Arabian Journal for Science and Engineering
https://doi.org/10.1007/s13369-021-05455-4

RESEARCH ARTICLE-ELECTRICAL ENGINEERING

Hardware Architecture Exploration for Deep Neural Networks

Wenqi Zheng[1] · Yangyi Zhao[1] · Yunfan Chen[1] · Jinhong Park[2] · Hyunchul Shin[1]

Received: 11 November 2020 / Accepted: 8 February 2021

Abstract

Owing to good performance, deep Convolution Neural Networks (CNNs) are rapidly rising in popularity across a broad range of applications. Since high accuracy CNNs are both computation intensive and memory intensive, many researchers have shown significant interest in the accelerator design. Furthermore, the AI chip market size grows and the competition on the performance, cost, and power consumption of the artificial intelligence SoC designs is increasing. Therefore, it is important to develop design techniques and platforms that are useful for the efficient design of optimized AI architectures to satisfy the given specifications in a short design time. In this research, we have developed design space exploration techniques and environments for the optimal design of the overall system including computing modules and memories. Our current design platform is built using NVIDIA Deep Learning Accelerator as a computing model, SRAM as a buffer, and DRAM with GDDR6 as an off-chip memory. We also developed a program to estimate the processing time of a given neural network. By modifying both the on-chip SRAM size and the computing module size, a designer can explore the design space efficiently, and then choose the optimal architecture which shows the minimal cost while satisfying the performance specification. To illustrate the operation of the design platform, two well-known deep CNNs are used, which are YOLOv3 and faster RCNN. This technology can be used to explore and to optimize the hardware architectures of the CNNs so that the cost can be minimized.

Keywords AI architecture · Neural network architecture · CNN · Design space exploration

1 Introduction

Machine learning techniques are pervasive, and in particular deep Convolution Neural Networks (CNNs) have demonstrated to be useful to solve challenging problems, such as object classification [15, 16], pedestrian detection [17, 18], and image enhancement [19, 20]. To achieve high accuracy, the complexity of the deep CNNs and the energy consumption of the CNNs are increasing. In recent years, many research teams proposed competitive architectures to cope with the high complexity of deep CNNs. For instance, Google has proposed TPU [3], and Cambricon has launched the DIANNAO series of accelerators [4–9]. In [21, 22], optimal design space exploration based on FPGA is explored to accelerate the CNN training. However, these works only focus on designing effective CNN computation or minimizing the memory access between off-chip memory and on-chip computing modules for CNN computation [23–25]. In this research, we try to explore the overall system design space including memories (DRAM and SRAM) as well as computing modules. A designer can efficiently explore the design space by changing the size of SRAM buffer as well as the size of the computing modules.

The size of the on-chip SRAM and the capacity of the computing modules are two main parameters to make trade-off between the neural network performance and resource cost. For efficient design space exploration, it is important to estimate the performance of the given neural network in

✉ Hyunchul Shin
shin@hanyang.ac.kr

Wenqi Zheng
zhengwenqi@hanyang.ac.kr

Yangyi Zhao
zyangyi@hanyang.ac.kr

Yunfan Chen
chenyunfan@hanyang.ac.kr

Jinhong Park
jhp.park@samsung.com

[1] Department of Electrical Engineering, Hanyang University, Ansan, Republic of Korea

[2] Samsung Electronics Inc, Suwon-si, Republic of Korea

Published online: 12 March 2021

various configurations, because a designer needs to choose the optimal configuration from many possible combinations. The objective of this research is to develop system design techniques and environments for the optimal design of a hardware architecture, including memories and computing modules, for deep CNNs. In this work, we have developed a program to estimate the performance of neural networks in a specific architecture. This technique helps to effectively explore the design space to reduce the chip area.

The main contributions of this work are summarized as follows.

1. We suggest an efficient platform for design space exploration for deep CNNs. As the applications of deep learning or deep CNNs get popular, this platform will be very useful to design an optimal architecture for a given deep CNN.
2. Since two critical parts for deep CNN processing are data transfer and computation, the platform supports data transfers from/to memories and computations.
3. Efficient performance estimation techniques are developed for capitalizing pipelining and parallel processing.
4. Extension to other memory modules and computing modules are possible. These techniques can help reducing the design cost and design time (time to market).

The rest of this paper is organized as follows. In Sect. 2, we introduce the overall platform to explore the design space. Section 3 presents the hardware architecture, including computing modules and memory modules. Section 4 describes how to estimate the performance. Section 5 presents the experimental results, illustrating how design tradeoff can be achieved. Finally, conclusions are given in Sect. 6.

2 Platform for Design Space Exploration

The chip area and performance are dependent on the on-chip SRAM size and the computing module size. It is important to explore the design space and to choose an optimal combination of sizes.

We designed a platform to explore the design space, as shown in Fig. 1. In the platform, two design parameters are the SRAM size and the computing module size. To process a neural network, the data necessary should be transferred and the required computation should be executed. The amount of data to be transferred is the sum of the neural network input feature map size and the weight parameter size for each layer. We estimate the performance of an architecture for a deep CNN by estimating the data transfer time and computation time considering pipelining. The performance estimation method is explained in Sect. 6. Our platform can efficiently explore design space for various configurations.

