FPGA Accelerated Transposed Convolutional Layers

Colman Glagovich and Daniel C. Stumpp
ECE, University of Pittsburgh
Pittsburgh, PA, USA
{cjg91, dcs98}@pitt.edu

I. Problem Description

For this research we propose the exploration of FPGA acceleration of the transposed convolution kernel. This kernel exists in many state-of-the art machine learning networks. These networks would benefit from a high performance FPGA implementation of the transposed convolution.

A. Transposed Convolution

The transposed convolution, sometimes referred to as a deconvolution or fractionally strided convolution, is a dense upsampling using learned parameters. Although sometimes referred to as a "deconvolution", it is not reversing the convolution. The transposed convolution can be considered conceptually as a direct convolution on a padded version of the input [1]. Fig. 1 shows a visual example of the transposed convolution implemented by convolving a 3×3 kernel over a padded 3×3 input to generate a 6×6 output feature map.

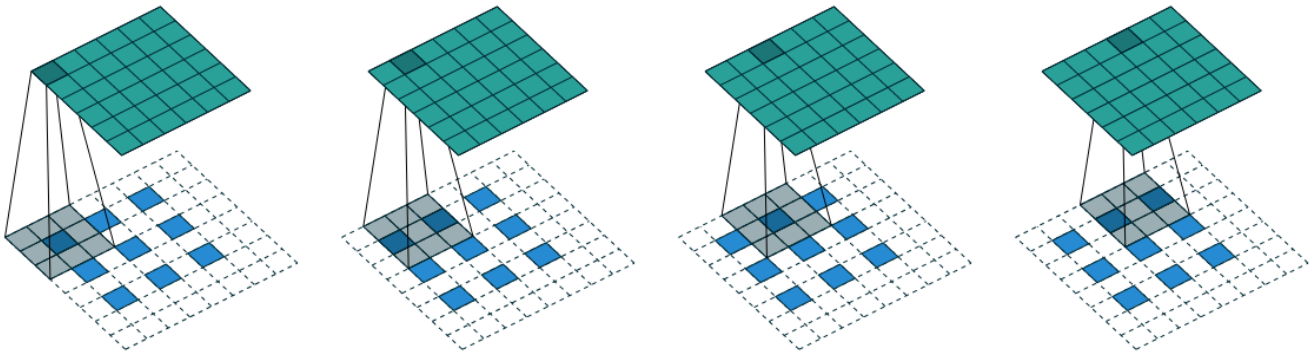

Fig. 1: Transposed convolution visualization. Represents kernel size of 3, stride of 2, and input padding of 1 [1]

The output size of the transposed convolution is defined as shown in (1), where k is the kernel size, s is the stride, i is the input size, and p is the input padding. Additionally, a is the number additional rows and columns of padding to add to the bottom and right of the input feature map [1].

$$o = s(i-1) + a + k - 2p \tag{1}$$

The extensive use of padding required to compute the transposed convolution as a standard direct convolution makes it computationally inefficient. Instead the transposed convolution can be implemented as matrix multiplication or with another optimized algorithm. Given a convolution matrix $\mathbf{C}$ used to compute the direct convolution, the corresponding transposed convolution can be found by multiplying the input feature map with $\mathbf{C}^T$ [1]. Other optimizations can be made to reduce the amount of inefficient computation incurred by multiplication by zero. These optimizations have the potential to be more fully leveraged using a custom acceleration architecture implemented on an FPGA.

B. Use Cases

The transposed convolution is used in many different machine learning applications. Any case that requires upsampling of a feature map with learned parameters is a likely use case for the transposed convolution. Semantic segmentation is a common instance of transposed convolution used [2]. Segmentation models typically involve an encoder that is a convolutional network followed by a decoder that is a ”deconvolutional” network, which leverages the transposed convolution to do learned upsampling. Another common use case is in Generative Adversarial Networks (GANs) introduced in [3].

II. Background and Prior Work

There is some existing work on the FPGA acceleration of 2D transposed convolutions for use in machine learning applications. An FPGA accelerator for Winograd transposed convolutions is introduced in [4].The Winograd algorithm is used to reduce the inefficiency of transposed convolution operation caused by the addition of zero padding. The transposed convolution is decomposed into smaller, more efficient operations, which each generate their own partial feature maps. These partial feature maps are then combined as shown in Fig. 2.

