Complement-based Stochastic Computing Multiplier Design for Convolutional Neural Network Acceleration

by *Jacob Hejderup*

For the degree of

Master of Embedded Systems

At Delft University of Technology

To be defended publicly on 29/06/2021 at 13:00

Supervisor
prof.dr. Sorin Cotofana

Thesis Committee:
prof.dr. Sorin Cotofana
dr.ir. Rene van Leuke
dr.ir. Stephan Wong

Preface

Before you lies the final product of my master studies at TU Delft, which also marks the end of my journey at TU Delft. I started this thesis during uncertain times, where many things were and are still changing. And without the help from the people around me, this outcome would not be possible.

First, I want to thank Sorin Cotofana for being my thesis supervisor and giving me the freedom to propose my project. I also want to express my appreciation for the thesis committee.

Secondly, I want to express my gratitude to my fellow students at TU Delft who helped me. Without your support, my time at TU Delft would be pretty dull. I also want to thank my friends who supported me during this journey.

Thirdly, I am grateful to my family for all the support and help. I want to thank my parents and my two brothers for their love and support.

Jacob Hejderup
Borås, June 21, 2021

Abstract

Recently, it has become popular to use Convolutional Neural Networks (CNNs) in embedded and portable devices. The popularity is based on their high accuracy rate in the field of Computer Vision (CV). However, CNNs are computationally intensive due to the convolutional layer, which accounts for over 90% of the operations. To overcome this problem, many researchers have exerted efforts to develop parallel and customised accelerators. Methods utilised in the accelerators range from bit optimisation to using fixed-point arithmetic, and to reducing the size of the network. Some researchers have also explored alternative computing paradigms such as Stochastic Computing (SC). The great advantage of SC is its ability to perform complex arithmetic with simple hardware. However, a major problem of SC is the trade-off between latency and accuracy. Thus, there have been several attempts to mitigate this factor, ranging from improving the generation of stochastic numbers to parallel bitstreams, to early terminations. This thesis proposes StoHej, a new SC multiplier design that combines stochastic bitstreams and complementary events. The multiplier has two input types, the first is the neural network feature value and the second is the weight value. The weight value determines how many iterations the computation requires. A complement event is utilised if the weight value is greater or equal to 0.5 since the complement of the event yields a smaller number. Thus, the worst-case latency has been reduced from $O(N)$ to $O(\frac{N}{2})$. The proposed multiplier was compared with a Conventional Stochastic Computer (CSC) multiplier and the BISC-MVM multiplier, which is the state-of-the-art for SC multipliers that uses an early termination mechanism. All multipliers were first tested in a software simulation in a general context. Accuracy and latency were measured in a software simulation. The results from these simulations showed a 3.2 times speedup for the proposed design compared to BISC-MVM, with no increase in computational errors. Then, StoHej and BISC-MVM were tested in a CNN inference application with the MNIST dataset. The multipliers were used in a Multiply-Accumulate (MAC) array that was implemented on an FPGA. The results from the experiment show that StoHej had a 1.7x speedup and no loss in accuracy compared to BISC-MVM. StoHej's energy consumption was reduced by 40% when compared to BISC-MVM. The Area-Delay Product (ADP) of StoHej was 30% smaller than BISC-MVM. StoHej's Area-Delay-Energy Product is 2.3x smaller than the BISC-MVM multiplier.

Contents

Chapter 1

Introduction

Artificial Neural Networks (ANNs), also known as Neural Networks (NNs), are a class of networks inspired by the way neurons function in the brain, as they are able to learn and adapt [1]. The ANNs' advantage over traditional algorithms is that they do not require an exact mathematical formulation to solve a problem. Instead, an ANN can approximate a mathematical function via a training process, which is useful for tasks that are hard to define in a mathematical way. These tasks include image classification, speech recognition, text classification, machine translation, art creation, and many more [12, 15, 18, 47, 1]. To further demonstrate the usefulness of ANNs, consider an image recognition application that categories handwritten digits. A traditional approach involves defining a mathematical function for each digit. However, there are many individual variations of the digits, which complicates the algorithm. In contrast, an ANN approximates a function by training on a data set, which in this case are images of handwritten digits.