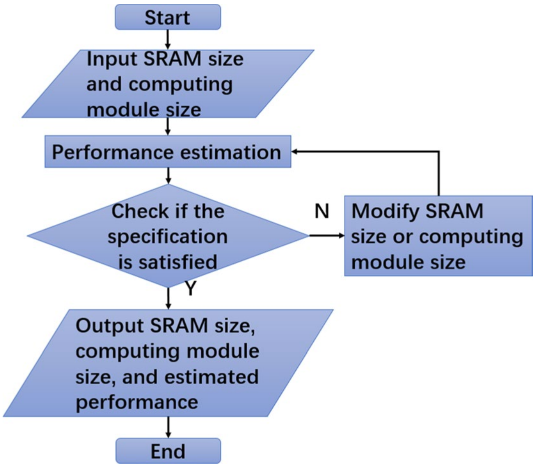

Start
Input SRAM size and computing module size
Performance estimation
Check if the specification is satisfied
N
Modify SRAM size or computing module size
Y
Output SRAM size, computing module size, and estimated performance
End

Fig. 1 Architecture design exploration platform for deep neural networks

For given SRAM size and computing module size, the NN performance is estimated. Then, the designer check whether the performance satisfies the specification. The designer may vary the SRAM size and computing module size until a satisfactory optimized design is obtained.

To illustrate the usefulness of the proposed platform, we explored the design of two well-known neural networks, as examples of two object detection methods. The object detection framework can be divided into two categories. One is a two-stage framework, which first performs regional recommendation and then performs object classification; the other one is a one-stage end-to-end framework that uses a network to do everything and outputs the results in one stage. A typical two-stage method is faster R-CNN [12], and a well-known one-stage method is You Only Look Once (YOLO) [13, 14]. We used these two CNN examples to evaluate our design platform.

3 Overall Hardware Architecture

In this section, we present the overview of our hardware architecture for deep neural networks and explain each block including memory modules and computing modules in detail. We also present the transmission rules between modules. During data transmission and computation, pipeline methods and parallel hardware structures are capitalized to reduce the execution time.

3.1 Overview of the Hardware Architecture

Since general-purpose CPUs can be a limiter for modern CNNs due to the lack of computational parallelism [10], a specific hardware accelerator is frequently needed to

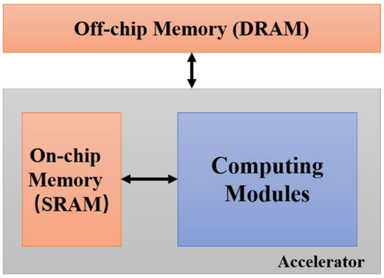


Fig. 2 Generic view of an accelerator

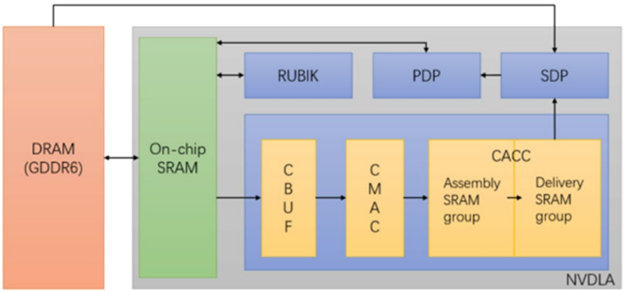


Fig. 3 Hardware Architecture using NVDLA computing modules and a GDDR6 based DRAM

capitalize parallelism and pipelining for many applications. Figure 2 shows a typical architecture for deep CNN computation, which consists of off-chip memory, on-chip buffer memory, and a set of computing modules.

Currently, our hardware architecture is built based on Nvidia Deep Learning Accelerator [2] with an on-chip SRAM buffer and a GDDR6 DRAM as an off-chip memory. In general, the NVDLA modules can be replaced by other modules such as TPUs [3] or DIANNAO series [4–9], and DRAM with GDDR6 can be replaced by a High Bandwidth Memory (HBM) [11] for wider data bandwidth at higher cost.

The detailed block diagram of the hardware structure is shown in Fig. 3. We use NVDLA as the computation model because it can support multiple commonly required computation modes, including convolution, pooling operation, activation, and normalization. The GDDR6 [1] DRAM as off-chip memory provides low-cost solution when compared to the HBM, and can be used for many applications requiring "low" and "mid" level performance. In Fig. 3, the RUBIK module is used to reshape layers in faster RCNN, the PDP is used for pooling layers, and the SDP is used for activation functions of a neural network.

3.2 Computing Module NVDLA

In NVDLA, a Convolution Buffer (CBUF), a Convolution MAC (CMAC), and a Convolution Accumulator (CACC) are main components used for convolution operations. The CBUF caches the input feature data and the weight data for the next stage computation. The CMAC receives the input feature data and the weight data and operates multiplications and additions. The results from the CMAC are called partial sums. The partial sums are accumulated by the CACC. The Planar Data Processor (PDP) for pooling layer can support MAX, MIN, and MEAN pooling methods. The Single Point Data Processor (SDP) is used for activation and batch normalization. The RUBIK for reshape layers can split or merge data. An atomic operation is a base step for direct convolution in NVDLA. Atomic operations can work when the channel width is less than or equals to 128. When the input feature map channel is more than 128, the channel operation is needed. A channel operation includes multiple block operations, and the number of block operation $= \text{ceiling}\left[\frac{\#\text{ input feature map channel}}{128}\right]$. In other cases, the channel operation is not required and the number of block operation is 1.

