Value-driven Synthesis for Neural Network ASICs

Zhiyuan Yang
University of Maryland, College Park
zyyang@umd.edu

Ankur Srivastava
University of Maryland, College Park
Ankurs@umd.edu

ABSTRACT

In order to enable low power and high performance evaluation of neural network (NN) applications, we investigate new design methodologies for synthesizing neural network ASICs (NN-ASICs). An NN-ASIC takes a trained NN and implements a chip with customized optimization. Knowing the NN topology and weights allows us to develop unique optimization schemes which are not available to regular ASICs. In this work, we investigate two types of value-driven optimized multipliers which exploit the knowledge of synaptic weights and we develop an algorithm to synthesize the multiplication of trained NNs using these special multipliers instead of general ones. The proposed method is evaluated using several Deep Neural Networks. Experimental results demonstrate that compared to traditional NNPs, our proposed NN-ASICs can achieve up to 6.5x and 55x improvement in performance and energy efficiency (*i.e.* inverse of Energy-Delay-Product), respectively.

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**;

KEYWORDS

Neural Network, Architecture, Energy-efficiency

1 INTRODUCTION

Conventional approaches for implementing neural networks have relied on CPUs, GPUs and neural network processors (NNPs). NNPs are neural network (NN) specific programmable implementation platforms which take advantage of the NN topology which conventional CPUs/GPUs cannot exploit. However, the performance and energy efficiency of NNPs are restricted by memory access and computation parallelism. While the cost of memory access (*e.g.*, latency, energy *etc.*) can be reduced using new on-chip memory technologies (*i.e.* eDRAM [3]), improving the computation parallelism (thus improving the performance) is often accompanied with substantial increase in the chip area and energy or degradation of the NN accuracy [3, 4]. Fast and efficient implementation of NNs in hardware is still an open problem.

In this work, we target at implementing trained NNs with ASICs. The ASIC implementation can help substantially improve the performance and energy efficiency compared to conventional NNPs (and CPUs/GPUs) through customized optimizations. While traditional synthesis techniques can be used to implement such neural network ASICs (NN-ASICs), NNs comprise other details that can be used to further optimize the designs. Since the topology and synaptic weights of a trained NN (Figure 1(a)) are known a priori, we can estimate the typical value each compute unit (*e.g.*, multiplier) will process before synthesizing the NN-ASIC. Therefore, we can substantially optimize the design of these compute units beyond what traditional synthesis can achieve. In this work, we focus on synthesizing the multiplication with value-driven optimized multipliers (Section 3). Compared to traditional NNPs, the performance and energy efficiency of the proposed NN-ASICs can be improved in a threefold manner:

- The size, energy and delay of a single multiplier is reduced due to the simplified circuit.
- More multipliers can be implemented due to the smaller size of a single multiplier, which increases the computation parallelism thus improving the performance.
- By hard coding weights in multipliers, the memory access cost is reduced due to lesser demand on off-chip memory access/bandwidth.

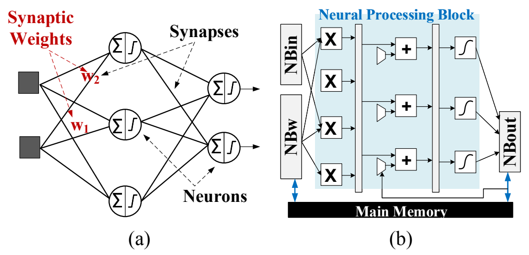


Figure 1: (a) An illustration of the neural network and (b) the DianNao [2] architecture (NBin = Input Cache; NBw = Weight Cache; NBout = Output Cache).

In Section 4, we propose an algorithm for synthesizing the multiplication of NNs with value-driven optimized multipliers such that the performance and energy efficiency are maximized. Such value-driven synthesis method does not affect the NN accuracy since it does not change the topology or the parameter of the NN. However, we can further improve the efficacy of our technique by observing that NN models can be trained to have more amenable weight values (Section 5).

The proposed value-driven synthesized NN-ASIC is evaluated using a set of Deep Neural Networks (DNNs) [7–9, 12, 13]. The baseline case is a DianNao-like NNP [2] implemented with general 16-bit Q9.7 fixed-point multipliers. As demonstrated by the experimental results (Section 6), compared to the baseline case, our method can achieve up to 6.5x and 55x improvement in performance and energy efficiency, respectively.

