA CoConv algorithm with pytorch

References

Ionut Cosmin Duta, Mariana Iuliana Georgescu, and Radu Tudor Ionescu. Contextual convolutional neural networks. *arXiv preprint arXiv:2108.07387*, 2021.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989. doi: 10.1162/neco.1989.1.4.541.

Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions, 2016.