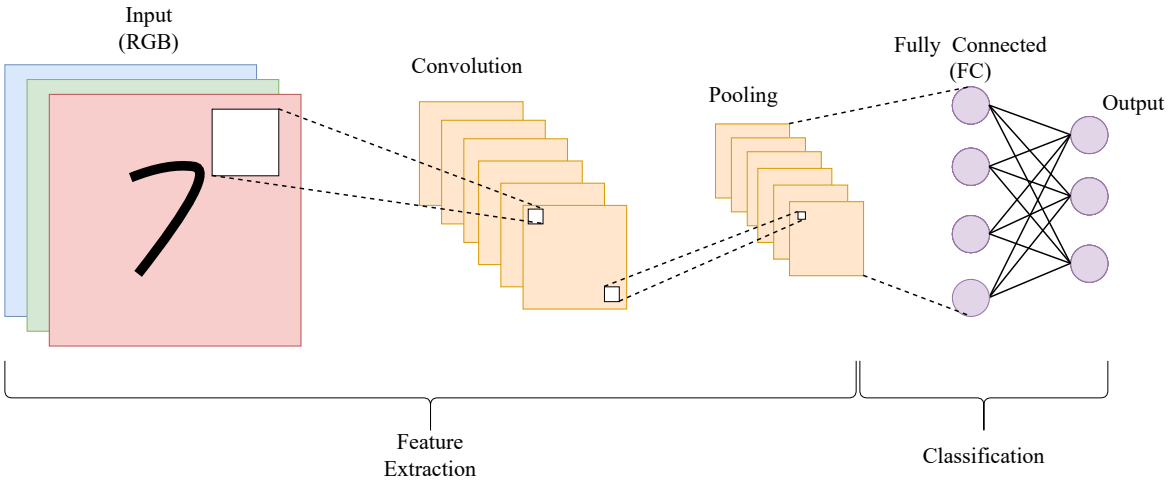


Figure 1.1: *Basic architecture of a Convolutional Neural Network (CNN)*

Convolutional Neural Networks (CNNs) are a particular type of ANNs, mainly used in Computer Vision (CV) applications [27]. A CNN has a similar architecture as an ANN, but it includes additional layer types such as the convolutional layer and the pooling layer as depicted in Figure 1.1. The convolutional layer slides a filter over the entire input image and performs a dot product between the filter and a section of the input image. The pooling layer's purpose is to subsample the convolved data. Due to these additional layers, CNNs outperform conventional ANNs in image-related applications [39, 30].

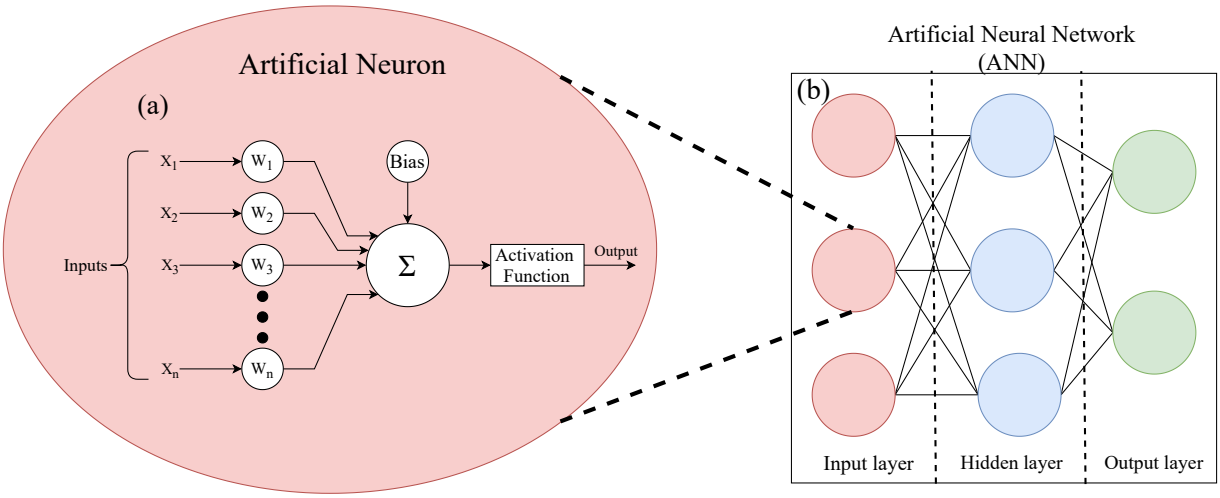


Figure 1.2: *The computational model of an artificial neuron and a diagram depicting an Artificial Neural Network (ANN)*

While ANNs and CNNs are appropriate for the previously mentioned applications, they require intense computation (multiplication dominated) and memory usage, and to illustrate this case, let us consider a network from a computational perspective. An ANN is composed of several artificial neurons, as shown in Figure 1.2. An artificial neuron consists of multiple inputs and has one output and each neuron forms an integrated part of the network. The neurone inputs can represent any feature value such as an image, text, audio, or any other type of data. To obtain a neuron's reaction, one needs to multiply every input with its corresponding weight and the resulting products are added together with a bias term [49]. Finally, a non-linear activation function processes the weighted sum and produces the neurone reaction. To get a better grasp of the computational requirement of the entire network, consider a simple single-layer neural network trained on the MNIST dataset, which consists of black-and-white images of handwritten digits from 0 to 9 [28]. The image size is 784 pixels or 28 by 28 pixels, which means that the first layer has ten neurons representing each digit. Each neuron has 784 inputs, which means we have to perform 784 multiplications and 783 additions to obtain the neuron's response. The computation needs to be iterated ten times, as there are ten neurons, which result in 7840 multiplications and 7830 additions. This calculation only applies to one layer, which is normally not deep enough to achieve a high accuracy rate. This problem is also found in CNNs, since a typical convolutional layer may require more than $10,000$ multiplications [45]. Serial computation of such an application would lead to long latencies in both training and inference. However, ANNs and CNNs exhibit a high degree of inherent parallelism that can be exploited to reduce latency and increase throughput [5].

Even though exploiting parallelism is necessary, it also comes at a high hardware cost, as many ANNs and CNNs are implemented on Graphical Processing Units (GPU) or more specialised hardware such as Tensor Processing Units (TPU) [9, 25]. These hardware platforms can perform training and inference with low latency and high throughput, due to the high utilisation of parallelism. However, due to their power consumption, High-Performance Computing (HPC) devices such as GPUs are not suitable for Internet of Things (IoT) edge computing applications, which are confined in terms of energy consumption and hardware budget [42]. The current solution for edge devices is to send the raw feature data to a cloud server that performs both inference and training. However, this solution has some drawbacks, such as the high energy consumption required to send the data to a cloud server [42], infringement on the users' privacy, and communication bandwidth dependent variable latency [16]. These disadvantages are demonstrated in an example concerning field monitoring application where multiple IoT nodes measure different parameters, such as monitoring the water condition in lakes. In this example, an NN is used

to predict Optical Dissolved Oxygen (ODO) level data [16]. With the current solution, each node in the network sends the raw data to a cloud server that performs the inference. The energy consumption of this solution depends on what kind of data are transferred. If the data size is large, the edge devices may not be able to support it due to bandwidth limitations. There is also a reliability issue in critical applications such as self-driving cars that cannot rely on external cloud servers for inference. For example, a car not detecting an object in time, due to a lost cloud server connection. The final issue is privacy concerns because the raw data are sent to a cloud server can be used for nefarious purposes [41].

