Accelerated Inference Framework of Sparse Neural Network Based on Nested Bitmask Structure

Yipeng Zhang[1] , **Bo Du**[1*] , **Lefei Zhang**[1] , **Rongchun Li**[2] and **Yong Dou**[2]
[1]School of Computer Science, Wuhan University
[2]National Laboratory for Parallel and Distributed Processing, National University of Defense Technology
{zyp91, remoteking, zhanglefei}@whu.edu.cn, {rongchunli, yongdou}@nudt.edu.cn

Abstract

In order to satisfy the ever-growing demand for high-performance processors for neural networks, the state-of-the-art processing units tend to use application-oriented circuits to replace Processing Engine (PE) on the GPU under circumstances where low-power solutions are required. The application-oriented PE is fully optimized in terms of the circuit architecture and eliminates incorrect data dependency and instructional redundancy. In this paper, we propose a novel encoding approach on a sparse neural network after pruning. We partition the weight matrix into numerous blocks and use a low-rank binary map to represent the validation of these blocks. Furthermore, the elements in each nonzero block are also encoded into two submatrices: one is the binary stream discriminating the zero/nonzero position, while the other is the pure nonzero elements stored in the FIFO. In the experimental part, we implement a well pre-trained sparse neural network on the Xilinx FPGA VC707. Experimental results show that our algorithm outperforms the other benchmarks. Our approach has successfully optimized the throughput and the energy efficiency to deal with a single frame. Accordingly, we contend that Nested Bitmask Neural Network (NBNN), is an efficient neural network structure with only minor accuracy loss on the SoC system.

1 Introduction

Neural Network first appears as a biologically inspired software algorithm in the artificial intelligence [LeCun *et al.*, 1990; Krizhevsky *et al.*, 2012]. Human organs perceive stimulation from the exterior environment and transmit related information to the corresponding brain section. The neurons in the brain undertakes a role of a intermediate messaging agent connecting the sensors (Finger, Eyes) with the processor (Brains). The working mechanism of neurons involves passing along the bioelectric signal to the right section in the brain; subsequently, the brain uses the prior accumulated knowledge to predict the outcome. Computer scientists generalize this process by means of linear algebra and formulize it using a combination of various layers (convolution layer, fully connected layer, etc). This happens to produce unexpected efficacy with the layers going deeper: the structure can be easily piled up, with the accuracy escalating tremendously. The layers are designed as an independent module and make it very easy for the designer to customize their own network. The DNN has therefore become ubiquitous in different scenarios[Liu *et al.*, 2019], including object tracking, instance segmentation, and has received acknowledgements from institutes and corporations worldwide.

However, as the AI task becomes more complicated, the prevailing neural network, whether AlexNet or ResNet, is of far larger size than the initial neural network. The overwhelming size of the neural network overruns the finite resources on the CPU. The conventional CPU-compiler structures are incapable of dealing with the massive incoming amount of repetitive data. Moreover, CPUs have some fatal drawbacks compared to the GPU, the most common computing platform nowadays in the DNN:

1. **CPUs have fewer ALUs compared to the GPUs**. CPU bears the task of coordinating the whole system. Therefore, the CPU is not supposed to undertake too much arithmetic computation. The role of CPU is more involved with instruction scheduling and optimization with finite arithmetic ability. Inversely, the GPU has abundant on-board ALUs and can concurrently handle multi-channels repetitive streaming data.
2. **GPUs do not follow a stringent instruction order**. As the GPU is not sensitive to the orders of the instruction, so it takes less time to arrange the executive order. In the training process, all training samples share a joint neural network model and there is no strict executive order for the dataset. Accordingly, the training samples can be sent into GPU's shared memory and concurrently executed.
3. **GPUs exhibit supreme arithmetic ability**. The GPU is specifically oriented at a highly competitive float point calculation, so the number of float registers in GPU is greater than that for CPUs. Moreover, ALU on GPU is also targeted at the float point operation and especially optimized for float point multiplication.

We can learn a lesson from the above aspects: GPU takes the place of the CPU due to its superior processing mechanism and architecture, but is it the very architecture that most suitable

[*]Corresponding Author

to the neural network task? Many studies have shown that the GPU is also not flawlessly in every AI scenario: GPU consumes too much energy, whereas in most daily cases, we only use the pretrained inference framework to deal with the data streaming. This implies that a low power alternative solution should be found to complement GPU functionality.