2 PRELIMINARIES AND MOTIVATIONS

Figure 1(a) illustrates an NN which consists of multiple interconnected neurons that perform weighted accumulation of synaptic signals and then feed the convolved value into non-linear activation functions. Currently, there is a growing demand of processing NN applications in NNPs which have better performance and energy efficiency compared to conventional CPUs and GPUs [2, 3, 6]. A typical NNP architecture is shown in Figure 1(b). On deploying the NN, the data (*i.e.* inputs, weights *etc.*) are fetched from the main memory to the on-chip cache, and then are processed through the

ISLPED '18, July 23–25, 2018, Seattle, WA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5704-3/18/07…$15.00
https://doi.org/10.1145/3218603.3218634

Neural Processing Block (NPB). The intermediate results may also be written back to the main memory. For large NNs, the memory access may become limited, thus motivating the wide study of techniques that reduce the cost of off-chip memory access. For instance, we can implement the main memory with eDRAM to achieve faster access [3]. Another factor that restricts the performance and energy efficiency of NNPs is the computation parallelism which is determined by the number of compute units (*e.g.*, multipliers, adders *etc.*). While increasing the number of compute units can improve the performance, this may substantially increase the chip area and energy. If the chip area or energy is a design constraint, there is little room to improve the performance of NNP. In order to solve this dilemma, some people propose "multiplier-free" NNPs [4] where the multiplication is realized using modules (*e.g.*, shifters) with lesser delay, area and energy than traditional multipliers. In order to achieve this, the weight of NNs should be rounded to typical values which may cause significant loss of accuracy when applied to large NNs. This means we need much extra effort to train the NN to regain the accuracy.

Contemporary NNPs are designed as general as possible such that they can be used to process a wide range of NNs. This kind of NNPs is based on the assumption that we have no knowledge of NN parameters before fabricating the NNP. However, as the NN technology becomes mature, for some applications, the NN is accurate enough with a pre-trained set of weights. For instance, we have already achieved high accuracy when using the NN to detect the handwritten digit [14]. In this case, after an NN is trained, there will be a demand of fabricating an NN-ASIC to process this NN. Then this NN-ASIC can be used as IP cores to build various systems. Compared to general NNPs, the ASIC implementation can substantially improve the performance and energy efficiency through customized optimizations. In this work, we investigate how the performance and energy efficiency of NN-ASICs are improved by synthesizing the multiplication of NNs with multipliers optimized according to the synaptic weight. As will be illustrated in Section 3, this kind of value-driven optimized multipliers has lesser area, delay and energy compared to general multipliers.

3 VALUE-DRIVEN OPTIMIZATION OF MULTIPLIERS

Although our work can be applied to NN-ASIC implemented with any types of multipliers, we will focus on fixed-point multipliers in this work. From now on, we use "general multiplier" to represent the 2-input 16-bit fixed-point multiplier designed for the Q9.7 fixed-point data format (Figure 2(a)). In this section, we will investigate the property of two value-driven optimized multipliers which can be used in NN-ASICs: (1) Fixed-weight Multipliers (Section 3.1) and (2) Varied-weight Multipliers (Section 3.2). Their properties are compared against the general multiplier. All multipliers are synthesized with 90nm standard cell library using Cadence RTL Compiler which also performs early timing, power and area analysis.

3.1 Fixed-Weight Multipliers

This kind of multipliers is illustrated in Figure 2(b). One input is hardwired with a constant value while the other input stays as a variable. Multipliers with different hard coded input values have distinct area, energy (*i.e.* Power-Delay-Product) and delay. In order to study the property of such multipliers, we randomly choose 300 different values within the range that can be represented by the signed Q9.7 fixed-point data format and hard code the value to one input of the multiplier (*i.e.* the Fixed-Input). Each of these multipliers is then synthesized using Cadence RTL Compiler and the area,

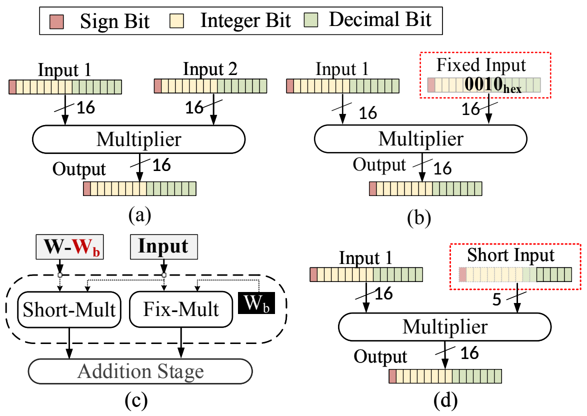


Figure 2: An illustration of (a) the general 16-bit fixed-point multiplier, (b) the fixed-weight multiplier (the fixed-input value is 0010_{hex}), (c) the varied-weight multiplier and (d) the short-input multiplier.

