Hardware Architectures for Accelerated Computing of Generative Adversarial Networks (GANs) for SHAKTI C-Class Processor

Sai Krishna Shanmukh Bachotti (EE19B009)
Guide: Prof. Kamakoti V., Dept. of CSE, IIT Madras
Advisor from SHAKTI: Hemant Kumar (CS20S055)

Abstract of the Problem

GANs have a high compute and memory requirement. GAN computations have irregular data dependencies which lead to a high amount of bandwidth pressure. Since GANs have both Convolutional and De-convolutional layers, they do not map well to conventional neural network accelerators which are designed only for convolutions. Further, the DeConv operations involve upsampling strategies which lead to repeated multiplications and a large number of ineffective computations. Hence there is a need for customized accelerators for achieving high efficiency with GANs.

Introduction

Artificial Intelligence has pervaded into a wide range of engineering disciplines in recent years. With increasingly available data, supervised learning models are thriving to become the state-of-the-art solutions. However, training and deployment of supervised learning models is not feasible in domains where procurement of labeled data is expensive and existing data size is inadequate for reliable model training. To solve this problem, data augmentation using Generative Neural Networks has been a novel and promising approach. Generative Adversarial Networks (GANs) are one of the most popular and effective unsupervised learning models to generate synthetic realistically looking data with high fidelity which share similar features as the original data. Several GANs have been proposed in applications like image to image translation [1], image compression [2], style transfer [3] and text to image conversion [4].

The underlying working principle in most of the GAN based models is the same. It consists of a synthesis network, also called the generator, which takes a low dimensional input such as a randomly sampled noise or a latent code produced by a decoder network from a sample of real image data. The aim of this generator is to learn to generate high dimensional output such as fake images with certain attributes which vary from model to model. GANs also consist of a Discriminator network which learns to distinguish the generated fake samples from the real data. Typically the discriminators are CNN based classifiers and are used only during the training phase of the model. The model is trained until the generator is well learnt to fool the discriminator. During deployment, the generator model can function stand alone without the involvement of a discriminator.

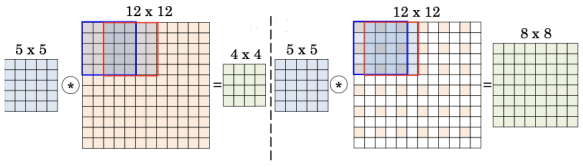

Figure 1: (Left) Representation of CONV operation, (Right) Representation of DeCONV operation

The generator network in GANs has Deconvolution (DeCONV) layers which are responsible for increasing the dimensionality of the output. Ideally, DeCONV operation has inverse functionality of that of the convolution operation. However, GANs use a mathematically closer operation called the Upconvolution also called the transposed convolution (TrCONV). Both words are used interchangeably in the rest of the paper. In Fig1, on the left, we see that a 12×12 image is convoluted with 5×5 filter resulting in a 4×4 output. Whereas the Upconvolution operation can be broken down into upsampling the image and then applying convolution. In Fig 1, on the right, we see that 6×6 image is upsampled to 12×12 by zero-insertion upsampling, resulting

Table 1: Profiling Upconvolution on SHAKTI C Class Processor on 3 different workloads

Image	Weight	Redundancy	Cycles
(16,16,3)	(16,3,3,3)	75.638%	145240
(16,16,3)	(32,3,7,7)	75.323%	903691
(32,32,3)	(32,3,7,7)	75.263%	4282445

in 8 × 8 image which is larger than the input we started with. However, the zero-insertion strategy leads to a large number of ineffective computations. With 4 × 4 input and a stride of 2, nearly 87% computations are redundant, and with 16 × 16 input and a stride of 32, 99.8% computations become redundant[5]. Table 1 characterizes the percentage of redundant multiplications in Upconvolution operation and the number of cycles of execution on SHAKTI C Class processor for three different workloads with different combinations of image size and weight size. This data is observed by compiling and executing C code of UpConvolution and run on bare-metal SHAKTI C Class processor.