In this paper, we propose a Nested Bitmask structure for use on the sparse neural network. Although there are many well pretrained sparse models provided on Github, we should not directly deploy the model on the SoC, as in the case for FPGA, the platform we are using in the experimental part. The numerous 0s in the hardware invariably occupy a bandwidth of 32 bits, and every operation instruction treats 0 as a float point number. Therefore, we should skip the operation over the 0 operand. The following list summarizes our work on the sparse neural network model:

1. We introduce the partitioned matrix technique into the encoding process. In the fully connected layer, the weight matrix is huge and exerts critical demand on the processor. In our design, we fully consider the reconfigurable mechanism and component reuse on the FPGA, then patition a single matrix into multiple uniform submatrices. These blocks can share the processing unit in a sequential pipeline structure, and fully improve the on-board utilization rate.
2. After the partioning of the weight matrix, we carefully examine the content in each block. This then gives rise to the following question: can we simply skip these zero blocks in order to save energy and access time? Here we bring in the Binary Decision concept[Liu and Tsang, 2017]: a prefixed bit is the premise of validation for the subsequent processing. If the prefixed bit is 1, the PE on the FPGA is informed that this block requires the computing and transmits the dataset into the FIFO. By contrast, if the prefixed bit is 0, the zero blocks will be directly discarded, which would save a tremendous number of time slots to be used for other blocks.
3. Moving down to the inside of the submatrice, we have found that the Bitmask format is preferable to the Relative Index format[Han *et al.*, 2015]. Relative Index is a very important sparse matrix format and has been widely adopted in many practices, but it inevitably has some scenarios for which it is unfit: the stride length should be moderate. Large stride imposes a longer bit length on the relative address, while conversely, a small stride would waste too many time slots for the inserting the 0. Accordingly, in our experiment, we replace the relative index with the bitmask to represent the nonzero element.

The remainder of the paper is organized as follows. Section 2 briefly introduces the relevant works. Section 3 illustrates the strategy we take to build the nested-bitmask neural network and includes strict mathematical derivations regarding the appropriate block size and the resource consumption estimation. In section 4, we discuss the implementation of nested structure on the FPGA, including the inner design of PE and its pipelined deployment. Section 5 summarizes the paper and outlines some prospects for sparse neural networks in the future.

2 Related Works

AI SoC[Sharma *et al.*, 2016b; Sharma *et al.*, 2016a] is aimed at building a highly coherent Neural Network(NN) oriented circuit to improve performance of NN task[Zhang *et al.*, 2016]. Giant corporations have already incorporated the AI neural network IP into their chipsets[Guo *et al.*, 2016; Han *et al.*, 2015; Han *et al.*, 2016a]. Studies of how to improve the performance of neural networks on SoC are the spotlight in the recent years. There are several trends in the neural network designed on the chip:

- **Network Compression.** The compression of the neural network[Han *et al.*, 2015] incorporates two classic and efficient approaches: Pruning and Quantilization. Network pruning, in brief, involves trimming the weight and neurons of no great importance and iteratively carrying out this operation. Lightweight neural network is of great significance for portable devices. So the trend in pursuing lightweight networks is a long lasting process.
- **Network Acceleration.** The neural network is never satiated with the current speed[Lu and Liang, 2018; Yu *et al.*, 2017; Wang *et al.*, 2017; Zhao *et al.*, 2017]. At the moment of the advent of the neural network, the research on the accelerating these networks also began[Posewsky and Ziener, 2018; Jain *et al.*, 2018; Zhang *et al.*, 2015]. The representative acceleration methods are mainly built on the FFT[Mathieu *et al.*, 2013], which fully utilizes the component reuse in the frequency domain.

Although the compression and acceleration are the highlights in today's research of Neural Network, in most cases, they are studied independently and moving on separate ways. Only a few works dabble in the research on how to simultaneously deal with those two before our work[Han *et al.*, 2016a; Lu and Liang, 2018]. Our work aims at giving considerations to both, accelerating the calculation while maintaining the high sparsity rate. In the next section, we would discuss the theoretic support behind our proposed model.

3 Mathematical Model of Nested-Bitmask NN

3.1 Network Pruning

Firstly, we dedicate some space to simply review the available knowledge regarding network pruning[Han *et al.*, 2016b; Fujii *et al.*, 2018]. Neural Network pruning is a milestone work in neural network compression. The pruning can ensure the same performance is retained while reducing much of the inessential computational workload. There are two aspects in the pruning: Edge Pruning and Neuron Pruning. The specific difference between two is shown in Figure 1.