(1) CBUF

The CBUF has separate bandwidths for the input feature data and the weight data. Thus, the input feature data and the weight data can be transferred simultaneously. The CBUF can read 128 Bytes of feature data and 64 Bytes of weight data or write 128 Bytes of feature data and 128 Bytes of weight data per clock cycle. When we calculate the transmission time, we take the longer of the feature data transmission time and the weight transmission time and ignore the shorter one.

(2) CMAC

The CMAC consists of multiple MAC cells. The initial configuration is 16 MAC cells. To improve efficiency, we can increase the number of MAC cells to 32. Each MAC cell contains 128 multipliers for INT 8. When the input feature map channel is less than or equal to 64, one MAC cell can be divided into two computation modules. When the input feature map channel is bigger than 64, one MAC cell constitutes one computation module. Each atomic operation spends 7 clock cycles.

(3) CACC

The main components in CACC include an adder array, an assembly SRAM group (ABUF), a delivery SRAM group (DBUF), and a truncating array. One element is 1*1*channel data and the channel width of one element is less than or equal to 128. The bandwidth from CMAC to ABUF is 16 elements. The bandwidth from ABUF to truncating array is 8 elements. The bandwidth from truncating array to DBUF is 16 elements. The bandwidth from DBUF to SDP is 4 elements. Based on these bandwidths, one can calculate the transmission time in CACC.

(4) SDP

The bandwidth from SDP to SRAM is 16 elements. The SDP receives 4 elements of data from the DBUF each clock cycle and transfers 16 elements of output to SRAM each 4 clock cycles.

3.3 Memory Module

Memory modules include an off-chip DRAM and an on-chip SRAM. GDDR6 8 GB × 16 mode can be used to transfer data efficiently between the DRAM and the SRAM. GDDR6 × 16 mode has burst length of 16. First, we calculate how many rows the data occupies with the GDDR6.

$$\#\text{row} = \text{ceiling}\left[\frac{\text{input size (bit)} + \text{weight size (bit)}}{\#\text{bit per cell} \times \#\text{ cell per row}}\right]$$

where the number of rows (#*row*) represents the number of rows the data occupies with the GDDR6, input size and weight size represent the input feature map size in bits and weight size in bits, respectively. The number of bits (#*bit*) per cell represents the size of bits per cell in GDDR6 DRAM. The number of cells (#*cell*) per row represents the number of cells in one row in GDDR6 DRAM.

Before READ or WRITE command, each row is activated. The activation command takes t_{RCD}. A READ command or a WRITE command takes t_{CCD} from one column to next neighboring column. After a READ command, a pre-charge command is issued and it spends t_{RTP}. To execute a WRITE command, the first valid data must be available at the input latch after the write latency (t_{wl}). The t_{DQ} is the WRITE completion flag. After a WRITE command, a pre-charge can also be issued after t_{WR}. To transfer the next row, the DRAM needs to be activated again, spending t_{RP}. Therefore, we summarize the following two equations to calculate the number of clock cycles to read or to write one row.

READ time for one row

$$= t_{RCD} + (n - 1)t_{CCD} + t_{RTP} + t_{RP}$$

WRITE time for one row

$$= t_{RCD} + (n - 1)t_{CCD} + t_{WL} + t_{DQ} + t_{WR} + t_{RP}$$

where n is the number of cells in one row. GDDR6 DRAM has different number of cells in different configurations. For GDDR6 8 GB × 16 mode as an example, n = 2^6 = 64.

READ time per layer

$$= \#\text{row} * \text{READ time for one row}$$

WRITE time per layer

$$= \#\text{row} * \text{WRITE time for one row}$$

Initialization time for GDDR6 is 700us and it is refreshed each 1.9 us under any commands. However, different extra clock cycles should be added for different commands, as shown in Table 1.

In addition to DRAM, we use an on-chip SRAM buffer in our hardware architecture. The SRAM is initially turned on by two single read or two single write commands, and then can operate burst read or burst write commands. A single read or write command can read or write 64 Bytes in two clock cycles, and a burst read or write command can read or write 4*64 Bytes in five clock cycles. When the size of data is less than 5*64 Bytes, the SRAM won't operate the burst command. When the rest data is less than 4*64 Bytes, SRAM stops the burst command and operates single read or write commands again.

3.4 Methods to Reduce the Processing Time

In deep neural network computation, there are many data and weight parameters to be transferred between modules. To avoid conflict, we first determine the transmission rules. The overview of data transmission for NN computation is shown in Fig. 4.

Table 1 GDDR6 Parameters used in the experiments

Timing Parameter	Command Sequence
t_{RCD}	Activation to READ
t_{CCD}	READ to READ
t_{RTP}	READ to PRE-CHARGE
t_{RP}	PRE-CHARGE to Next Activation
t_{WL}	2 clock cycles
t_{DQ}	2 clock cycles
t_{WR}	2 clock cycles
$t_{whenrefreshinActivation}$	256 clock cycles
$t_{when\ refresh\ in\ READ/WRITE}$	180 clock cycles
$t_{when\ refresh\ in\ PRECHARGE}$	204 clock cycles