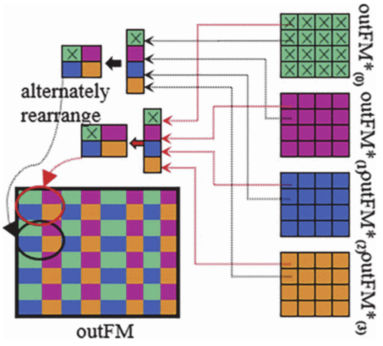

Fig. 2: Rearrangement of partial feature maps into output feature map [4]

The work also proposes an FPGA acceleration architecture for their proposed Winograd transposed convolution algorithm. Fig. 3 shows the high level dataflow of the proposed accelerator in [4]. Line buffers are used to overlap communication and computation and multiple processing elements are instantiated to allow parallel computation of the different filter channels simultaneously.

The Winograd processing element is the only part of the system that requires DSP resource. The work also deploys a parallelism-aware memory partition technique to enable efficient memory access. This results in efficiently partitioned BRAM in the FPGA fabric, minimizing data access bottlenecks while maximizing the available on-chip memory resources.The proposed Winograd transposed convolution architecture is tested on various GAN architectures such as DCGAN, Disco-GAN, Art-GAN, and GP-GAN. The design achieves a maximum average GOPS of 851.8 when using the Xilinx ZCU102 board for design implementation and testing. They report a 21$\times$ increase in DSP efficiency compared to previous work.

Another work presenting an FPGA implementation of the transposed convolution is [5]. To overcome the inefficiency of zero padding to compute the transposed convolution, they present an algorithm that is amenable to FPGA acceleration. Each input pixel is multiplied by the entire filter kernel. Where kernels overlap, the outputs are summed. Finally the excess border outside the necessary output size is removed. The method they present is shown in Fig. **??**.

The accelerator design stores all the inputs and weights in off-chip DDR. Input and output BRAM buffers are used to store data read from memory before computation and to store results before transfers to memory. The design is implemented such that arbitrary strides, kernel sizes, and padding may be used for the transposed convolution. Internal register arrays are used to store data that is used multiple times due to kernel overlaps when using the redesigned algorithm proposed.

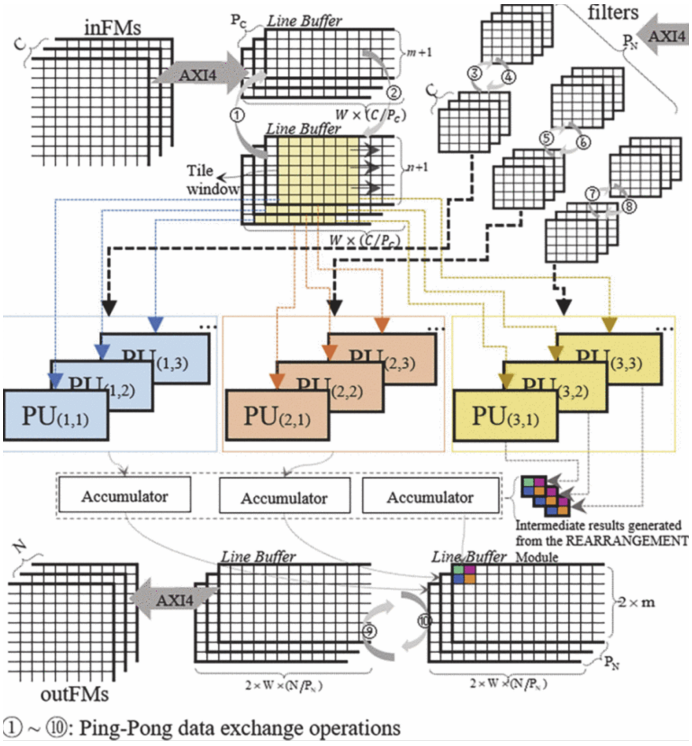

Fig. 3: High level FPGA flow for Winograd-based transposed convolution accelerator.