As Figure 1 depicts, current algorithm is actually within the scope of the Edge Pruning. In the process of pruning, the weight matrix is generalized into the loss function as a regularizer and it can be formulated as follows:

$$Loss = f(w_1, w_2, ..., w_n) + \sum_{i=1}^{n} L_x(w_i) \tag{1}$$

where L_x denotes the regulations on the weight matrix; it could be either L_0, L_1, L_2 or L_F norm, but the primary objective is to obtain the sparse formation of the weight matrix. At

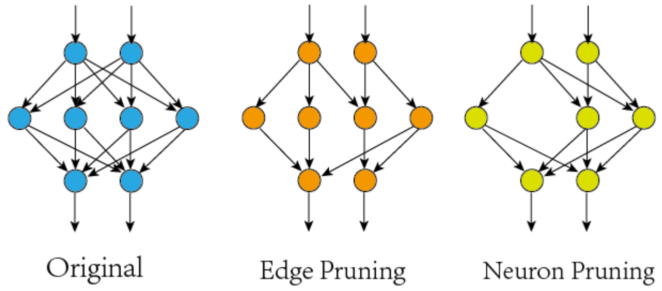

Figure 1: Comparison of pruning definition

the beginning of training, the weight matrix can be expressed via the expression $W_{i\_prior} = [w_0, w_1, w_2, w_3, ..., w_{k-1}, w_k]$, where i indicates the loop index. Presuming there are t loops in the training process, in the loop 1, let us suppose w_3 is below a certain threshold, $|w_3| < Thres$, such that its value would be clear to zero. The weight matrix can thus be rewritten into the another expression: $W_{1\_post} = [w_0, w_1, w_2, 0, ..., w_{k-1}, w_k]$. Based on the weight matrix obtained in the loop 1, the loss function in the loop 2 will be overwritten into the following formula:

$$Loss_1 = f(w_1, w_2, 0, ..., w_n) + \sum_{i=1, i\neq 3}^{n} L_x(w_i) \qquad (2)$$

With the increasing iterations of the loops, the matrix's sparse degree gradually accumulates, and only a small fraction of elements with nonzero value will contribute to the final result. However, in the edge pruning phase, disconnected values are represented by 0. But in the scope of Computer Aided Design (CAD), FPGA treats every number as float equally, and every number takes 32 bits by default if no specific format is defined. Accordingly, while the edge pruning has significant mathematical meaning and works well on the software end, if we want to achieve the accelerating function on the hardware end, we are required to tailor the shape of the weight matrix. This process is referred to as Neuron Pruning. Neuron Pruning can tailor the shape of the weight matrix and make the matrix more compact than ever. We can use two simple examples to illustrate this point: in Figure 2, the matrix on the left is the edge pruned matrix, whereas on the right is the neuron pruned matrix, where the unnecessary neurons are eradicated.

We can use Figure 2 to discuss the necessity of Neural Network Pruning. If we use the pruning approach to get a sparse weight matrix on the left, the matrix multiplication formula can be rewritten as the following:

$$Y = Relu(WX + b) = Relu(\begin{bmatrix} w_{00}x_0 + w_{01}x_1 \\ w_{10}x_0 + w_{11}x_1 \end{bmatrix}) \qquad (3)$$

where the $w_{00}x_0$ and $w_{11}x_1$ equal 0 , and do not contribute to the final result, but the hardware would treat them equally and consistently spare on-board resources for the computation. However, the neuron pruning has foreseen this situation and has tailored the shape of the weight matrix in order to skip the All-Zero Block. The neuron pruning only conserves the nonzero block; respectively in the neuron pruning, the formula is demonstrated as follows:

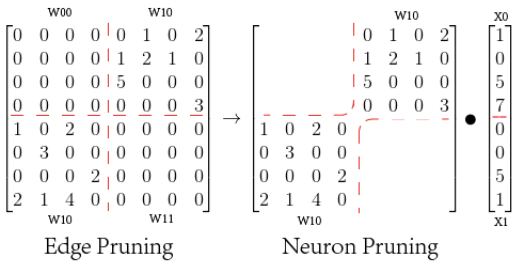

Figure 2: Comparison of the multiplications after pruning

$$Y = Relu(WX + b) = Relu(\begin{bmatrix} w_{01}x_1 \\ w_{10}x_0 \end{bmatrix}) \qquad (4)$$

Although the final outcomes of formulas 3 & 4 are equal, the computational workload of formula 4 has grossly been compressed to the half of the origin. So it fully demonstrates the necessity of deployment of the neuron pruning instead of edge pruning on the SoC. The thorough compression of the neuron pruning can not only reduce the RAM storage, but also speed up the computation in the meantime. In the next subsection, we would further explain how to partition the block and encode the neuron-pruned matrix.