The traditional convolution neural network accelerators like Eyeriss, Envision, or our home grown ShaktiMAAN focus on optimizing multiply and accumulate operations and their scheduling algorithms to support maximum parallelism. However these architectures are indifferent to the zero upsampling strategy which is the primary culprit in causing redundancy in the DeCONV layer computation time of GANs. Hence there is a need for a better architecture to solve this problem.

Accelerated Architectures for GANs

Chang et al. [6] propose a Winograd algorithm based accelerator for DeCONV operation. Here they transform the DeCONV layer to a CONV layer problem where the number of filters are increased by a multifold. Later these filters and images are transformed to the Winograd domain in which the convolution operation becomes elementwise multiplication. Di et al. [7] propose an FPGA architecture for implementing TrCONV. They observe that the CONV operation, when applied to a 5×5 filter and a 6×6 padded input, can be divided into four sub-filters, each with a different number of non-zero elements. To optimize the implementation, they reorganize these sub-filters into a 3×3 shape, enabling the transformation of one

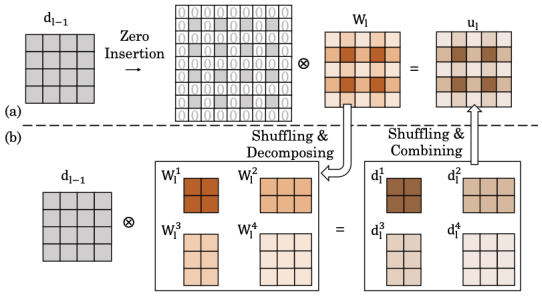

Figure 2: Decomposition based approach for DeCONV

TrCONV with a 5×5 filter into four CONV operations between 3×3 filters and suitably padded input feature map. Ineffectual operations are eliminated through this process, and this technique is referred to as decomposition. Subsequently, these four CONV operations are executed using the Winograd fast algorithm. By reducing the sub-filter size to 3×3, the implementation benefits from the application of the Winograd fast algorithm. Both the architectures utilize decomposition technqiue and Winograd based convolution.

Another effective and novel approach was proposed by Chen et al. [8] which utilizes Decomposition technique (Fig 2.). They utilize a Processing In Memory (PIM) - based Resistive RAM architecture for accelerating GAN computations. This architecture benefits from PIM as they directly perform arithmetic operations like shift and add near memory without bringing them all the way to the core or compute engine. This approach reported a 7.6× speedup over a Geforce GTX 1080 GPU and is also energy efficient. However, the ReRAM are not reliable in the long run as they last upto 3.5 years for continuous GAN workload. Jiale Yan et al.[9], proposed a reconfigurable and efficient architecture called Generative Network Accelerator (GNA) that can operate in both CONV and DeCONV acceleration modes for the same hardware. They also included an extended architecture to avoid additional off-chip memory access in residual layer computation, which are also found in some GAN based model architectures. They also provide various precision modes and bandwidth reconfiguration for processing elements and memory buffers respectively, resulting in flexible bit-width support. In this paper, we adapt some of the ideas discussed to design a DeCONV layer accelerator.

GNA: Algorithm, Architecture & Bluespec Implementation

In typical convolution accelerators, each processing element (PE) is mapped to do multiply and accumulate (MAC) operations and compute one output pixel. This is referred to as Output Oriented Mapping (OOM). When OOM is applied to Transposed Convolution operation, we can observe that PEs are load imbalanced. Refer to Fig.3, we can see that in a CONV operation, each PE has equal load of 9 multiplications. However, in DeCONV operation, we find loads of 1,2 and 4 effective multiplications per PE leading to poor hardware utilization. Hence, we use an input oriented mapping (IOM), where each PE is mapped to one input pixel. The proposed algorithm ensures that all loads are balanced and hardware utilization is maximum. To understand the algorithm, we specify some variables and parameters to characterize the problem. Let the input image be of the size (N, H, L) where N is the number of channels, H is the height and L is the length. Each weight filter is of the shape (N, Kh, Kl) and there are M such filters. Then the output is of the shape (M,R,C). We define two tiling parameters Tl and Tm. Tl is the number of values chosen from a given channel and a given row of input image for the computing engine. Tm corresponds to the number of filters chosen for the computing engine. The original work also includes a tiling parameter for the number of channels (Tn) but that results in a 3D grid of processing elements and we skip it to develop a lighter architecture. By choosing Tn = 1, we restrict our problem to a 2D grid of processing elements. Since no tiling is done along channels, we also interchange Loop N and Loop Kl (as compared to the original algorithm) and computation of Loop Kl is part of computing engine in the proposed algorithm of this paper.

```
for(h = 0; h < H; h++) Loop H
  for(m = 0; m < M; m+=Tm) Loop M
    for(kkh = 0; kkh < Kh; kh++) Loop Kh
      for(l = 0; l < L; l+= Tl) Loop L
        for(n = 0; n < N; n++) Loop N
        // Computation in Core
          for(kkl = 0; kkl < Kl; kkl++)
            r <- s*h + kkh
            c <- s*l + kkl
            wi : m to m + Tm,
            ii: l to l + Tl,
            out[wi][r][c] += inp[n][h][ii]
            *w[wi][n][kkh][kkl]
```