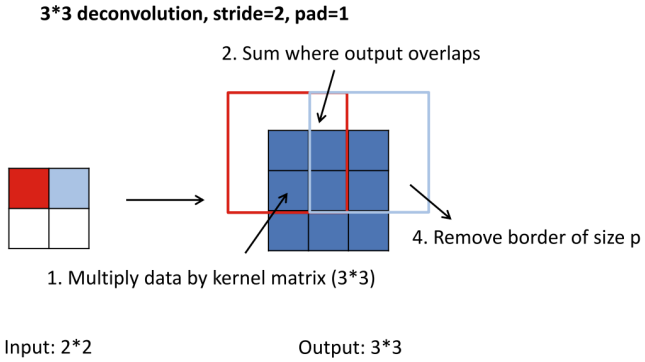

Fig. 4: Visualization of FPGA optimized transposed convolution (i.e. deconvolution).

The design achieves 90.1 GOPS on the Xilinx XC7045 using 32-bit fixed-point precision and operating at 200 MHz. This equates to a 34.7× improvement over comparable previous work [5]. It is noted that this work aimed to develop both a convolutional and transposed convolutional accelerator. When testing their full pipeline, they show that the FPGA implementation is much more energy efficient than the CPU and GPU versions, while only being slightly slower than the GPU version tested with an Nvidia Titan X.

Other transposed convolution acceleration architectures have been proposed with varying designs. Uni-OPU implements an unified architecture for convolutions, transposed convolutions, and nearest-neighbor transposed convolutions [6]. FlexiGAN implements an end-to-end acceleration architecture for GANs, including the transposed convolution kernel. The FlexiGAN results are exceeded by the performance achieved in [5]. Overall there are a variety of works in this area, but common among them is the emphasis on how the algorithm is refactored to become amenable to FPGA implementation. The acceleration techniques vary based on these varied algorithms, however, latency hiding with double buffering and maximization of DSP utilization are common strategies that

allow high GOPS performance to be achieved.

III. Methodology

This project accelerates the transposed convolution kernel by applying an iterative optimization methodology. Beginning with the naive implementation, at each stage we determine the performance bottleneck and attempt to reduce its effect, repeating until we are satisfied with kernel performance. We take advantage of common HLS optimizations such as explicit caching, DDR bus width optimization, and PE duplication/loop unrolling. We researched techniques for convolution tiling and loop ordering, and we implemented this ordering in hardware and verified its output with real performance measurements.

Initially we will implement the transposed convolution as a direct convolution using input padding. This will be our naive solution. We will then develop more efficient solutions by leveraging the sparsity of the computations and omitting multiplication by zeros. As the efficient design implementation progresses, further FPGA and HLS optimization techniques will be added to enable higher performance.

The iterative optimization process is described briefly here. The baseline implementation, v0, has no HLS optimizations and is our reference point. The first iteration, v1, implements tiling on the output channels and input height and width dimensions along with unrolling in the input channels dimension. V1 also removes the need for a sparse padded buffer and instead performs conditional computation on an unpadded input. Note that the input channel is not tiled since that would introduce the need to write and read partial output results in DRAM. The second iteration, v2, implements wide memory accesses and multiple AXI ports. Wider memory accesses improve memory bandwidth utilization. We noticed, starting in v2.1, that we had redundant reads of the bias and input due to the ordering of our loops. We reordered the read loops starting in v2.1 to remove these accesses. The v2.2 implementation adds further parallelism on top of v2.1 by unrolling in one of the kernel dimensions as well do effectively double the number of MAC units operating in parallel. Finally, our v3 implementation decouples DRAM reads and writes from the compute logic by using streams. This enables a dataflow architectue where reads to the input matrix, kernel, and bias occur at the same time as output computation and output writing. Note that there are also subversions for each of these versions mentioned where smaller optimizations were made. These main versions and the subversions are listed in table I.

TABLE I: Summary of design optimizations applied in each iteration of the design.

Version	Optimization(s)
v0.0	Baseline with padded input
v0.1	Baseline without padded input
v0.2	Tiling output channels and input height+width
v1.0	Unrolling computation in input channel dimension
v2.0	Multiple AXI ports, wide access for DDR
v2.1	Wider access for DDR, increased output channel tile size. Removed redundant bias and input reads.
v2.2	Added unrolling in kernel dimension
v3.0	Streaming data to conv kernel, dataflow, implement 512-bit access for all DDR, multiple DDR Banks

IV. Artifacts Description

This section describes how one could reproduce our results. These results are the runtime measurements and the resource utilization measurements of the kernels.