3.2 Analysis of Weight Matrix Partitioning

As we have mentioned in the previous section, partitioning is the first step towards the our proposed model. But we have just encountered a problem on the size of the partitioning kernel. Here we would like to discuss in details on the mathematical derivations on the selection of right kernel size. Our approach to partition the block is not complicated, and we just follow two simple rules: I. Partitioning the matrix, II. Encoding the Nonzero Block. Let us take the example in the figure 2 to illustrate our point: the edge pruning do not change the formation of the matrix, and just leave the unnecessary weight as zero. However, at the neuron pruning stage, we have to find a efficient way to compress the neuron-pruned matrix. Here we adopt the bitmask technique to the post-partitioned operation. In the Figure 2, at the edge pruning section, the total weights storage are 256 bytes. Assuming the matrix is decomposed by the red-dashed line, the representative bitmask can be expressed as the following matrix $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, then the subsection of w_{10} and w_{01} is also encoded into the two consecutive container: Bitmask Stream Container(BSC) and Nonzero Value Container(NVC). The nonzero block w_{01} can be interpreted into two joint matrix: the flattened bitmask vector and the nonzero value container.

Firstly, let us assume a weight matrix with dimensions $M \times N$; the total float point storage is therefore $4MN$ bytes. Moreover, we should also consider the operations to complete the multiplication and accumulative add-up. We can instantiate an IP from Vivado to acquire the basic parameters required to proceed with the approximation. A simple expression can be used, such as float-point multiplication add-up logic, $d = a \times b + c$, which is an indispensable core part of the fully connected layer. In the Xilinx VC707 series, this simple combination takes five cycles to process. If there is a sequence of flattened image with length N, this process can be generalized as a linear multiplication of two matrices of sizes $M \times N$ and $N \times 1$; the FLOPs can then be calculated accordingly as $2MN + M$ for the WX part. So far, we have finished the storage and OPs analysis on the normal fully connected layer. By introducing a factor of k to control the sparse degree of the W, typically the sparse degree can be as high as 90% and even above, so the immediate storage space for the weight value drops to $E[kw] = kMN$. But there is an additional expenditure for the storage, namely the block bitmask matrix, which is strongly related to the factor k. We presume the block kernel is of size $p \times q$ and can be divided into $\frac{M}{p} \times \frac{N}{q}$ parts. Therefore the total weight storage for the sparse fully-connected layer is $E_{fc}\left\{Byte\right\} = 4(1-k)MN + \frac{MN}{8pq} + \frac{(1-k^{pq})MN}{8}$. By comparing the inequation of the $E_{fc}\left\{Byte\right\} \leq 4MN$, we can briefly estimate the kernel size pq's relation to the sparsity degree. Accordingly, the blocked neural network OPs are $\frac{MN}{pq} + (1-k^{pq})MN + 2(1-k)MN + M$, such that $\frac{MN}{pq} + (1-k^{pq})MN$ denotes the OPs for the block-level scanning and $2(1-k)MN + M$ is the computational workload for the float-point operation. Across the formula, the MN is the largest term and far greater than the other terms, and we can see that the OPs is reduced to approximately $1-k$ times the size of the original one. We can therefore see that the nested-block structure is very efficient for the sparse neural network model.