energy and delay of each multiplier is analyzed. Figure 3(a)-(c) illustrate the distribution of normalized area, energy and delay of the fixed-weight multiplier, respectively, among the selected samples. The data (x-axis in Figure 3(a)-(c)) are normalized to the general multiplier. The results demonstrate that compared to the general multiplier, the area and energy of the fixed-weight multiplier can be substantially reduced. The reason of such improvement is that when one input of the multiplier is fixed, the circuit of multiplier can be simplified by using fewer gates (and connections). For example, if the value of the fixed input is 0010_{hex} (as illustrated in Figure 2(b), which represents 0.125 in Q9.7 fixed-point data format), the corresponding fixed-weight multiplier just needs to fulfill left-shifting *Input 1* by 5-bits. This circuit is much simpler than a general multiplier.

According to this phenomenon, we can use a fixed-weight multiplier to perform the multiplication with a specific synaptic weight (by fixing the value of one input to the synaptic weight). Compared to merely using the general multiplier, the use of fixed-weight multipliers can lead to several improvements in the performance and energy efficiency of NN-ASICs: **(1)** Compared to general multipliers, the area of a single fixed-weight multiplier is much smaller, hence we can implement more multipliers within a certain chip area to achieve higher computation parallelism. **(2)** Since the energy of a single multiplier is substantially smaller, the total computation energy can be reduced. **(3)** Since the synaptic weight is hard-coded, the main memory access for weight can be avoided thus improving the memory access efficiency.

3.2 Varied-Weight Multipliers

In the extreme case, we can implement a fixed-weight multiplier dedicated to each multiplication. In reality, this may not be feasible due to the chip area constraint thus necessitating sharing multipliers between synapses. However, a fixed-weight multiplier can only be shared by synapses with the same weight. In some cases, this leads to suboptimal solutions. Considering an NN which contains 100 synapses with weight w_0 and five synapses each with weight w_1, w_2, w_3, w_4, w_5, respectively. If the area constraint only allows 6 multipliers in total, we can only have one fixed-weight multiplier for weight w_0 and all the 100 synaptic operations are performed on this multiplier. This leads to 100 cycles to perform all the multiplications in this NN (assuming each multiplier only performs one multiplication in one clock cycle and multipliers execute in parallel).

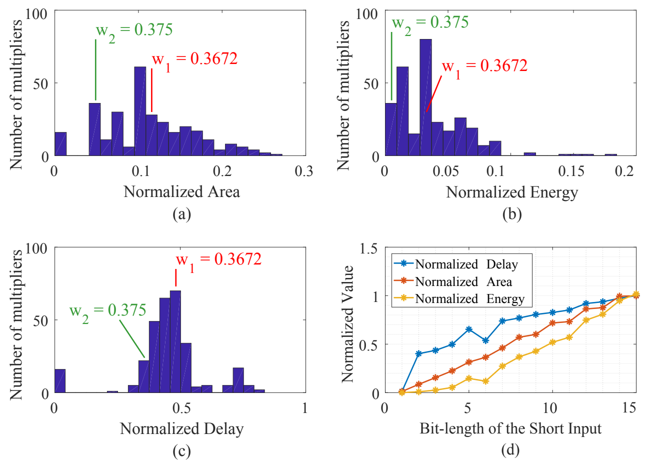


Figure 3: The distribution of the normalized (a) area (b) energy (Power-Delay-Product) and (c) delay for the fixed-weight multiplier. The red and green markers indicate the property of two multipliers with different fixed-input values ($w_1 = 0.3672$, $w_2 = 0.375$). (d) The normalized area, energy and delay for the short-input multiplier with different bit-length of the short-input. (The data are normalized to the general multiplier.)

On the other hand, if we can have a **Varied-weight Multiplier** such that the synaptic operations with w_1, w_2, w_3, w_4, w_5 can all be performed on this multiplier (in totally 5 cycles), we can add another four fixed-weight multipliers for w_0. In this case, the total number of cycles to perform all the multiplication is reduced to 20.