A. Software Requirements

This project uses Vitis 2019.2. The code to reproduce results is available at the github repo https://github.com/danielstumpp/transposed-conv-accel.

B. Hardware Requirements

One needs access to a Xilinx U200 FPGA card to reproduce the results of the project.

C. Experiment Workflow

The following steps outline the workflow to build and test the design. For this example it is assumed that we are building v2.0, however the steps remain the same for other versions. It is also assumed that the repository has been cloned and Vitis 2019.2 is installed and on the path of whatever machine is being used.

- Navigate to `transposed-conv-accel/hw/v2.0/`
- The source code can be found in `./src/`
- To build run `make all TARGET=hw`
- Run `./host ./artifacts/transpose_conv.xclbin` to execute the design

All relevant build artifacts will be located in the `./artifacts/` folder after a build is performed. The unneeded output from the builds can be cleared using `make cleanall`.

D. Evaluation and Expected Results

For evaluation of our transpose convolution design we select the configuration used in one layer of the UNet architecture. The layer uses a 2×2 kernel with a stride of 2 and no padding. The configuration also uses a 100×100 input with 256 channels and results in a 200×200 output size with 128 channels. When tiling is used the width and height dimensions have a tile size of 50. for versions before v2.1 the tile size for the output channels dimension is 16. For v2.1 and higher the tile size used for the output is 32 to allow full 512-bit AXI port widths.

We expected the initial implementation to be incredibly inefficient because it does not employ explicit caching. By removing the padded input in v0.1, we expected to see large improvements. V0.1 greatly reduced the amount of computation done and it reduced unnecessary BRAM usage because it takes advantage of the sparse nature of transpose convolution. We expected v0.2 to improve over v0.1 by orders of magnitude because of the data reuse employed in tiling. The improvement between v0.2 and v1.0 is easier to quantitatively predict because the difference is not in memory optimization but in PE duplication. V1.0 unrolls by a factor of 256 so we expect to the compute cycles to decrease from approximately 5 billion cycles to 20 million cycles. Between v2.0 and v2.1 we expect to see a halving of computation cycles because the unroll factor is doubled.

The latency of each iterative optimization was measured and reported in Table II. Beginning with v0 we see a very long runtime of 36 minutes. By v2.0, most of the memory optimizations were complete and we see a more reasonable runtime of 235.29 ms. With v2.1, we double the tile size in the output channel dimension and remove redundant reads of bias and input due to loop ordering. This yields a speedup of $1.68\times$. Further unrolling in v2.2 reduces runtime to 116.75 ms, which is an improvement over the 140.46 ms of v2.1 but is not a two-fold increase. The v3.0 optimization is a streaming optimization on top of the optimizations of v2.1. It demonstrates improved latency with 125.75 ms, indicating that with more time, a v3.1 optimization that implements streaming on top of the doubled unrolling of v2.2 would lead to even better performance.

TABLE II: Runtime latency results for designs built.

Version	Runtime (ms)
v0.0	2,160,000
v0.1	794,133
v0.2	66,299.4
v1.0	3,482.62
v2.0	235.29
v2.1	140.46
v2.2	116.75
v3.0	125.75

The results in Table III show the resource utilization for all of the designs built. The v0.0 and v0.1 implementations of the design show low overall resource utilization especially when considering DSPs and BRAM which are the primary resources of interest for this work. When tiling with buffering is introduced in v0.2 we see a jump in the number of BRAM required for the design. BRAM continues to increase in v1.0-v2.2 as the amount of partitioning is increased due to further unrolling. v3.0 builds off of v2.1 and only has unrolling and partitioning in the input

channel dimension which is why BRAM utilization is less than v2.2 which also partitions in one of the kernel dimensions. In designs that unroll only by the input channels (256) we see approximately 256 DSPs being used. When one of the kernel dimensions is also unrolled in v2.2 the DSPs jump to slightly over 512 as expected. Other operations such as index calculation, particularly for the accesses to the input, require some DSPs in the designs but are limited when compared to the DSPs required for the unrolled MAC operations.