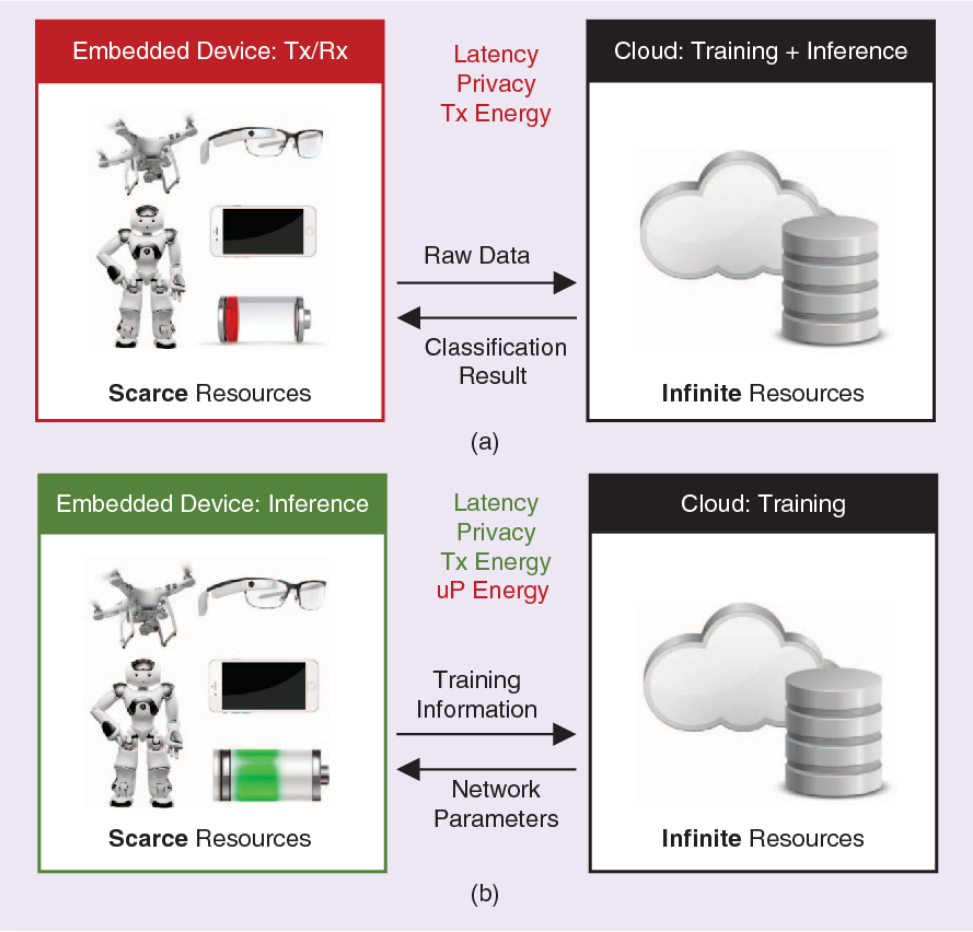


Figure 1.3: *Two ways of performing ANN inference in an IoT application, adapted from [42]*

To mitigate these issues some researchers have proposed the idea of moving inference from cloud servers to edge devices [42] as displayed in Figure 1.3. The training of the ANN is still performed on cloud servers since the training requires more resources. The main advantage of this proposal is that it consumes less energy to compute the inference when compared to transferring the raw data. It also improves reliability, reduces bandwidth usage and energy consumption, and enables privacy. However, a key challenge is to perform the actual inference on platforms with severely limited hardware resources. Thus, ANN acceleration for edge devices should achieve two main goals, namely fast computation and low energy consumption. ANN acceleration for these applications has mainly focused on reducing the ANN size, using fixed-point arithmetic, and bit-optimised computations of the neurons' reaction. [22, 21, 35]. While these methods help to propagate ANNs and CNNs on IoT edge devices, they are not sufficient on their own. For example, a binary multiplier is about 7.8x larger than a binary adder and requires 6.8x more energy than a binary adder [37]. The size of the binary multiplier limits the number of multipliers a computation platform may embed and as such the utilization of the ANN intrinsic parallelism. For

example, the NN accelerator in [10] can only use up to 272 multipliers due to area limitations, while a typical single convolutional layer requires more than $10,000$ independent multiplications [45]. These examples clearly suggest that low-cost multipliers are necessary for ANN inference on edge devices, as conventional multipliers make up a significant part of the chip's real estate. Hence, to be able to exploit the inherent ANN and CNN parallelism, researchers have started to look at alternative methods for low-cost arithmetic such as Stochastic Computing (SC) [31].