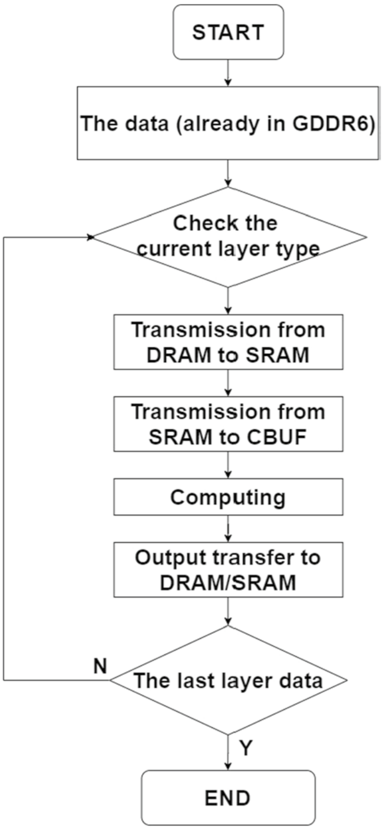

START
The data (already in GDDR6)
Check the current layer type
Transmission from DRAM to SRAM
Transmission from SRAM to CBUF
Computing
Output transfer to DRAM/SRAM
N
The last layer data
Y
END

Fig. 4 Flow of data transmission

The Algorithm 1 finishes when the output data of the last layer is transferred to the DRAM. Then the neural network performance estimation can be finished, since all the processing times are known.

For each memory modules, we have to check two conditions before transmission. First, we check whether the memory module has available space for the next data. Second, we check whether the memory module is not busy (no reading or writing). When the memory module meets both of the above two conditions, the transfer begins. Otherwise, the transfer is postponed until the conditions are met.

(Figure) After transmission, we check whether all the data.

of the layer have been transferred to the memory module. If all the data have been transferred, the transfer is complete. When the data is in the SRAM buffer, we check whether the computing module is processing previous data. After the computing module finishes the processing, new data is transferred for the next processing. If the summation of input feature size, weight size, and output feature data size is bigger than the SRAM size, the results need to be sent to the DRAM to prevent overflow. In other cases, the results are stored in the SRAM to be used as the input of the next layer.

Now we describe how to reduce the processing time for a given neural network by optimizing the hardware architecture. In computing modules, CACC can start processing as soon as the CMAC module completes computing. While the CACC module is working, the CMAC module can continue to process new data from CBUF because the results of CMAC are stored in the CACC module. Based on the above observation, we devised a pipelining technique to reduce the inference time of a neural network.

The pipelining technique is applied for computing processes from CBUF to SDP. When the partial sums are sent to ABUF, CMAC is available for the next computing. Therefore, the next data can be processed in CMAC while the previous partial sums are sent to ABUF.

The number of CMAC cells is the same as the number of partial sums in the CMAC. When the number of CMAC cells is 16, the number of partial sums is also 16. If the number of CMAC cells is increased to 32, the number of partial sums is also increased to 32. In these two cases, let k be the number of processing operations required for the data stored in CBUF. The value of k is determined by weight size, input data size, and stride. Based on the current parameters, the CBUF transfers the input feature map data and the weights to the CMAC, taking 2 clock cycles. The CMAC processes the data and 16 partial sums in 7 clock cycles. The 16 partial sums are stored in the ABUF, taking 10 clock cycles. The partial sums are accumulated in the ABUF and become the 16 accumulative sums. The 16 accumulative sums are transferred from the ABUF to the truncating array in 2 clock cycles and then transferred from the truncating array to the DBUF in 1 clock cycle. From the DBUF to the SDP, the transfer takes 4 clock cycles. After activation in the SDP, the 16 accumulative sums become 16 elements of the output feature maps. Finally, the 16 elements of the output feature maps are stored in the SRAM in 1 clock cycle. Figure 5 shows the first case, i.e., there are 16 partial sums. Each row shows the formation process of 16 elements of the output feature maps. Since there are k processing operations in Fig. 5, the total number of the output elements is $16*k$. We used three-stage pipelining with a period of ten clock cycles, as shown in Fig. 5. Using the pipelining technique, the number of the total clock cycles is $10\,k+17$.

Figure 6 corresponds to the second case, i.e., there are 32 partial sums. Because the bandwidth from CMAC to ABUF is fixed, the 32 partial sums need to be transferred as two groups. Each pair of two rows shows one processing of 32 elements of output feature maps. Since there are k processing operations in Fig. 6, the total number of the output elements is $32*k$. In this case, two rows of processing are required to compute 32 partial sums. Using four-stage pipelining with a period of ten clock cycles, the number of total clock cycles to finish the computation shown in Fig. 6 is $20\,k+17$.

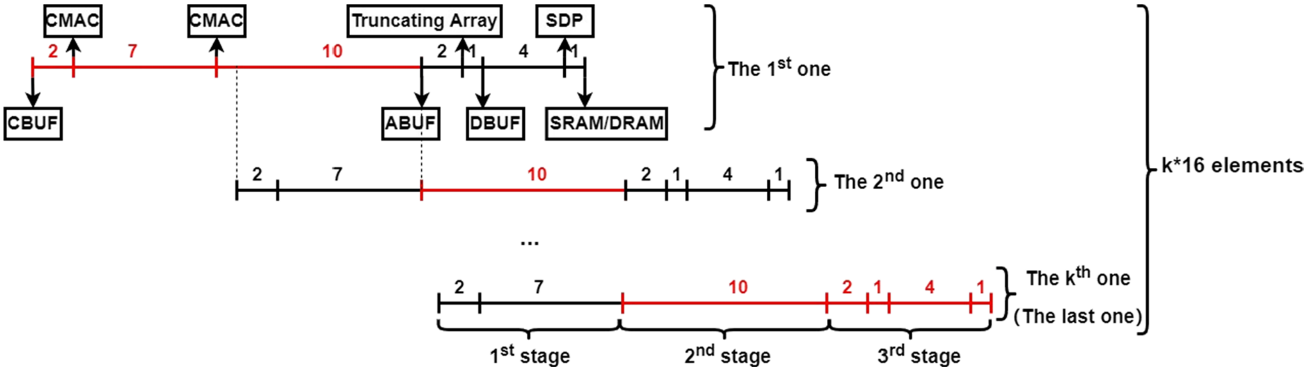


Fig. 5 Data transfer timing diagram (16 MAC cells)