Figure 2(c) illustrates a varied-weight multiplier. It is composed of a **fixed-weight multiplier** and a **short-input multiplier** (introduced later). The fixed-input value of the fixed-weight multiplier is the **base weight (w_b)**, while the short-input multiplier is used to calculate the multiplication between the input and the *weight variance* ($w - w_b$ as illustrated in Figure 2(d) where w is the original weight). The results of the two multipliers are then summed up in the addition stage. Figure 2(d) illustrates a **Short-input Multiplier**: the bit-length of one input (*i.e.* the short input) is shortened to represent the least significant bits of a Q9.7 fixed-point data while the other input stays as a variable. Figure 3(d) shows the normalized area, energy and delay of the short-input multiplier with the short input of different bit-length. According to the result, as the bit-length decreases, the area/delay/energy of the short-input multiplier becomes smaller than the general multiplier. Therefore, it is possible that a varied-weight multiplier has lesser area and energy than a general multiplier even if it is composed of a fixed-weight multiplier and a short-input multiplier. For example, in the varied-weight multiplier illustrated in Figure 2(d), let $w_b = 0.375$. If we would like to use this multiplier to process synaptic operations with weights between 0.375 (*i.e.* 0030_{hex}) and 0.3984 (*i.e.* 0033_{hex}), the bit-length of the short-input multiplier contained in this varied-weight multiplier should be 2. Based on the data in Figure 3, the area and energy of this varied-weight multiplier is only 15% and 4% of the general multiplier, respectively. Moreover, in this example, we can use 2 bits to store the weight variance instead of memorizing the original weight (16 bits) thus reducing the memory access.

Algorithm 1: Multiplication-Synthesis Algorithm

```
1 Build the library for the value-driven multipliers
2 Implement each multiplication with a dedicated multiplier
3 Estimate the area, $N_{clk}$ and $EDP$
4 Insert the initial design into $POL$
5 Cluster multipliers to meet the chip area constraint
6     Select one design from the $POL$
7     use one multiplier to perform multiple multiplications
8     estimate the area, $N_{clk}$ and $EDP$
9     update $POL$
```

4 VALUE-DRIVEN SYNTHESIS OF NN-ASICS

The NN-ASIC is developed based on the popular NN chip architecture (Figure 4(a)) [2]. When the NN is small enough, we can implement each operation with a compute unit; otherwise, the compute unit should be reused to meet the chip area constraint. Different from [2, 3], the multiplication stage in our proposed NN-ASICs is synthesized using the **value-driven optimized multipliers**. The synthesis algorithm will be introduced in this section.

Problem Statement. Given a total multiplier area constraint (A_{mult}) and a trained NN, we would like to determine the number and type of value-driven optimized multipliers as well as the mapping of multiplications to these multipliers. The objective would be to maximize the performance and energy efficiency of the multiplication stage.

4.1 Algorithm for Single-layer NNs

In this subsection, we will introduce the **Multiplication-Synthesis Algorithm** for single-layer NNs (as illustrated in Algorithm 1 and Figure 4(d)-(h)). This algorithm can be modified to solve the synthesis problem for multiple-layer NNs (Section 4.2).

Given a single-layer NN (*e.g.*, Figure 4(b)), the algorithm first builds the "library" for fixed-weight and short-input multipliers using the synthesis tool according to the synaptic weights of the NN (line 1 in Algorithm 1). Since the number of unique synaptic weights in most NNs is not large due to the weight-sharing between synapses, the size of the fixed-weight multiplier library will be finite. Besides, the size of the short-input multiplier library is limited by the bit-length of the short-input (thereby the library size is also finite). After this, the design of the multiplication stage is initialized by implementing a fixed-weight multiplier dedicated to **each multiplication** (as illustrated in Figure 4(d)). Then the area, execution time and energy efficiency of this initial design are evaluated. The area is determined by summing up the area of all multipliers used in the design; the execution time is represented by the number of clock cycles for performing all the multiplications in the NN (N_{clk}) as shown in Equation 1a; the energy efficiency is represented by 1/Energy-Delay-Product ($\frac{1}{EDP}$) where EDP is estimated using the product of total energy and N_{clk} (Equation 1b).

$$N_{clk} = \max_{j \in \mathcal{J}} N_{OPM,j} \tag{1a}$$

$$EDP = Energy \times N_{clk}. \tag{1b}$$

In Equation 1a, $\mathcal{J}$ is the group of all multipliers in the design. $N_{OPM,j}$ is the number of operations allocated to the j^{th} multiplier. Assuming each multiplier only performs one multiplication in one clock cycle, then we need $N_{OPM,j}$ cycles to perform the $N_{OPM,j}$ multiplications mapped to the j^{th} multiplier. Therefore, $\max_{j\in\mathcal{J}} N_{OPM,j}$ indicating the overall clock cycles to process this