Moreover, we can carry out the latency estimation accordingly. In the branch prediction stage, we adopt the static branch prediction, which implies that the default block binary representative is 0, and it usually takes approximately 2 clock cycles to decide which way to proceed. In the mathematical derivations, we assume the delay in the branch prediction is t_{delay}, the multiplication is t_{mult} and the add-up is t_{add}. If the sparsity degree is k, which can be greater than 90%, the time interval for a single block operation could be written in the following formula:

$$E_{SGL}\left\{t\right\} = \int_{i=1}^{M}\int_{j=1}^{N}[(1-k)(2t_{delay} + t_{mult} + t_{add}) + kt_{delay}]didj \tag{5}$$

Simplifying the above, formula gives us the below:

$$E_{SGL}\left\{t\right\} = MN[(1-k)(t_{mult} + t_{add} + t_{delay}) + t_{delay}]$$

Thus, the above formula demonstrates the expectation of the time latency on the direct implementation of the single layer of bitmask on the sparse neural network. t_{delay} denotes the latency to fetch the instruction and operand, and we also have to consider the process time for the static branch prediction unit. Considering that the degree of sparsity in the model is large, we set the default path to skip as many elements as possible. When the pointer is confronted with a nonzero value, the register would be flushed to reconsider the case. We can use a schematic graph (see Figure 3) from the scope of hardware constitution to help us understand this problem.

We promote this architecture to multiple layers and can thus see that in the matrix with high sparse degree rate, the multiple layers could be more efficient in the FPGA-based calculation. It is noted that in the scope of a single block, the time latency obeys a binomial distribution. Firstly, we are looking for the probability of skipping a block. Only in the circumstance that all the elements are 0 can the computational unit skip the block. Each element has a probability of k of being zero, and inversely the probability of a nonzero element is $1-k$. Statistically, a block filled entirely with the 0 has a probability of $P\left\{skip\right\} = k^{pq}$, and respectively the probability that not all elements are equal to zero is $P\left\{no\_skip\right\} = 1 - k^{pq}$. According to the definition of binomial distribution, the expectation of the binomial distribution to happen once is shown in the following:

$$E_{BLK}\left\{t\right\} = P\left\{skip\right\}t_{skip} + P\left\{no\_skip\right\}t_{no\_skip} \tag{6}$$

As we can see from the above formula, the expectation time latency of a single block is composed of two parts, namely the expected latency to skip and not to skip the block. We would like to expand the t_{skip} and t_{no_skip} in detail. In fact, if the decoded instruction tells us that the block could be discarded during the computation, the t_{skip} could be very straightforward, in that $t_{skip} = t_{delay}$. Otherwise, the expression of t_{no_skip} could be rather complicated compared to that of t_{skip}. This is because we not only have to consider the delay during the binary decision chain, but also spend considerable time on the multiplication and accumulative add-up. The t_{no_skip} can be expressed via the following formula in detail:

$$t_{no\_stop} = pqt_{delay} + E\left\{elem! = 0\right\}(t_{mul} + t_{add} + t_{delay})$$

Noting that $E\left\{elem! = 0\right\}$ is the expectation of the number of nonzero values in a single block, it can be inferred that:

$$E\left\{elem! = 0\right\} = pqk \tag{7}$$

So the above formula can be further simplified as follows:

$$t_{no\_stop} = pqt_{delay} + pq(1-k)(t_{mul} + t_{add} + t_{delay}) \tag{8}$$

Substitute the t_{no_stop} into Formula 11, and we can obtain the final result as follows::

$$E_{BLK}\left\{t\right\} = k^{pq}t_{delay} + (1-k^{kq})[pqt_{delay} + pq(1-k)(t_{mul} + t_{add} + t_{delay})] \tag{9}$$

$$E_{MULT}\left\{t\right\} = \int_{i=1}^{\frac{M}{p}}\int_{j=1}^{\frac{N}{q}} E_{BLK}\left\{t\right\}didj \tag{10}$$

Subsequent analysis will discuss the circumstance that the multiple bitmask outperforms the single layer, and that could be expressed under the following formula:

$$E_{SGL}\left\{t\right\} \geq E_{MULT}\left\{t\right\} \tag{11}$$

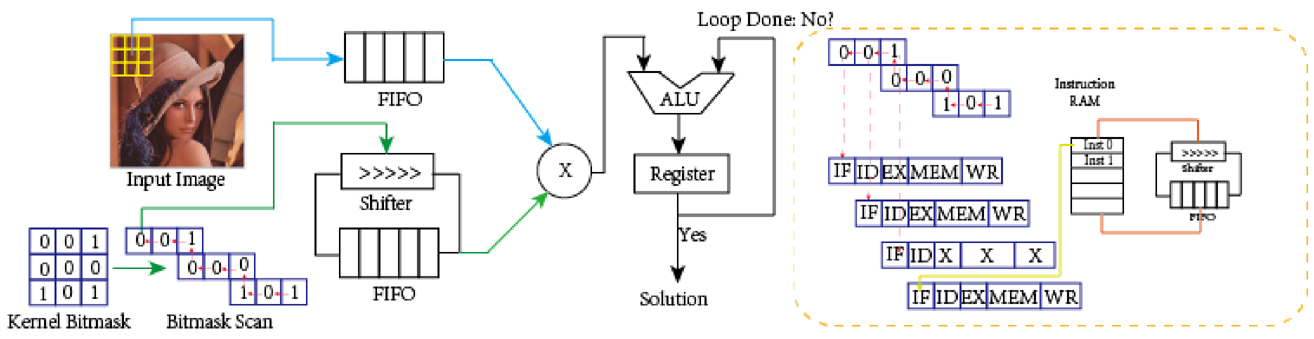

Figure 3: Sketch map of proposed approach in hardware constitution

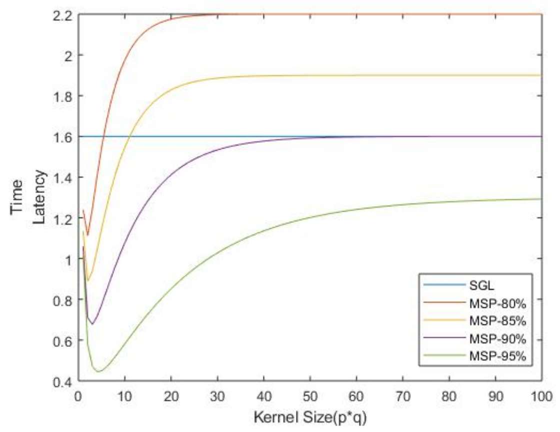

Figure 4: Kernel size vs. different sparsity degree

We are looking for an appropriate kernel size to partition the weight matrix in order to efficiently accelerate the inference process. Our first step is to set the pq as a unity variable, by controlling the pq to obtain the best partitioning outcome. In the following paragraph, we will compare the curve of the E_{SGL} and E_{MULT} in a single graph. We can observe that under different degrees of sparsity, if an appropriate kernel size is picked up, the expectation time for the multiple layers can be much less than the case for the single layer. Figure 4 depicts the curve of the time latency corresponding to the increase of kernel size. The curve of single layer of bitmask (SGL) is a flat line, underlying the baseline of the our experiment. The time latency for multiple layers of bitmask (MSP) first reaches a bottom and then rebounds to a steady extreme value, overlapping the single layer's curve. This implies only appropriate kernel can have an apparent acceleration phenomenon. Figure 4 depicts the performance of different degrees of sparsity under the multiple bitmask layers. We can notice a interesting fact that with higher sparsity rates, the nested bitmask is more likely to reduce the time latency during the operation. In Figure 4, the weight matrix with a sparsity rate of 95% has the minimal time latency. Accordingly, thanks to the advanced pruning approach enabling the deep compression of the neural network, the pruning rate is usually above 90%. Therefore it does not have to take too much time in seeking the optimum point. In our experiment, the compression rate of the first fully-connected layers could reach as much as 2%, and the second fully-connected layer is 7%. Both fully-connected layer are applicable for the nest-bitmask structure.

Subsequently, we carefully devise our experiment, however, we do not bring convolution layers into consideration at this time, since the existing pruning approach is far from perfect and the pruning rate for the convolution layer is unsatisfactory. With the emergence of deeply compressed neural network model in the forseeable future, our proposed structure will be applied on the convolutional layers in the follow-up work.

4 Experimental Part

4.1 Experimental Settings

In our proposed model, we employ a well pretrained sparse LeNet model [Zhang *et al.*, 2018] and make full use of the parallelism and reconfigurability of the FPGA. The default setting of the HLS assumes the PE number is only one. We can unroll the loops to execute iterations concurrently due to the PE numbers we have set. A single PE cell in the convolutional layer is consisted in the following way. The register fetches the data from the selected ROI (Region of Interest), and transfers it to the processing unit, then passes it along to the neuron of the next layer. The independent working mechanism of PE can concurrently deal with massive throughput. In Table 1, vanilla LeNet-5 is the normal or edge-pruned model's direct deployment on FPGA and we have pipelined the convolutional layer in the same manner on both structures[1,2]. In the Vanilla[1] structure, we have also pipelined the fully connected layer, whereas we do not apply any optimization on the fully-connected layer of Nested Bitmask[2] for two reasons: I. Hierarchical Bitmask structure has great potential for the SNN even without any optimization provided by FPGA; II. Pipeline structure is inapplicable for the architecture with comprehensive use of if-else structure[3] (see Table 2).