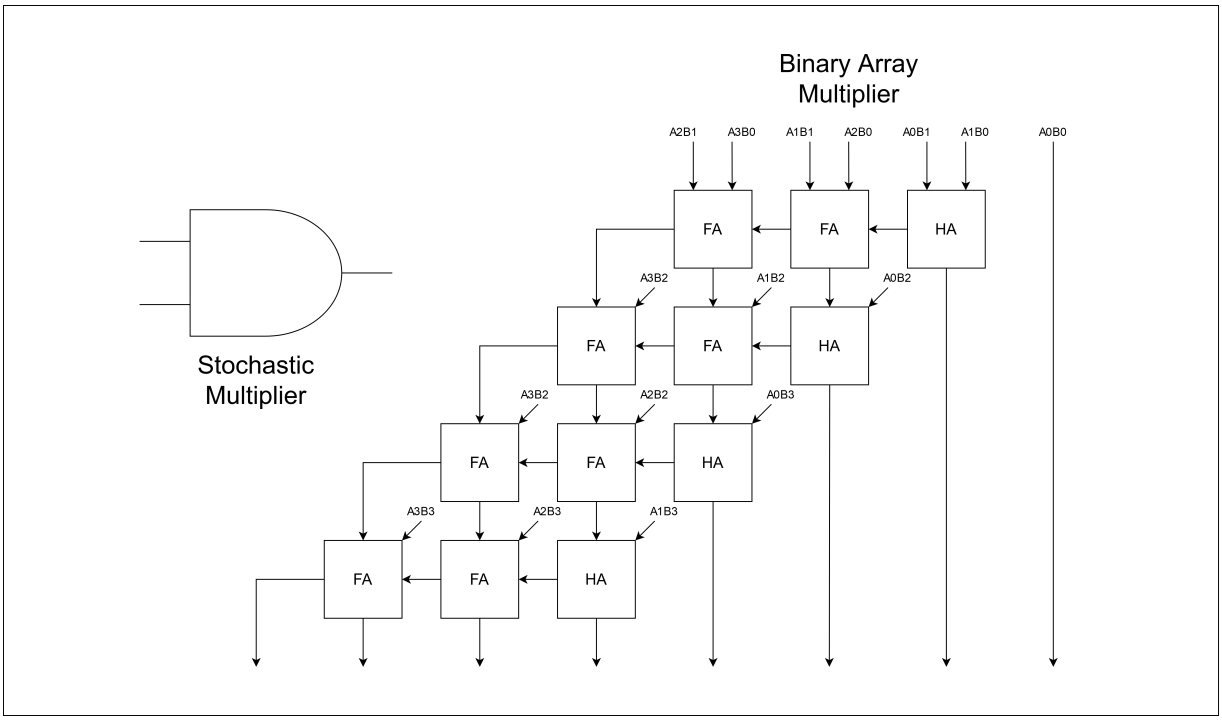


Figure 1.4: *Schematic of a conventional SC multiplier and a binary array multiplier.*

Stochastic Computing (SC) is an alternative computing paradigm that processes data in the form of random or pseudo-random bitstreams. The ratio of $1s$ and $0s$ in the stream determines the value of the stream. For example, If 25% of the streams are $1s$, then the stream represents the numerical value 0.25. Streams can have different orders of $1s$ and $0s$ and different lengths and still represent the same value as long as the ratio is the same. For example, the streams: $(0,0,1,0), (1,0,0,0), (0,1,0,1,0,0,0,0)$ all have the same value because they have the same ratio of $1s$, as shown in Table 1.1. SC can implement complex arithmetic operations with simple and small circuitry, which means that SC can provide inexpensive hardware support for highly parallel applications. For example, in Figure 1.4, an SC multiplier is implemented with a single AND-gate, in contrast to a binary tree multiplier that requires an AND-gate matrix to generate the partial products, a bit-compression network to compress the partial products to two horizontal bit lines, and an adder. Due to the simple circuitry that SC provides, a lot more multipliers can take up the same chip real estate, which results in a higher degree of parallelism utilisation, and lower energy consumption.

Stochastic Bitstream	**Binary**	**Number of 1s**	**Number of 0s**	**Ratio**	**Decimal Number**
0, 0, 0, 1	0.01	1	3	1/4	0.25
1, 0, 0, 0	0.01	1	3	1/4	0.25
0, 1, 0, 1, 0, 0, 0, 0	0.010	2	6	2/8	0.25

Table 1.1: A table of different stochastic bitstreams and their values.

While SC has a strong potential for low-cost and highly parallel implementations, there are some drawbacks. One of the main problems is the large number of computational iterations required to achieve an acceptable accuracy. There are three main reasons for the large number of iterations in SC computation, namely random fluctuations, correlation between data streams, and inefficient information storage [4, 23]. All these problems contribute to an increase in latency, since the usual solution is to increase the bitstream size, which leads to a longer runtime.

1.1 Research Questions

The thesis aims to investigate how to improve SC multiplication in terms of reducing the number of iterations without significantly increasing the area, and how the multiplier design affects CNN performance. Therefore, the following research questions are in place:

1. **Can the number of computation cycles in an SC multiplier be reduced without significantly increasing the area overhead?** SC multiplication is a potential candidate for edge computing devices, due to its small area usage and low energy consumption. The low-cost SC design would enable low-cost hardware support for highly parallel applications and further proliferation of CNNs on embedded and portable devices. There are some ways to reduce the number of iterations, namely exploiting the redundancy of a bitstream. In unipolar coding, only the $1s$ contribute to the final value in a bitstream, which means that it is possible to ignore $0s$ and terminate the multiplication operation early without affecting accuracy, which improves SC multiplication in the context of ANN and CNN inference.
2. **What would be the best way to organise a set of SC multipliers to accelerate a CNN application?** CNNs are accelerated in several ways and the most common way is to speed up the kernel computation by using an array of multipliers and an adder tree. However, there are also alternative configurations such as a Matrix-Vector Multiplication (MVM). In this configuration, a single weight value is multiplied with multiple feature values. The type of configuration can affect the multiplier efficiency. Therefore, it is important to understand how the configuration affects hardware efficiency.
3. **How is CNN performance affected by the proposed SC multiplier?** CNN performance is defined by four metrics. The first metric is the accuracy rate, which is defined by the number of correct predictions divided by the total number of predictions. If the accuracy rate is too low, then the network is useless. The second metric is latency. The third metric is area consumption and the fourth is energy consumption. These metrics are used to measure the potential implications of the proposed SC multiplier.