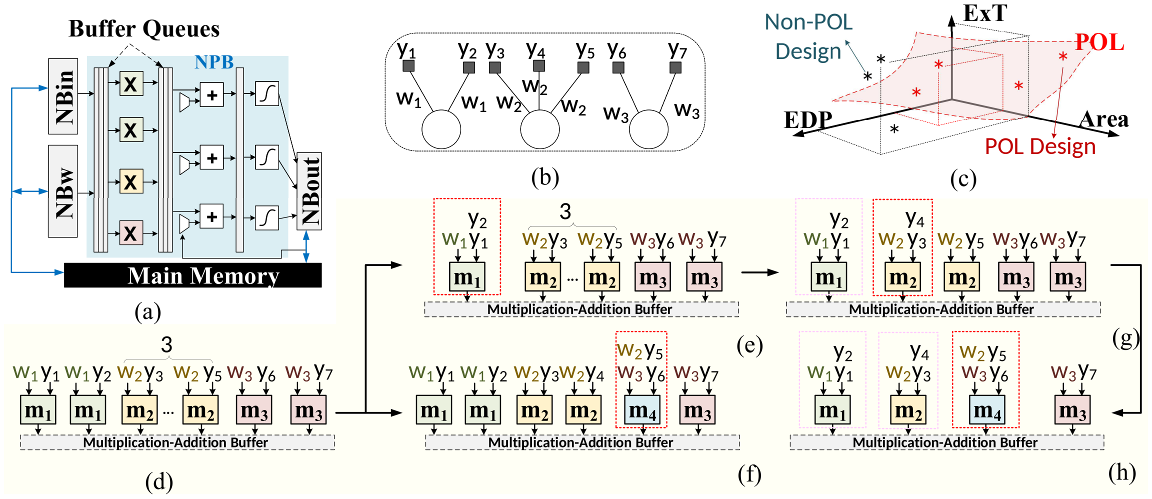


Figure 4: (a) The proposed NN-ASIC architecture. (b) A single-layer neural network. (c) The illustration of the Pareto Optimal List (POL) with ExT representing the execution time, Area representing the multiplication stage area and EDP representing the Energy-Delay-Product. (d) The initial design of the multiplication stage by implementing a fixed-weight multiplier dedicated to each multiplication in the neural network shown in Figure 4(b) (each multiplier is represented with a rectangle). (e)(f) two possible designs after clustering a pair of multiplications in the initial design (note that m_4 represents a varied-weight multiplier which can perform the multiplication with weight equal to w_2 and w_3). (g) On possible clustering result based on the design in (e). (h) One possible clustering result based on the design in (g).

NN since each multiplier executes in parallel. In Equation 1b, *Energy* is the total energy.

This initial design has the shortest execution time (and the smallest EDP) yet the largest area. If the area of this design is smaller than A_{mult}, it is the final design of the multiplication stage. Otherwise, we will iteratively cluster two multiplications and allocate them to a single multiplier to reduce the area at the cost of increasing the execution time and EDP. On one hand, we can limit to clustering only those multiplications with the same weight (as illustrated by Figure 4(d)(e)). On the other hand, we can use a varied-weight multiplier to perform multiplications with similar weights (as shown in Figure 4(d)(f)). In either case, the number of multipliers is reduced (thus the area of the multiplication stage is reduced) while the number of cycles to perform the multiplication increases (*e.g.*, $N_{clk} = 2$ in Figure 4(e)(f)). The clustering process continues until we find a design whose area is smaller than A_{mult} while the execution time and EDP are minimized. For the example shown in figure 4, if the area constraint only allows 4 multipliers, one possible approach of the clustering process is illustrated by Figure 4(d)→(e)→(g)→(h).

Intuitively, we can go through all the designs with area under the constraint to find the optimal design (*i.e.* minimized execution time and EDP). However, the size of the design space increases exponentially with the number of synaptic weights, thus making the brute force search method infeasible. In order to increase the efficiency of our algorithm, after acquiring the designs resulted from clustering the initial design, we will only memorize the **Pareto Optimal designs** (with respect to area, EDP and execution time) in the Pareto Optimal List (*POL*) and discard other designs (line 4 in Algorithm 1). In the following process, we select the designs in the *POL* to further cluster the multipliers and update the *POL* after a new design is acquired. Since the size of the *POL* is almost constant compared to the whole design space, we can thereby significantly improve the efficiency of finding the optimal design. In order to speed up the clustering process, instead of clustering two multipliers in one step, we can use one multiplier to perform multiple multiplications (line 7 in Algorithm 1).

4.2 Algorithm for Multiple-layer NNs

In reality, an NN often contains several layers and some DNNs even consist of hundreds of layers [7]. In this case, we initialize the multiplication stage by implementing a fixed-weight multiplier dedicated to each multiplication in **all** layers of the NN. Then we follow the same framework as introduced in Section 4.1 to cluster multiplications to reduce the area until we find a design with an area smaller than A_{mult}. We assume the process of different layers is not pipelined. Therefore, the execution time of the multiple-layer NN is the sum of the execution time of all layers in the NN.