TABLE III: Resource utilization results for designs built.

Version	LUT	LUTAsRAM	REG	BRAM	DSP
v0.0	3801	335	6028	50	15
v0.1	2900	234	4301	1	10
v0.2	9518	261	12599	121	71
v1.0	6125	421	10046	289	271
v2.0	8965	340	14991	311	265
v2.1	9963	494	18132	334	265
v2.2	17862	494	29250	462	521
v3.0	11803	464	24040	364	267

E. Floor Plan

The floor plan for two of the designs are shown below. Fig. 5 shows the flor plan for the v2.1 design, which unrolls with a factor of 256. Fig. 6 shows the floor plan design for the v2.2 design which doubles the unroll factor to 512 by unrolling in one of the kernel dimensions. We observe that multiple SLRs are being used in the design, which likely contributed to timing issues that were encountered when trying to scale up the unroll factor even further beyond 512.

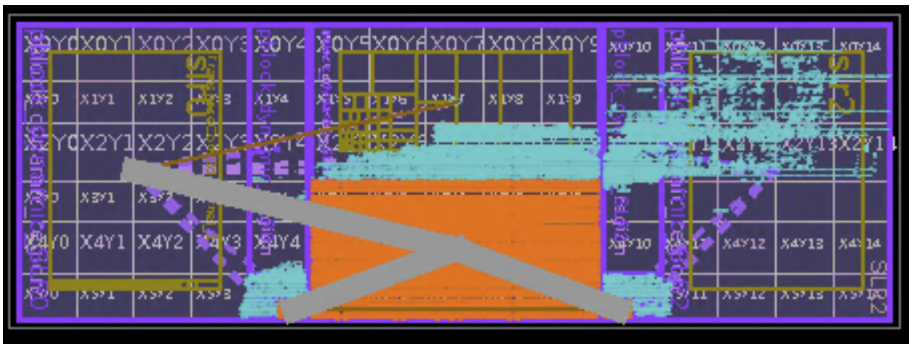

Fig. 5: FPGA floor plan for v2.1.

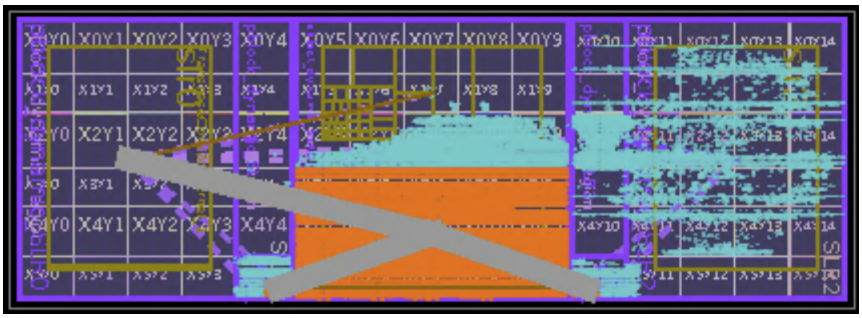

Fig. 6: FPGA floor plan for v2.2.

V. Publicity

We consent to the inclusion of this paper in the ECE2195 class project website.

REFERENCES

[1] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv preprint arXiv:1603.07285*, 2016.

[2] H. Noh, S. Hong, and B. Han, "Learning deconvolution network for semantic segmentation," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1520–1528.

[3] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[4] X. Di, H. Yang, Z. Huang, N. Mao, Y. Jia, and Y. Zheng, "Exploring resource-efficient acceleration algorithm for transposed convolution of gans on fpga," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 19–27.

[5] S. Liu, H. Fan, X. Niu, H.-c. Ng, Y. Chu, and W. Luk, "Optimizing cnn-based segmentation with deeply customized convolutional and deconvolutional architectures on fpga," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–22, 2018.

[6] Y. Yu, T. Zhao, M. Wang, K. Wang, and L. He, "¡italic¿uni-opu¡/italic¿: An fpga-based uniform accelerator for convolutional and transposed convolutional networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 7, pp. 1545–1556, 2020.