1.2 Contributions

In this section, we briefly describe the contributions of this thesis. We divide them into SC multiplier design, stand-alone multiplier performance, and CNN performance. This thesis has the following contributions:

- A new SC multiplier design called StoHej that utilises complementary events to achieve the smallest number of cycles to reach the product. StoHej builds upon the following observations: (i) The multiplication of operands operands residing between [0, 1] results in a product between 0 and the minimum among them. (ii) One can compute the product by either starting from 0 and counting upwards to the product or starting from the minimum among the input operands and counting

downwards. (iii) By using the complement of one operand, the multiplier can count down from the other input operand to the product. (iv) It is more advantageous to use the complement of the input operand when the operand is larger than 0.5 because the complement is $1-p$, and a smaller number means fewer cycles. These features make it possible for StoHej to reduce to worst-case latency from $O(N)$ to $O(\frac{N}{2})$. The detailed design is presented in Section 4.2.

- Three additional StoHej applicable techniques to further optimise its performance in terms of cycle count when utilised in certain applications. The three additional technique are the following: (i) Scaling the input operand, (ii) Utilising multiple Search Areas (SAs), and (iii) Multiple reduction cycles. (ii) Make use of the same complement mechanism for a smaller range, which means instead of using the complement for numbers between $[0.5, 1]$, the multiplier instead checks if the operand is between $[0.25, 0.5]$, which changes the complement equation from $1-p$ to $0.5-p$. This report uses the term Search Area (SA) to refer to this range, which means that multiple SAs checks for multiple ranges. Multiple SAs are used when scaling is not practical for an inference CNN application and a majority of input operands are not larger than 0.5. (iii) Utilisation of multiple reductions, which uses the complement mechanism repeatedly. The number of repetitions is configurable and should be chosen carefully as additional repetitions require additional hardware. Note that the first and second techniques can be applied when a majority of the trained CNN weights are below 0.5. The scaling technique scales the values over 0:5 so that the complement mechanism is utilised more. The additional techniques are presented in Section 4.4.

- The third contribution is the evaluation of StoHej, BISC-MVM, and CSC multipliers in a standalone context. BISC-MVM stands for Binary Interfaced Stochastic Computing Matrix-Vector Multiplication and is the state-of-the-art for SC multipliers utilised in CNN inference applications. The three multiplier algorithms were implemented in C and all possible input combinations were tested for each bitstream size from 8 to 512 bits. In this experiment, StoHej proved to be 3.2x and 5.5x faster than BISC-MVM and CSC, respectively. In terms of accuracy, CSC has the lowest accuracy compared to StoHej and BISC-MVM, which provide the same accuracy, which indicates that StoHej's usage of complements did not cause any accuracy degradation. To evaluate area and energy consumption, we implemented StoHej and BISC-MVM on an FPGA. StoHej has a larger area than BISC-MVM, which was expected to the additional hardware required to compute with the complement. However, when considering the Area-Cycles-Energy Product (ACEP), StoHej had 2.9x smaller ACEP when compared to BISC-MVM when 75% of the weight values are over 0.5. We also performed an analysis to determine the number of weight values that need to be over 0.5 for the single SA StoHej to be cost-effective, which indicates that for smaller bit-sizes at least 25% needs to be over 0.5, however as the bit-sizes increases the percentage decreases. For 9 bits precision, less than 1% need to be over 0.5 to be cost-effective when compared to BISC-MVM.

- The fourth contribution is the evaluation of the proposed multiplier in a CNN context. StoHej and BISC-MVM were tested in a LeNet-5 network with the MNIST dataset. The different multiplier algorithm such as StoHej, BISC-MVM, CSC, Fixed-point, and Floating point was implemented on a GPU to obtain accuracy-rate for the different designs. In addition, StoHej with different SAs and scaling were also tested, to see their effect on accuracy rate and latency. To obtain energy and area data, StoHej and BISC-MVM were organised in a Multiply-Accumulate (MAC) array that was implemented on an FPGA. The experiment showed that StoHej with 3 SAs had a speedup of 1.7x and no loss in accuracy compared to BISC-MVM. StoHej has a larger area usage, however, when using the Area-Delay Product (ADP), StoHej had 30 % reduction when compared to BISC-MVM. The Area-Delay-Energy Product (ADEP) of StoHej is $2.3x$ smaller than BISC-MVM. StoHej's area overhead when compared to BISC-MVM is 29.5%, however StoHej also had a latency reduction of

38.7% and an energy reduction of 40.0%.

1.3 Outline

The thesis has the following outline: Chapter 2 provides necessary background information about SC, ANNs, and CNNs. Chapter 3 explores relevant SC literature. Chapter 4 describes the proposed multiplier design (StoHej). Chapter 5 documents a set of experiments, and finally, Chapter 6 concludes the thesis with a discussion and possible future work.

Chapter 2

Background

This chapter provides background information related to Stochastic Computing (SC), Artificial Neural Networks (ANNs), and Convolutional Neural Networks (CNNs). The first section explains SC fundamentals and how it differs from conventional arithmetic. The second section describes how conventional SC arithmetic implements multiplication and the benefits of SC arithmetic in the context of highly parallel applications. Section 2.1.1 discusses how SC number generation affects the accuracy, the number of clock cycles per computation, and the relationship between accuracy, latency, and information efficiency. The section also discusses how improving the quality of stochastic numbers leads to a more efficient SC multiplier for CNN applications. The second part of the chapter provides an overview of ANNs and CNNs, how they work, why CNNs are the state-of-the-art at image-related applications, a computational analysis of a simple CNN application, and how SC multipliers can improve hardware efficiency for CNNs.