4.3 Accounting the Memory Access

The algorithm noted above assumes perfect off-chip memory access such that the weights and intermediate data can be accessed immediately from the main memory without any cost when they are needed. In reality, this is not possible, hence we need to exploit the on-chip cache to avoid heavy off-chip memory access. However, if the total chip area is a constant, there is a competition between allocating resources for compute units (*i.e.* multipliers, adders *etc.*) and the on-chip cache. Too much cache may cause too few compute units which leads to higher compute latency while too little cache may force us to have higher main memory access. In this work, we use binary search method to determine the optimal area for multipliers and the area for cache (supposing the total chip area is constant): we first initialize an area for the multiplication stage (A_{mult}) and use the method introduced above to design the multiplication stage. Other compute stages and the on-chip cache are designed accordingly; then we evaluate the performance and energy efficiency of the design using an in-house simulator and update A_{mult} accordingly.

5 SYNTHESIS-DRIVEN OPTIMIZATION OF WEIGHTS

Since the NN-ASIC is synthesized according to the synaptic weight of an NN, the value of weights will affect the performance and energy efficiency of the NN-ASIC. For instance, suppose there is a synapse in the NN with weight equal to 0.3672 ($002f_{hex}$). To perform this multiplication, we can use a **fixed-weight multiplier** with the fixed input equal to 0.3672 whose normalized area and energy are 0.12, 0.038, respectively (as illustrated by w_1 in Figure 3(a)(b) and note that the data are normalized to the general multiplier). However, if we round this weight to 0.375 (0030_{hex}), we can use a multiplier with much smaller area and energy as shown by w_2 in Figure 3(a)(b). As shown by this example, if we perturb the weight to a nearby value such that the multiplication can be performed by a smaller and more efficient multiplier, we may further improve the performance and energy efficiency of the NN-ASIC.

The process of perturbing weights might degrade the NN accuracy. In order to control the degradation, we select a threshold, R, for perturbing the weight. If $\frac{|w'-w|}{|w|} \leq R$ (where w is the original weight while w' is the new weight), we can perturb w to w'; otherwise, the perturbing process is forbidden. Given a set of unique weights in an NN (*e.g.*, $\{w_1, w_2, ..., w_n\}$), we assume the multiplication with each weight is operated using a dedicated fixed-weight multiplier, hence each weight is associated with a cost (determined by the area, delay and energy of the multiplier): $\{c_1, c_2, ..., c_n\}$. In order to determine the new value for each weight, we cluster the weights and map the weights in each cluster to the weight with the smallest cost in this cluster. For example, if $\{w_1, w_2, w_3\}$ are clustered, the new value for each of the three weights is the weight with the smallest cost ($\min(c_1, c_2, c_3)$). In this work, we use a dynamic programming based method to clustering the given set of weights and assigning the new weight to each cluster such that the total cost (*i.e.* summing up the cost of all weights) is minimized under the constraint of R. After perturbing the weight, we use the same method as introduced in Section 4 and 4.3 to synthesize the NN-ASIC. In Section 6.4, the effect of this weight optimization method will be studied in detail.

6 EXPERIMENTAL RESULTS AND ANALYSIS

6.1 Setup of the Experiment

In this section, we use our value-driven synthesized NN-ASICs to process several trained DNNs: Lenet[9], AlexNet[8], VGG-16 Net[12], GoogleNet[13] and ResNet[7]. The pre-trained parameters are taken from the Caffe Model Zoo [1] and we round the parameter to the Q9.7 fixed-point data format using Ristretto [5]. For each NN, we use the method introduced in Section 4 and Section 4.3 to design the NN-ASIC and the total chip area (including compute units, on-chip caches *etc.*) is fixed to $A_{chip} = 5mm^2$. Following this we use the in-house simulator to evaluate the total execution time and energy. The results are compared against the **Baseline Case**, where a general NNP is used to process all the NNs. The NNP used in the Baseline Case is based on the DianNao architecture which contains 256 general fixed-point multipliers and 16 16-input fixed-point adders [2]. The total chip area is also $5mm^2$ and the on-chip cache size is 40KB. The clock frequency of each NN-ASIC and the NNP is identical. We use Cacti [11] to calculate the area of the on-chip cache (Fully-Associated) for storing one-byte data and this value is $10.6\mu m^2$. The bandwidth of main memory is 250GB/s while the energy of one byte access is 0.2nJ [2, 3]. The latency and energy of accessing the on-chip cache is neglected [3].