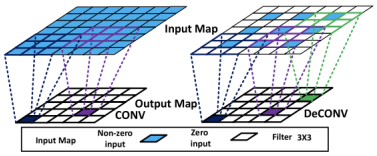

Figure 3: Load Imbalance in OOM for DeCONV

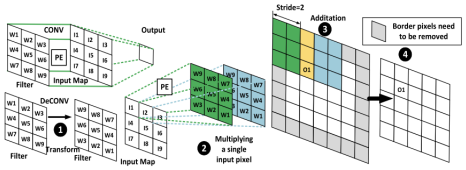

Figure 4: DeCONV using Input oriented mapping

The DeCONV algorithm is shown before. The crux of the algorithm is that each processing element (PE) multiplies an input pixel to a filter matrix. These products of different inputs are strided across the output pane and overlapped areas are summed to get the output (as shown in Fig. 4). To compute the stride, a mathematical equation for unified stride is defined as S = (convolution stride)/(1 + pad). Zero insertion up sampling strategy ensures the pad to be 1, hence S = 0.5 for convolution stride of 1. This is mathematically equivalent to striding the products of each input pixel with a stride of 2. In this way, PEs workload is balanced throughout the computation.

The architecture for this accelerator (as shown in Fig. 5) comprises an input buffer which stores the image input and on which the DeCONV is performed. The weight buffer contains the weights of M filters. A tile of weight and input are sent to the computing core where each input is multiplied to all weights of the tile. The coordinator module computes the (r,c) output coordinate based on the position (h,l) from which the image tile is picked. The output partial sums are then sent to corresponding location in the output buffer. We also maintain a cold buffer to stitch and accumulate the overlapping portions of outputs that are computed across the iterations. In the original paper,

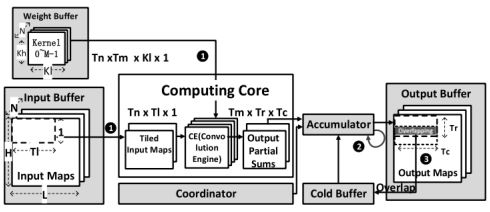

Figure 5: Architecture for GNA

```
l = Variable('L')
h = Variable('H')
n = Variable('N')
k = Variable('K')
m = Variable('M')
T0 = Variable('T0')

constraints = []
constraints.append(l*h*n <= 4992)
constraints.append(m*n*k*k <= 13824)
constraints.append(2*l + k <= T0)
constraints.append(h >= 70)
constraints.append(l >= 70)
constraints.append(n >= 1)
constraints.append(m >= 1)
constraints.append(k >= 117)

objective = T0 # Twall problem
m = Model(objective, constraints)
sol = m.solve(verbosity = 1)
Twall = sol["variables"]["T0"]
```

```
Using solver 'mosek_cli'
 for 6 free variables
  in 9 posynomial inequalities.
Solving took 0.0199 seconds.
```

Figure 6: Using Geometric Programming solver to estimate hardware constraint for Cold Buffer. Here, *Twall* solution is 256