2.1 Stochastic Computing

Stochastic Computing (SC) is an alternative computing paradigm that represents numbers by streams of random bits [44], where a single bit being 1 has a probability of p and the probability of a bit being 0 is $1-p$. For example, an 8-bit stream with the sequence: $(0,1,1,0,0,0,1,1)$ represents $\frac{4}{8}$, as the numerical value of a stream is derived by dividing the number of 1s by the total stream length. Therefore, multiple sequences can represent the same numerical value, as the order does not impact the evaluation of the stream [3]. $\frac{1}{2}$ can be represented by different sequences such as: $(0,1,0,1)$, $(1,0,0,1)$, $(0,0,1,1,0,1,0,1)$, as the ratio of 1s to 0s for the mentioned sequences is $\frac{1}{2}$. A longer bitstream has a higher resolution, however, an SC circuit needs to evaluate more bits, which increases the computation time. This type of bitstream has several names, such as stochastic number, bit sequence, stochastic bitstream, and bitstream. The report uses these terms interchangeably and refers to a bitstream with random bits or pseudo-random bits.

As only the ratio of $1s$ and $0s$ determines the numerical value for a stochastic number, many codes or combinations of bits represent the same value [17]. This representation has two consequences for SC circuit design, the first consequence is the arithmetic simplification, as all bits in a stochastic number share one unitary weight. The second consequence is that a longer bitstream is required to store the same information compared to conventional non-redundant binary arithmetic, where every value is represented by one unique code or bit combination. The difference in information storage efficiency can be observed by comparing the accuracy of N-bit numbers in both systems. Accuracy for SC is equal to $\frac{1}{N}$, where N is the bitstream length, however, the accuracy for conventional positional number representation is equal to 2^{-N} for N-bit fractional numbers. From these two consequences, we can conclude that numerical representation

for both systems has benefits and drawbacks. SC's numerical system is better for synthesising complex arithmetic with few logic gates, due to lacking of carries, however, conventional arithmetic has better information efficiency, which increases accuracy for the same amount of bits. For example, in a 5-bit fractional number the Unit in the Last Position (ULP) is equal to 2^{-5} and to obtain the same precision in SC, the stochastic number needs 32 bits as $\frac{1}{32} = 2^{-5}$.

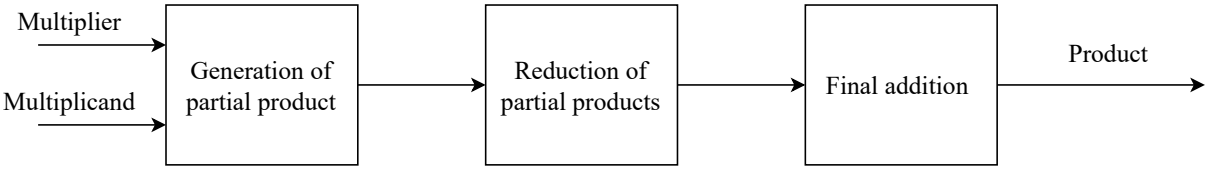


Figure 2.1: *Block diagram of a generalised binary multiplier.*

As the thesis focuses on multiplier design, we compare a conventional tree multiplier with a standard SC multiplier. A conventional binary multiplier as seen in Figure 2.1 consists of three parts: generation of partial products, reduction to 2 of the partial products, and carry propagate addition [7]. The partial products are produced in parallel by using an AND-gate matrix with N^2 elements. Every bit of the multiplicand and the multiplier gets AND-ed together and shifted to the corresponding position determined by the combined weight of the bits. To derive the final product, the circuit needs to add the partial products together. They are first reduced to two numbers, by using Full Adders (FAs) and Half Adders (HAs), and then a conventional adder produces the final product. From a complexity analysis, we can derive that the generation of the partial products is $O(1)$, as there is no dependency between the bits. However, the reduction part has $O(log(N))$ layers and $2log(N)$ gate levels. A tree multiplier can perform multiplication in one or two clock cycles, however, due to propagation delays and gate delays, the clock period is rather large. The high propagation delay of a binary multiplier means that the circuit needs to run at a rather low frequency. Another drawback is the large chip real estate it requires, which limits the number of multiplication units that can be implemented on a hardware platform. As such it is not suitable for applications that have heavily constrained hardware resources and exhibit high parallelism, such as CNN inference on IoT devices, for which exploiting parallelism is vital to achieving high performance and practical utilisation.

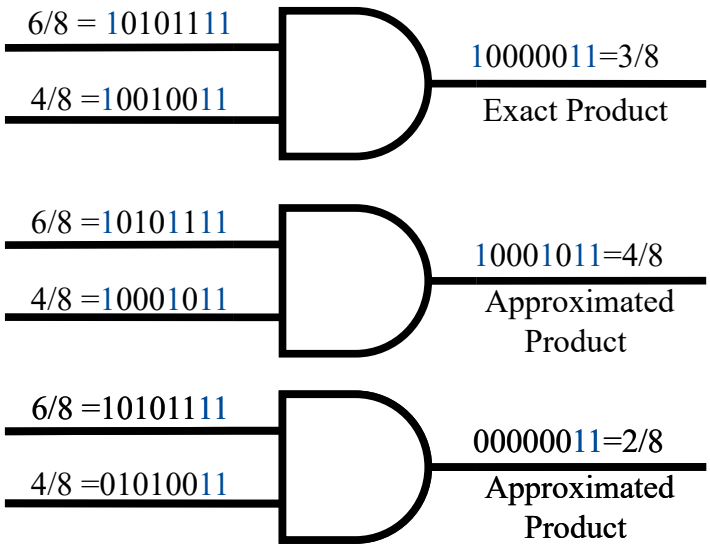


Figure 2.2: *SC multiplication with* $\frac{4}{8} \cdot \frac{6}{8}$ *with different results.*