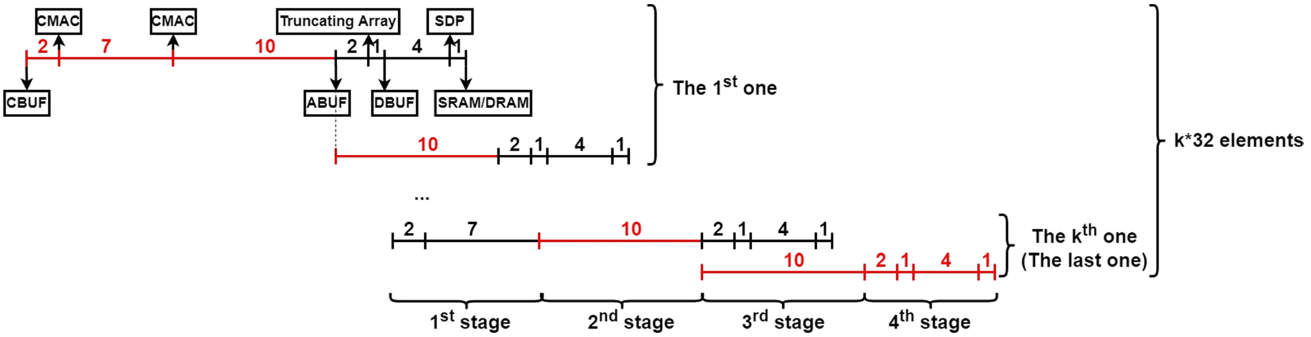


Fig. 6 Data transfer timing diagram (32 MAC cells)

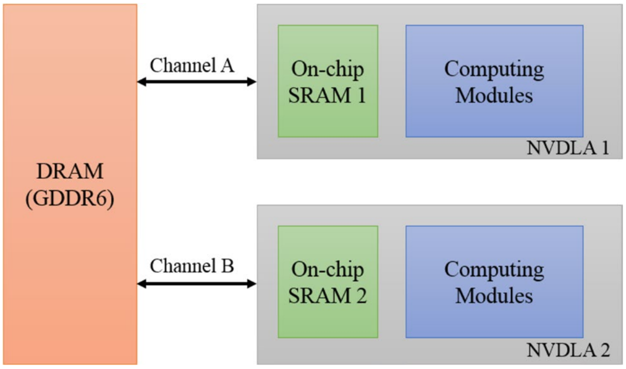


Fig. 7 Parallel accelerator hardware structure

To reduce the processing time further, parallel processing can be adopted. The parallel processing uses a greater number of hardware blocks to reduce the processing time. Since GDDR6 has 2 channels to read or write, two parallel accelerator architectures can be connected as shown in Fig. 7. The two accelerators are connected with each channel of a GDDR6 DRAM. On the one hand, the channel A is connected with the GDDR6 DRAM and NVDLA1. On the other hand, the channel B is connected with the GDDR6 DRAM and NVDLA2. The two accelerators are able to process two images, simultaneously. Because the two accelerators use different DRAM channels, there is no conflict between the two image processing paths. Therefore, significant speedup can be achieved. However, the speedup by using two parallel hardware modules is less than twice, since the two DRAM channels are utilized even for one set of SRAM and computing modules.

4 Processing Time Estimation

We have developed an architecture exploration platform to efficiently find an optimal architecture for a given neural network to satisfy the required performance. To effectively explore the design space, the performance of a specific architecture should be estimated in a short time.

When a neural network is given, the size of the feature map and the size of the weight are determined for each layer. Then, the on-chip SRAM size and the MAC cell size are given by the user. The performance of the given neural network is estimated by using the estimation algorithm, shown in Fig. 8. Table 2 lists important parameters used for estimation.

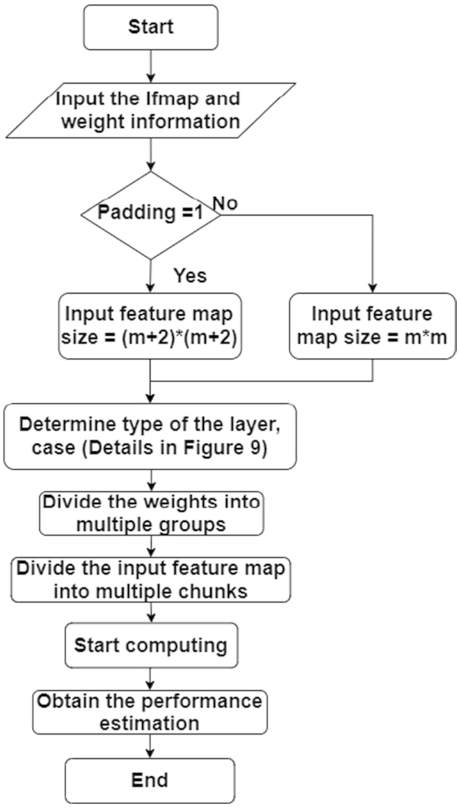

Start
Input the Ifmap and weight information
Padding =1
No
Yes
Input feature map size = (m+2)*(m+2)
Input feature map size = m*m
Determine type of the layer, case (Details in Figure 9)
Divide the weights into multiple groups
Divide the input feature map into multiple chunks
Start computing
Obtain the performance estimation
End