6.2 Comparison to the Baseline Case

In this section, we report the execution time and energy using our proposed NN-ASIC (designed following the method introduced in Section 4 and 4.3) to process the DNNs. The results are compared against the Baseline Case. Figure 5(a) and (b) show the total execution time and energy of NN-ASICs, respectively. In order to illustrate the improvement of NN-ASICs, the data in the figures are normalized to the Baseline Case. Figure 5(c) and (d) illustrate the percentage in total execution time and in energy due to the off-chip memory access, respectively, in both Baseline Case and NN-ASICs. As illustrated in Figure 5(a)(b), NN-ASICs can achieve up to 6.5x faster execution speed and 9.3x lower energy compared to the Baseline Case. As the result, NN-ASICs can achieve up to 55x improvement in energy efficiency (*i.e.* $\frac{1}{EDP}$). This improvement is mainly due to the use of value-driven optimized multipliers:

- Since each multiplier is much smaller than the general multiplier, we can implement more multipliers in the multiplication stage to increase the computation parallelism.
- Since the energy of each value-driven optimized multiplier is much smaller than the general one, the total computation energy will be reduced (as illustrated by Figure 5(b)(d)).
- Due to the hardwired weight, the memory space required for storing the synaptic weight is significantly reduced. In the Baseline Case, the size of NBw is 32KB, while in NN-ASICs, we only need an average of 8KB NBw to store the same amount of weights. As a result, in NN-ASICs, more data can be stored on chip with less cache area. Therefore, we can allocate more compute units while the off-chip memory access is still reduced. In this experiment, the total cache size for Lenet, AlexNet, GoogleNet, VGG-16 and ResNet are 42KB, 30KB, 36KB, 22KB and 36KB, respectively.

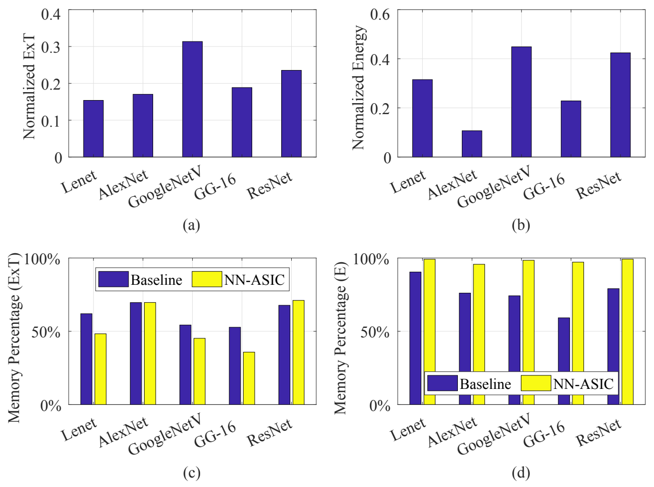
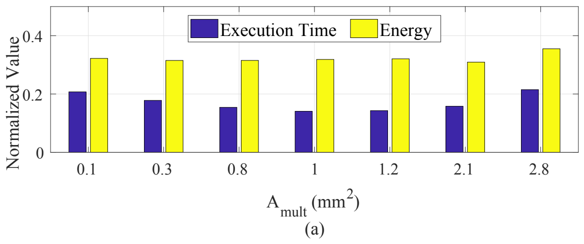


Figure 5: (a) The normalized execution time (ExT) and (b) the normalized total energy for NN-ASICs (all data are normalized to the Baseline Case); the percentage in (c) total execution time and (d) total energy due to off-chip memory access.

6.3 Trade-off Between On-chip Cache and Multipliers

As stated in Section 4.3, the value of A_{mult} will affect the execution time and energy of the NN-ASIC due to the trade-off between the on-chip cache and multipliers. In order to illustrate this effect, we synthesize NN-ASICs for Lenet [9] with different A_{mult} (ranging from $0.1mm^2$ to $2.8mm^2$). Other settings are the same as in Section 6.2. The result is illustrated in Figure 6(a) and the data is normalized to the Baseline Case (*i.e.* assuming DianNao is used to



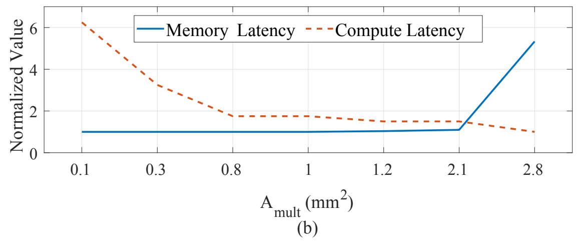


Figure 6: (a) The normalized execution time and energy for NN-ASICs with different A_{mult} for Lenet [9]; (b) the trade-off between the off-chip memory access and compute latency for different A_{mult}.

process Lenet). Figure 6(b) shows the trade-off between the off-chip memory access latency and the compute latency with different A_{mult} and the data is normalized to the minimum memory access latency and compute latency among these cases, respectively. As illustrated by Figure 6(a), there exists an optimal A_{mult} which leads to the largest performance and energy efficiency. For Lenet (with the specific parameters used in this experiment), this value is around $1mm^2$. If A_{mult} is small, the computation parallelism degrades thus causing the compute latency to increase (as shown in Figure 6(b)). On the other hand, large A_{mult} causes the on-chip cache size to decrease and the increased number of memory access will degrade the performance of the NN-ASIC (due to the increased memory latency). However, within a large range of A_{mult} around the optimal point, our proposed technique can always achieve substantial reduction in the execution time (over 5x reduction) and energy (over 2.5x reduction) compared to the Baseline Case.