SC implements multiplication differently due to the non-positionality of stochastic numbers, which

allows the utilisation of a single AND-gate to perform the multiplication of two stochastic bitstreams [44, 8]. To further explain, assume two independent ideal stochastic numbers A and B, then the AND-gate's output probability is $A \cdot B$ [44]. An example of this type of multiplication is presented in Figure 2.2 with three different results. The upper part illustrates an ideal SC multiplication resulting in the exact product value, while the other results are approximate products. Input A is equal to $\frac{6}{8}$, which is confirmed by the 6 $1s$ and 2 $0s$ in the bitstream. Input B is equal to $\frac{4}{8}$ because the stream has 4 $1s$ and 4 $0s$. As seen in Figure 2.2, the output of the AND-gate is $\frac{3}{8}$, which is due to the fact there are three instances when both PA and PB are 1. The product is exact and the AND-gate behaves like a multiplier. However, due to the random nature of SC streams, it is possible to get an approximated result, which is the case in the middle and bottom parts of Figure 2.2. In the middle and bottom cases the 1s in B are occurring in different orders resulting into 4 and 2 overlapping 1s, thus the output is 4/8 and 2/8 instead of 3/8, respectively. In some applications, approximated results are acceptable, which means that the target application needs to have some resilience against inaccuracies for SC to work well. The benefits of this design are fewer gates, smaller propagation delay, and a higher degree of parallelism. However, as a consequence of the random nature, there are computation inaccuracies and lower precision when compared to conventional arithmetic. Another consequence is the number of clock cycles needed to compute a multiplication. Clock cycle in this context refers to the processing of a single bit and not of the entire bitstream. In the example shown in Figure 2.2, the multiplication takes eight clock cycles, which is acceptable in some cases. However, for larger precision, the required clock cycle number can become unreasonably high. For example, to double the precision of a binary number, we only need to add one more bit, while in contrast for SC, we need to double the stream length to achieve the same resolution, which results in a clock cycle count doubling. While there are many more processing cycles for SC, the clock period is significantly shorter than the one of a conventional tree multiplier and thus, the multiplier can run at a higher clock frequency. This analysis clearly suggests that SC would be a better option for parallel applications that do not require high precision, such as CNNs, as smaller multipliers can utilise more of the available parallelism. A tree multiplier has better precision, but fewer multipliers can be instantiated on a computation platform, which limits the application available parallelism exploitation. Note that in this section, we assume ideal stochastic numbers, while, in reality, many non-ideal situations may occur in stochastic bitstreams. For some operations, the non-ideality can heavily compromise the operation or change it to a different one.

Before discussing approaches to increase accuracy and decrease latency, consider the entire SC datapath, shown in Figure 2.3. The datapath includes the conversion of binary numbers to a stochastic representation, the arithmetic operation, and finally, the conversion of the resulting stochastic number to binary [8]. Since SC interprets numbers as probabilities, and probabilities are in the 0 to 1 range, the binary inputs to a Stochastic Number Generator (SNG) are also in the range of 0 to 1. Therefore, the input operands are fractional numbers ranging from 0 to 1. The SNG is the first part of the stochastic datapath and is responsible for converting binary numbers to stochastic bitstreams. A conventional SNG compares the input operand to a random source. If the binary value is greater than the random source, the comparator outputs 1. In the other case, the comparator outputs 0. For every clock cycle, the comparator outputs one bit of the bitstream. The goal of an SNG is to output a bitstream that has an identical value with the input value. However, sometimes this does not happen due to random fluctuations in the Random Number Generator (RNG) [23]. The random source can be a True Random Number Generator (TRNG) or a Pseudo-Random Number Generator (PRNG). The conventional choice for a random source is a Linear Feedback Shift Register (LFSR). The second step in the stochastic datapath is the actual arithmetic operation. SC representation is not radix positional thus all bits have unitary weight and, as such, there are no carries to be considered in the calculations.

The third and final part of the SC datapath is the conversion from a stochastic number to a binary