Fig. 8 Flow diagram to compute the execution time for each layer of NN

Table 2 Parameters for performance estimation

Parameters	Definitions
m	The width or height of input feature map
C	The number of the input feature map channels
W	The number of the weights in the layer
M	The number of MAC cells
AS_h	The width or height of accumulative sum
P	The accumulative sum channel

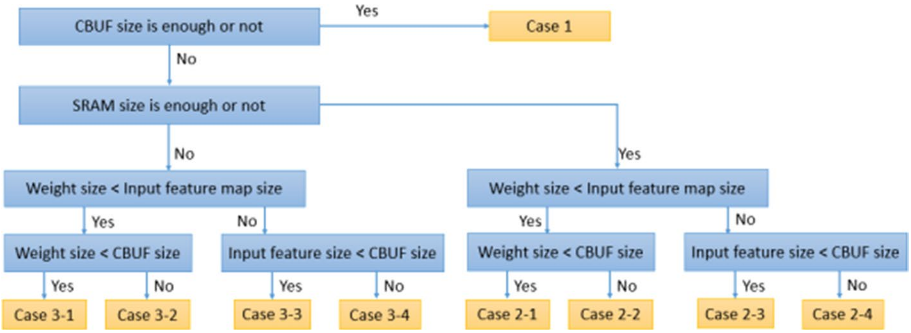
Let $m =$ input_width = input_height, then the initial feature map size is $m*m*$channel. First, we need to determine if the input image needs padding. When there is no padding (padding = 0), the input picture size is $m*m*$channel. When one pixel padding is used (padding = 1), the input picture size is $(m+2) * (m+2) *$ channel.

Second, we decide the type of the layer, to compute the number of block operations and the number of weight groups. The number of block operations is determined from the number of input channels, the number of weight groups is determined from the weight size, and the number of chunks is determined from the input size (width, height). The weight size and the input feature map size of the current layer are used to determine the case type for performance estimation, as shown in Fig. 9. The performance (speed) estimation is performed for each case in the figure. This estimation is rather straight forward and thus it is not described in detail.

Third, we determine the number of block operations and the number of weight groups as shown in Fig. 10. Let C be the number of the input feature map channels, W be the number of the weights in a layer, and M be the number of MAC cells. Since one MAC cell can hold one weight ($C > 64$) or two weights ($C < = 64$), when $W > M$ ($C > 64$) or $W > 2$ M ($C < = 64$), we have to divide the weights into multiple groups. A weight group has a set of multiple weights to be processed in the CMAC, simultaneously. When C is bigger than 128, multiple block operations are necessary. We need to calculate the number of block operations and the number of weight groups. When C is less than or equal to 128, the number of block operation is 1. When C is bigger than 128, the number of block operations is **ceiling**$[C/128]$. Then, we determine the number of weight groups. When C is less than or equal to 64, one MAC cell can be divided into two computation units. In other words, one MAC cell can process two weights, i.e., M MAC cells can process up to 2 M.

weights at once, to reduce processing time. When W is bigger than 2 M, the weights need to be divided into several weight groups to be processed group by group. The number of weight groups is **ceiling**$[W/2M]$. Each group can have up to 2 M weights. When W is smaller than or equal to 2 M,

Fig. 9 Classification of case type of each layer


CBUF size is enough or not
Yes
Case 1
No
SRAM size is enough or not
No
Yes
Weight size < Input feature map size
Yes
No
Weight size < CBUF size
Yes
No
Case 3-1
Case 3-2
Input feature size < CBUF size
Yes
No
Case 3-3
Case 3-4
Weight size < Input feature map size
Yes
No
Weight size < CBUF size
Yes
No
Case 2-1
Case 2-2
Input feature size < CBUF size
Yes
No
Case 2-3
Case 2-4