4.2 Experimental Results

Direct implementation of Neural Network on the FPGA is often not acceptable because in most cases that the pre-and post-optimization latency and intervals are totally different. It is observed from the table 1 that if we treat the sliding kernel as a PE, a system with multiple PEs simultaneously working can achieve an acceleration rate approximately 12.6X.

Structure	LeNet-5	LeNet-5	Vanilla LeNet-5	Vanilla[1] (Multi PEs)	Nested BitMask	Nested Bitmask[2] (Multi PEs)
Platform	Intel i7-8700	GTX 1060	Virtex-7 VX485T	Virtex-7 VX485T	Virtex-7 VX485T	Virtex-7 VX485T
Freq(Hz)	3.2GHz	1.5GHz	117.6MHz	100MHz	110.65MHz	100MHz
Clock(s)	0.31ns	0.66ns	8.49ns	10.5ns	9.04ns	10.5ns
Dynamic Power(W)	65W	120W	21.65W	21.65W	21.65W	**21.65W**
Minimum Latency(s)	10.2us	14.2us	161.01ms	12.76ms	142.24ms	**10.76ms**

Table 1: FPGA performance compared to the advanced CPU and GPU

Layer	Vanilla FC0	Vanilla FC1	Nested FC0	Nested FC1
Unpipelined Min Latency(s)	29.2ms	365us	**161us**	**25.2us**
Pipelined Min Latency(s)	2.21ms	30.2us	N/A[3]	N/A[3]

Table 2: Latency comparison

Resource	BRAM	DSP	FF	LUT
Vanilla PE_{Mult}	120	48	28630	91662
Nested PE_{Mult}	**118**	**30**	**14222**	**44075**

Table 3: Resource consumption.

We have also noticed that the Nested Bitmask with multiple PEs in CONV layer can eliminate the min latency even further than the vanilla neural network with fully pipelined structure and also consumes less in the way of on-board resources (see Table 3). We must also point out an interesting phenomenon: in the nested bitmask's fully connected layer, we do not bring in any optimization approach and the results are already better than the optimized vanilla architecture, because the Look-Up-Table contains the ingredients of both the comparitor and concurrent Process Engine(PE). Economically, the comparitor is cost-efficient compared to the PE on the pipeline structure. This is because the PE structure is multifaceted and tries to fit in different demands, including the basic addition, multiplication, and complicated function, like accumulative-addtion,etc, whereas the comparitor's structure is relatively simple, as it only compares the prefixed bit with 1/0 and decides whether or not there should be a branch jump. At this point, some would question why we are stressing the minimum latency here. Should we consider the maximum latency on our design? We should not deny the potential impact of the maximum latency on our design, because it could happen in reality. However, there is a premise for this paper that we adopt a model with a degree of sparsity above 90%, and the sparsity degree of the fully connected fc1 is in fact above 98%. So even if the range of the latency of the nested bitmask is from $10.76ms \sim 43.51ms$, where 43.51ms is the upper bound, the maximum latency. But this would only occur in the extremely rare cases, namely in those where there are too many values in the subblock needed for the computation. In a highly sparse matrix, many blocks are actually all-zero and could be discarded right away after the comparitor tells us the choice. So the nested bitmask structure is a very promising technique on the FPGA.

Although the FPGA's computational ability cannot be compared to the state-of-the-art CPU and GPU (FPGA's latency is greater than those two), it is a compatible solution for circumstances in which power requirement is low. The power requirements of the CPU and GPU are rigorous compared to the FPGA. Moreover, FPGA is usually the prototype and testing board of the ASIC design. Despite the FPGA's processing speed being constrained by its working frequency, or more precisely, the crystal oscillator, the Xilinx HLS provides us a powerful C programming tool that enables us to quickly develop a customized circuit to satisfy our need and the actual designed ASIC could be better than the results obtained on the FPGA, not to mention that our model is actually not an example of extreme performance; the BRAM_18K, DSP48E, FF and LUT are not exploited to 100% capacity. The advanced technology could radically improve the performance obtained via our algorithm.