Table 2: Paper proposed design parameters

Buffer	Size
Image Buffer	19.5KB
Weight Buffer	54KB
Output Buffer	136.5KB
(Tm, Tl)	(16,16)

the cold buffer is of (K -S) length, where K is size of kernel and S is the stride. Every image tile tl sent to CE, generates an output of size Tm × Tl × K. The Tl × K matrix is unloaded to the output buffer in steps of K and the last (K - S) values are saved in the cold buffer which are utilized in the next iteration. To directly unload the entire partial products as and when computed, we use a larger cold buffer which is of size of maximum length of the output. This design choice saves Tl cycles for each tile at the expense of additional hardware. The estimation of the hardware budget for cold buffer, given the sizes of input buffer and weight buffer can be written as posynomial constraints and is geometric programming problem. Using MOSEK (Licensed) on Google Colaboratory, we model the problem (as show in Fig. 6). Table 2 contains the hardware budget for the proposed architecture. The tiling parameters are chosen equal for a symmetric PE grid i.e., Tm = Tl = 16. A Bluespec simulation for these design choices tested positive for functional correctness when run on the same three workloads used before and the the number of simulation cycles were measured (shown in Table 3)

Table 3: Bluespec Simulation Cycles for GNA

Image	Weight	Sim. Cycles
(16,16,3)	(16,3,3,3)	915
(16,16,3)	(32,3,7,7)	7227
(32,32,3)	(32,3,7,7)	25371

Table 4: Synthesis Report for Scaled Down Version of GNA

Type	Number
Slice LUTS (Logic)	2458
Slice Registers	766
DSP48E1	54

The simulation cycles are comparatively smaller in magnitude w.r.t to Table1, as the clock frequency of the processor will be different from the hardware designed. To get an estimate of the target clock period we need to generate the Verilog code from the Bluespec design. However, this couldn't be done on my personal machine as the stack overflowed (upto 2GB allocation) for the design metrics proposed in the paper. This can be primarily attributed to the huge unfolding that happens when all register units of a buffer are initialized to 0. Also, the cold buffer needs to reset after completion of every row of image, hence adding more to the unfolding degree. The inputs are read in using RegFile package provided by Bluespec which internally utilizes readmemh from Verilog. However, this setup has a limitation on the number of read ports i.e., only 5 read ports are available and becomes a bottleneck for file to buffer transfer. Hence, a scaled down version of the model was used with tiling parameters as (4,4) and a total hardware budget of 4KB which resulted in successful Verilog generation. The generated Verilog synthesized successfully when run on Vivado for XA Artix FPGA. The mapped hardware resources are also listed in Table 4.

Future Work

The Bluespec implementation of GNA enables it to be integrated with processor after adding a few more features. The current design reads/writes the data from/to a file. A direct memory access or a protocol based memory transfer logic should be written between the main memory unit of the processor and the buffers of the GNA. Also, some particular input and weight configurations can make

the output buffer overflow without completing the entire computation. An exception handling module should also be added to stop the execution, load the warm content of the buffer to memory and resume the operation. The current hardware accelerator is designed for 32b data. Implementing precision based acceleration allows it to operate in 16b/8b mode for lighter workloads which makes it energy-efficient.

Code Repository for Bluespec implementation of GNA

References

[1] StarGAN: Unified Generative Adversarial Networks for Multi-Domain Image-to-Image Translation by Yunjey Choi et al.,2018

[2] High-Fidelity Generative Image Compression by Fabian Mentzer et al., 2020

[3] P2-GAN: Efficient Style Transfer Using Single Style Image by Zhentan Zheng et al., 2020

[4] A Style-Based Generator Architecture for Generative Adversarial Networks by Tero Karras et al. from NVIDIA, 2019

[5] A Survey of Hardware Architectures for GANs

[6] Towards design methodology of efficient fast algorithms for accelerating generative adversarial networks on FPGAs by Chang et al.,2020

[7] 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 International Conference on Field-Programmable Technology (ICFPT), 2019, pp. 19–27.

[8] ZARA: A Novel Zero-free Dataflow Accelerator for Generative Adversarial Networks in 3D ReRAM by Fan Chen et al.,2019

[9] GNA: Reconfigurable and Efficient Architecture for Generative Network Acceleration by Jiale Yan et al.,2018