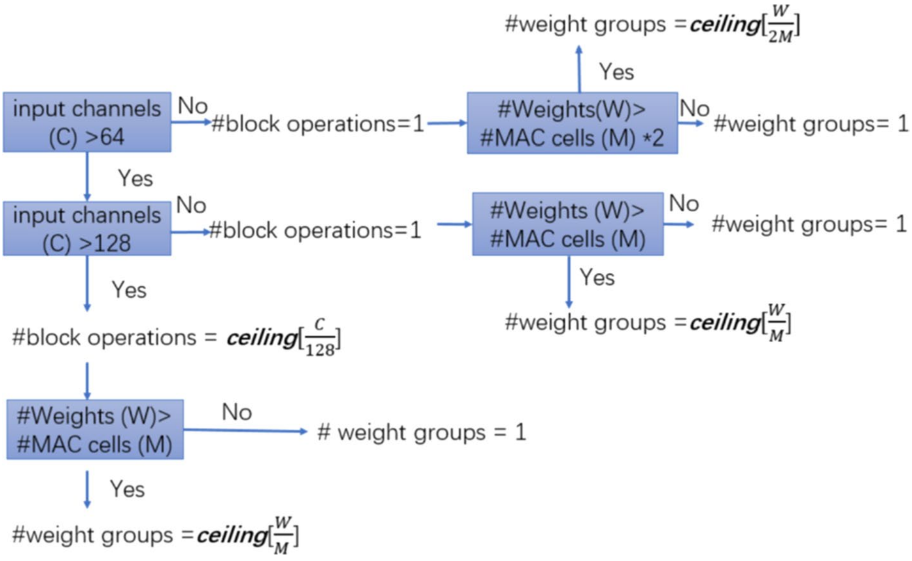

#weight groups = *ceiling*[$\frac{W}{2M}$]
Yes
input channels (C) >64
No
#block operations=1
#Weights(W)> #MAC cells (M) *2
No
#weight groups= 1
Yes
input channels (C) >128
No
#block operations=1
#Weights (W)> #MAC cells (M)
No
#weight groups= 1
Yes
#weight groups = *ceiling*[$\frac{W}{M}$]
Yes
#block operations = *ceiling*[$\frac{C}{128}$]
#Weights (W)> #MAC cells (M)
No
# weight groups = 1
Yes
#weight groups = *ceiling*[$\frac{W}{M}$]

Fig. 10 The method to decide the number of weight groups and the number of block operations

the weights do not need to be divided and thus the number of weight group is 1.

When C is bigger than 64, one MAC cell forms one computation unit and the number of the weights in one weight group should not exceed the number of the MAC cells. When $W>M$, weights need to be divided into weight groups to be processed group by group. The number of weight groups is **ceiling**[W/M]. Each group can have up to M weights. When W is smaller than or equal to M, the number of weight groups is 1.

After computation in CMAC, the partial sums are accumulated and formed accumulative sums in ABUF. The accumulative sums are sent to the next stage until the accumulative is complete. To prevent accumulation overflow, the accumulation sum precision of INT34 is used instead of INT8. The size of the ABUF should be bigger than the size of one accumulative sum at least. Because ABUF size is fixed, the input feature map should be divided into multiple 3D chunks (width*height*channel) to be processed chunk by chunk, if the size of the input feature map is too large. The input feature map is processed chunk by chunk. One chunk is processed to form an accumulative sum. All the accumulative sums are integrated into the output feature map. There can be overlaps between two adjacent chunks due to data sharing. However, there is no overlap between two adjacent chunks when filter size is 1*1, since no data is shared. When the filter size of the layer is 3*3 and the stride size is 1, the overlapping size is 2. When the size of the filter is 3*3 and the stride size is 2, the overlapping size is 1. The maximum capacity of ABUF is 136B*128 of INT8. Let AS_h be the accumulative sum height and P be the accumulative sum channel. The accumulative sum size for one kernel is $AS\_h^2 * P$, under the condition that accumulative sum height = width = AS_h. The examples we used in our experiments satisfy this condition and most of the well-known neural networks resize the image into the same height and width [12, 14]. However, our method can also be used when this condition is not satisfied, and thus this condition does not affect the universality of our method.

5 Experimental Results and Analysis

To understand how the on-chip memory storage and computation modules determine the performance of neural network computation, we develop techniques to estimate the performance and to use it for capitalizing the exploration of the design tradeoffs. We estimated the processing times of GDDR6 based DRAM, SRAM, and NVDLA in each stage with pipelining and parallel architecture.

The processing time was estimated by counting the number of clock cycles required to compute all the layer of a given deep neural network.

With a reasonable assumption, the clock frequency of 1.5 GHz was used for the GDDR6, and the clock frequency of 1.0 GHz was used for both of SRAM and NVDLA. By using the above frequencies, the performance estimation results for YOLOv3 and faster RCNN are shown in Table 3. Table 4 shows the performance when two sets of computing hardware modules are used for parallel processing.

From the result tables, one can see that when the number of MAC cells is doubled, the performance can not be improved twice. That is because the total time is composed of the computing time and transmission time. Note that the two channels of the DRAM are utilized with one computing set as well as with two computing sets. From the six cases in Tables 3 and 4 the performance of YOLOv3 ranges 0.124 to 0.345 s/frame, and the performance of faster RCNN

Table 3 Performance estimation with one computing set

(a) YOLOv3

SRAM size	16 MAC cells	32 MAC cells
1 MB	0.345 s/frame	0.231 s/frame
2 MB	0.320 s/frame	0.206 s/frame
4 MB	0.315 s/frame	0.201 s/frame

(b) Faster RCNN

SRAM size	16 MAC cells	32 MAC cells
1 MB	0.191 s/frame	0.129 s/frame
2 MB	0.182 s/frame	0.120 s/frame
4 MB	0.175 s/frame	0.113 s/frame

Table 4 Performance estimation with two computing sets

(a) YOLOv3

SRAM size	16 MAC cells	32 MAC cells
1 MB	0.210 s/frame	0.154 s/frame
2 MB	0.186 s/frame	0.129 s/frame
4 MB	0.181 s/frame	0.124 s/frame

(b) Faster RCNN

SRAM size	16 MAC cells	32 MAC cells
1 MB	0.107 s/frame	0.077 s/frame
2 MB	0.098 s/frame	0.067 s/frame
4 MB	0.091 s/frame	0.061 s/frame

ranges 0.061 to 0.191 s/frame. This shows that a designer can easily make trade-offs between the performance and the hardware cost. This technology can be used to explore the design space and to optimize the hardware architecture so that the cost can be minimized while satisfying the required performance. This research is useful for a designer to find an optimal solution by exploring the hardware design space efficiently.