6.4 Effects of the Weight Optimization

In this section, we will study the impact of perturbing the synaptic weight using the method introduced in Section 5. The results are shown in Table 1. In this table, the execution time (ExT) and energy are normalized to the Baseline Case as in Section 6.2. “No Pert” represents the case where the weight is not perturbed (*i.e.* the results shown in Section 6.2); “Perturb” illustrates the results after perturbing the weight with $R = 0.25$ (Section 5). The NN accuracy (Acc) for Lenet is tested using MNIST data set [10] while that for other NNs is tested using ImageNet data set [8]. For VGG-16 and ResNet, the top-5 accuracy is used. w_{affect} represents the percentage of the number of weights that are affected by the perturbing method. As shown in the table, for AlexNet, GoogleNet and VGG-16, the w_{affect} is very small (<1%), which indicates the original set of weights is almost optimal with respect to the hardware property. Therefore, perturbing weights causes negligible reduction in the execution time and energy. On the other hand, the w_{affect} for Lenet and ResNet is relatively large (73.9% and 10% respectively), indicating a large portion of weights are optimized by the perturbing process (Section 5). As a result, both the execution time and energy are further reduced. It should be noted that when $R = 0.25$, the accuracy degradation for the NNs is very small which can be easily improved through retraining (*e.g.*, [5]).

Table 1: The Effect of the weight optimization method (ExT = Execution Time; Acc = Neural Network Accuracy; w_{affect} = the percentage of the number of weights that are affected by the perturbing method)

Neural Networks		ExT	Energy	Acc	w_{affect}
Lenet [9]	No Pert	**0.16**	**0.32**	**99.1%**	–
	Perturb	**0.1**	**0.27**	**99%**	**73.9%**
AlexNet [8]	No Pert	0.17	0.11	55.7%	–
	Perturb	0.17	0.11	54.8%	$\ll$0.1%
GoogleNet [13]	No Pert	0.31	0.45	64.3%	–
	Perturb	0.3	0.45	63.3%	0.11%
VGG-16 [12]	No Pert	0.19	0.23	89.0%	–
	Perturb	0.19	0.23	88.4%	0.3%
ResNet [7]	No Pert	**0.29**	**0.43**	**88.8%**	–
	Perturb	**0.26**	**0.42**	**87.8%**	**10.0%**

7 CONCLUSIONS

In this work, we target at implementing a specific NN with the NN-ASIC. Instead of using the general multiplier, we propose to exploit the value of synaptic weights of the NN during the synthesis of the NN-ASIC to achieve substantial improvement in performance and energy efficiency with no impact on the NN accuracy. We evaluate our proposed value-driven synthesized NN-ASIC with a set of state-of-the-art DNNs. The experimental results demonstrate that compared to traditional NNPs with general multipliers, our proposed method can achieve up to 6.5x and 55x improvement in performance and energy efficiency, respectively.

ACKNOWLEDGMENTS

The authors acknowledge that the work has been funded by NSF grant 1301375.

REFERENCES

[1] BVLC.Caffe model zoo. [online] http://caffe.berkeleyvision.org/model_zoo.
[2] T. Chen, et al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, volume 49, pages 269–284. ACM, 2014.
[3] Y. Chen, et al. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.
[4] R. Ding, et al. LightNN: Filling the Gap between Conventional Deep Neural Networks and Binarized Networks. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 35–40. ACM, 2017.
[5] P. Gysel. Ristretto: Hardware-oriented approximation of convolutional neural networks. *arXiv preprint arXiv:1605.06402*, 2016.
[6] S. Han, et al. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 243–254. IEEE Press, 2016.
[7] K. He, et al. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.
[8] A. Krizhevsky, et al. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
[9] Y. LeCun, et al. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pages 396–404, 1990.
[10] Y. LeCun, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
[11] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. 2001.
[12] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
[13] C. Szegedy, et al. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.
[14] L. Wan, et al. Regularization of neural networks using dropconnect. In *Proceedings of the 30th international conference on machine learning (ICML-13)*, pages 1058–1066, 2013.