5 Conclusion

In this paper, we have proposed a hierarchical bitmask structure for the placement of the sparse neural network on the SoC (FPGA). We have proven it that there is a strong relationship between the sparsity degree and the partitioning kernel size in terms of the probability theory, and we also compare the latency curves of single layer of bitmask vs. double layers, which demonstrates the validity of our proposed model. The experimental part has also emerged as a solid evidence to back up our theory from the quantitative analysis, which suggests that the nested bitmask structure is a very promising technique for the highly sparse neural network.

Acknowledgements

This work was supported in part by the National Natural Science Foundation of China under Grant 61822113 and Grant 61771349, alongside with National Key R & D Program of China under Grant 2018YFA0605501 and Natural Science Foundation of Hubei Province under Grant 2018CFA050 and 2018CFB432. Without the provided support, we would have been unable to proceed with our experiment and make this work known to the public.

References

[Fujii *et al.*, 2018] Tomoya Fujii, Shimpei Sato, and Hiroki Nakahara. A threshold neuron pruning for a binarized deep neural network on an fpga. *IEICE Transactions on Information and Systems*, 101(2):376–386, 2018.

[Guo *et al.*, 2016] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Song Yao, Song Han, Yu Wang, and Huazhong Yang. From model to fpga: Software-hardware co-design for efficient neural network acceleration. In *Hot Chips 28 Symposium (HCS), 2016 IEEE*, pages 1–27. IEEE, 2016.

[Han *et al.*, 2015] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[Han *et al.*, 2016a] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 243–254. IEEE, 2016.

[Han *et al.*, 2016b] Song Han, Jeff Pool, Sharan Narang, Huizi Mao, Enhao Gong, Shijian Tang, Erich Elsen, Peter Vajda, Manohar Paluri, John Tran, et al. Dsd: Dense-sparse-dense training for deep neural networks. *arXiv preprint arXiv:1607.04381*, 2016.

[Jain *et al.*, 2018] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 776–789. IEEE, 2018.

[Krizhevsky *et al.*, 2012] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[LeCun *et al.*, 1990] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, and Lawrence D Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pages 396–404, 1990.

[Liu and Tsang, 2017] Weiwei Liu and Ivor W. Tsang. Making decision trees feasible in ultrahigh feature and label dimensions. *Journal of Machine Learning Research*, 18:81:1–81:36, 2017.

[Liu *et al.*, 2019] Weiwei Liu, Donna Xu, Ivor W. Tsang, and Wenjie Zhang. Metric learning for multi-output tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 41(2):408–422, 2019.

[Lu and Liang, 2018] Liqiang Lu and Yun Liang. Spwa: an efficient sparse winograd convolutional neural networks accelerator on fpgas. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[Mathieu *et al.*, 2013] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.

[Posewsky and Ziener, 2018] Thorbjörn Posewsky and Daniel Ziener. Throughput optimizations for fpga-based deep neural network inference. *Microprocessors and Microsystems*, 60:151–161, 2018.

[Sharma *et al.*, 2016a] Hardik Sharma, Jongse Park, Emmanuel Amaro, Bradley Thwaites, Praneetha Kotha, Anmol Gupta, Joon Kyung Kim, Asit Mishra, and Hadi Esmaeilzadeh. Dnnweaver: From high-level deep network models to fpga acceleration. In *the Workshop on Cognitive Architectures*, 2016.

[Sharma *et al.*, 2016b] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 17. IEEE Press, 2016.

[Wang *et al.*, 2017] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. Dlau: A scalable deep learning accelerator unit on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):513–517, 2017.

[Yu *et al.*, 2017] Jincheng Yu, Yiming Hu, Xuefei Ning, Jiantao Qiu, Kaiyuan Guo, Yu Wang, and Huazhong Yang. Instruction driven cross-layer cnn accelerator with winograd transformation on fpga. In *Field Programmable Technology (ICFPT), 2017 International Conference on*, pages 227–230. IEEE, 2017.

[Zhang *et al.*, 2015] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[Zhang *et al.*, 2016] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 20. IEEE Press, 2016.

[Zhang *et al.*, 2018] Tianyun Zhang, Shaokai Ye, Yipeng Zhang, Yanzhi Wang, and Makan Fardad. Systematic weight pruning of dnns using alternating direction method of multipliers. *arXiv preprint arXiv:1802.05747*, 2018.

[Zhao *et al.*, 2017] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 15–24. ACM, 2017.