6 Conclusion

In this work, we proposed techniques to estimate the execution time of the chosen hardware architecture for a deep neural network. This technology can be used to explore the design space and to optimize the hardware architecture so that the cost can be minimized while satisfying the required performance. It is useful for a designer to efficiently estimate performance in hardware design. In our experimental results, YOLOv3 can be executed in 0.124 to 0.345 s depending on the hardware architecture (resources). Similarly, the faster RCNN can be executed in 0.061 to 0.191 s. By changing the hardware architecture such as SRAM sizes, computing module types, and the number of computing modules for parallel processing, various tradeoffs can be evaluated and an optimal design can be found efficiently. The technique developed in this research looks very useful for the reduction of both cost and time to market.

Acknowledgements This work was supported by Samsung Electronics Co., Ltd.

References

1. Graphics Double Data Rate (GDDR6) SGRAM Standard JESD250B. https://www.jedec.org/standards-documents/docs/jesd250b. (2018)
2. NVIDIA,NVIDIA open source ML accelerator, http://nvdla.org, (2018)
3. Norman, P. J.; Cliff, Y.; Nishant, P. et al.: In-Datacenter Performance Analysis of a Tensor Processing Unit. In Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA '17, pp 1–12 (2017).
4. Tianshi, C.; Zidong, D.; Ninghui, S. et al.: DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In: The 19th international conference on Architectural support for programming languages and operating systems, 269–284 (2014)
5. Yunji, C.; Tao, L.; Shijin, L. et al.: DaDianNao: A machine-learning supercomputer. In: The 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 609–622 (2014)
6. Zidong, D.; Robert, F.; Tianshi, C. et al.: ShiDianNao: shifting vision processing closer to the sensor. In: The 42nd Annual International Symposium on Computer Architecture, 92–104 (2015)
7. Daofu, L.; Tianshi, C.; Shaoli, L. et al.: PuDianNao: A polyvalent machine learning accelerator. In: The Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, pp 369–381 (2015)
8. Shaoli, L.; Zidong, D.; Jinhua, T. et al.: Cambricon: An instruction set architecture for neural networks. In: The 43rd International Symposium on Computer Architecture, 393–405 (2016)
9. Shijin, Z.; Zidong, D.; Lei, Z. et al.: Cambricon-X: An accelerator for sparse neural networks. In: The 49th Annual IEEE/ACM International Symposium on Microarchitecture 20, pp 1–12 (2016)
10. Manoj, A.; Han, C.; Michael, F.; Peter, M.: Fused CNN accelerator. In: The 49th Annual IEEE/ACM International Symposium on Microarchitecture 22, pp 1–12 (2016)
11. High Bandwidth Memory (HBM) DRAM JESD235C. https://www.jedec.org/standards-documents/docs/jesd235a. (2020)
12. Shaoqing, R.; Kaiming, H.; Ross, G.; Jian, S.: Faster R-CNN: Towards real-time object detection with region proposal networks. Computer Vision and Pattern Recognition (cs.CV), arXiv: (2015) 1506.01497
13. Joseph, R.; Santosh, D.; Ross, G.; Ali, F.: You Only Look Once: Unified, Real-Time Object Detection. Computer Vision and Pattern Recognition (cs.CV), arXiv: (2016) 1506.02640
14. Joseph, R.; Ali, F.: YOLOv3: An Incremental Improvement. Computer Vision and Pattern Recognition (cs.CV), arXiv: (2018)1804.02767
15. Qijie, Z.; Tao, S.; Yongtao, W. et al.: M2Det: A Single-Shot Object Detector based on Multi-Level Feature Pyramid Network. In: AAAI (2019)

16. Sachin, M., Mohammad, R., Linda, S., Hannaneh, H.: ESPNetv2: A Light-weight, Power Efficient, and General Purpose Convolutional Neural Network. In: Proceedings IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR) 2019, pp 9190–9200
17. Wei, L.; Shengcai, L.; Weiqiang, R. et al.: High-Level Semantic Feature Detection: A New Perspective for Pedestrian Detection. In: CVPR (2019)
18. Irtiza, H.; Shengcai, L.; Jinpeng, L. et al.: Pedestrian Detection: The Elephant. In: The Room. arXiv preprint arXiv:2003.08799 (2020)
19. Chen, W.; Wenjing, W.; Wenhan, Y.; Jiaying, L.: Deep Retinex Decomposition for Low-Light Enhancement. In British Machine Vision Conference (2018)
20. Thang, V.; Cao, V. N.; Trung, X. P. et al.: Fast and Efficient Image Quality Enhancement via Desubpixel Convolutional Neural Networks. In: ECCV workshop (2018)
21. Enrico, R.; Marco, R.; Anna, M. N. et al.: Pareto Optimal Design Space Exploration for Accelerated CNN on FPGA. In: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)
22. Cheng, L.; Man-Kit, S.; Hongxiang, F. et al.: Towards efficient deep neural network training by FPGA-based batch-level parallelism. In: 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)
23. Song, H.; Xingyu, L.; Huizi, M. et al.: EIE: efficient inference engine on compressed deep neural network. In: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture
24. Yongming, S.; Michael, F.; Peter, M.: Escher: A CNN Accelerator with Flexible Buffering to Minimize Off-Chip Transfer. In: 2017 IEEE 25th annual international symposium on field-programmable custom computing machines
25. Arthur, S.; Francesco, C.: Optimally Scheduling CNN Convolutions for Efficient Memory Access. IEEE transactions on computer-aided design of integrated circuits and systems (2019)