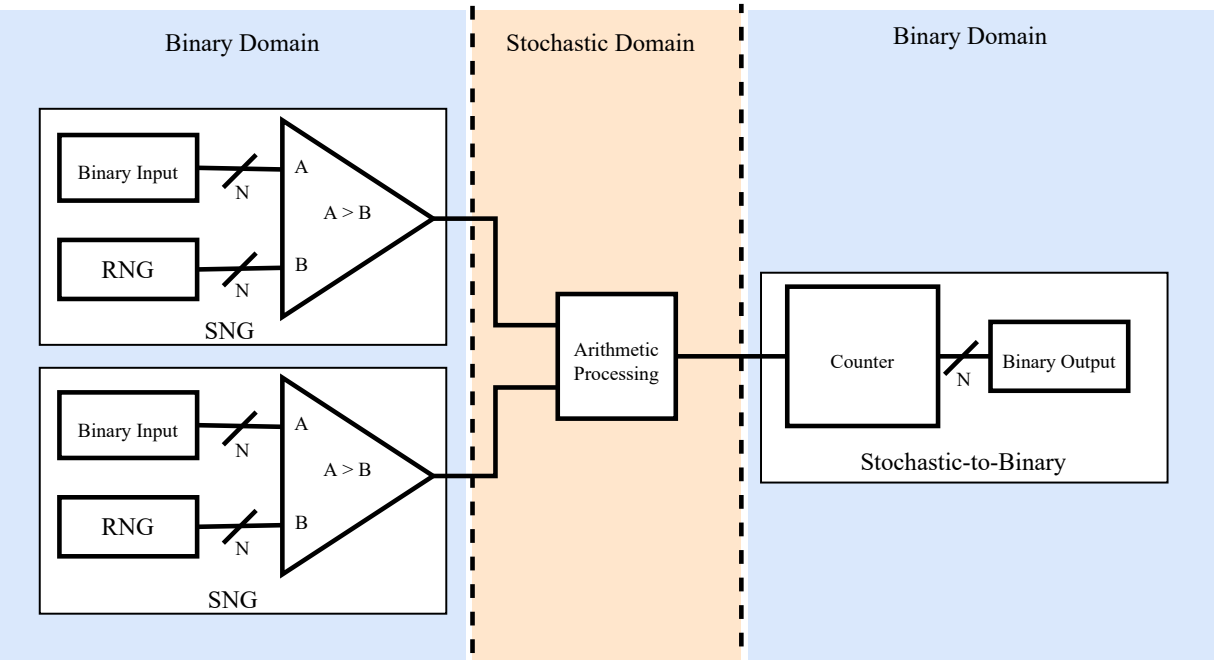
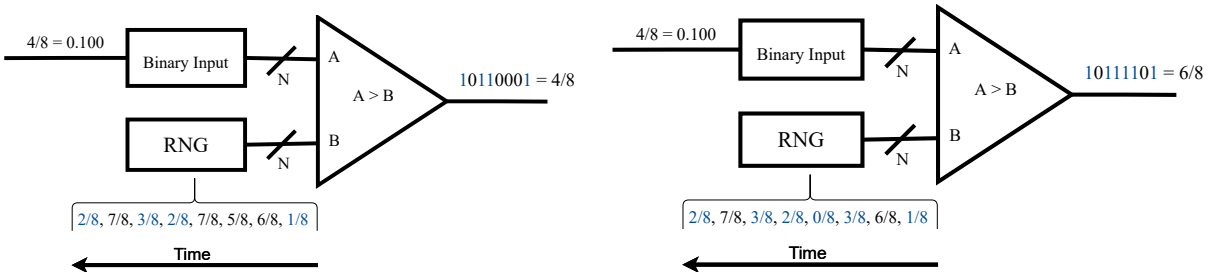


Figure 2.3: *Data path of a Stochastic Computing (SC) system.*

number. To derive the value of a stochastic number, we need to know the number of bits that are 1 and the total bitstream length, which is already known at design time. Thus, it is sufficient to use a counter that increments for each incoming 1. The counter starts at 0 at the beginning of the computation and after each clock cycle, the counter value increases if the current bit of the stream is one. The most critical part of the datapath is the SNG unit because it affects the quality of the generated stochastic bitstreams. A poor quality bitstream can lead to increased errors that must be compensated for with longer stream lengths that result in longer runtime.

2.1.1 Stochastic Number Inaccuracies



(a) *Accurate Conversion in an SNG.* (b) *Inaccurate Conversion due to random fluctuations.*

Figure 2.4: *The effects of random fluctuations on stochastic bitstream conversion.*

The inaccuracies in SC are due to several reasons. The first reason is the inherent fluctuations that occur in RNGs [4]. The effect of fluctuations can be quite severe, as the obtained stochastic bitstream may represent the wrong value. For example, consider a sequence of eight random numbers generated during the conversion of $\frac{4}{8}$ into a stochastic bitstream, see Figure 2.4a. Half of the random sequence must be less than $\frac{4}{8}$, to produce the correct bitstream. However, due to the random nature of RNGs, it is possible to generate other ratios. In Figure 2.4b, six numbers are below 0.5 and thus represent the wrong ratio or value. This effect is present in all probabilistic occurrences, e.g., the flip of a coin. The probability of either heads or tails falling is 0.5. However, for a small sequence like the one in Figure 2.4, it is possible

to obtain a ratio that strongly favours heads or tails. Therefore, a common solution is to increase the bitstream length, as in this way the value of the stream gets closer to the desired value. The error due to random fluctuations decreases when the binary number is closer to 0 or 1. For a value close to 0, a less likely combination is required to produce a false bitstream as much of the random sequence must be smaller than the small binary value, which is rarer. This is also true for values near 1 since a large part of the random sequence must be larger than the large binary value. Another way to think about this is the possible combinations of $1s$ and $0s$ in the stochastic bitstream for 0, 0.5, and 1. There is only one way to combine a stream that consists only of $0s$. This is also true for a stream consisting of only $1s$. In these cases, random fluctuations do not affect the representation accuracy. However, there are many ways to order a stream that consists evenly of $1s$ and $0s$. For example, all 1s can be placed at the beginning and then all $0s$ or a uniform pattern that alternates between 1 and 0. Researchers have studied this problem and proposed alternative solutions [4, 23, 44, 31]. Chapter 3 discusses possible solutions to reduce the impact of random fluctuations.

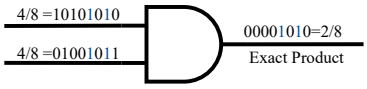


Figure 2.5: SC multiplication with uncorrelated streams.

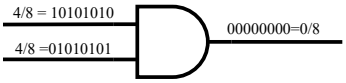


Figure 2.6: SC multiplication with correlated streams resulting in $\frac{0}{8}$.

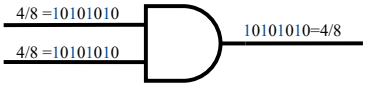


Figure 2.7: SC multiplication with correlated streams resulting in $\frac{4}{8}$.

The second reason for SC inaccuracies is the correlation between the two input streams [4, 23]. SC can only compute accurately if the input data streams are sufficiently uncorrelated or independent. To demonstrate the effects of correlation, consider a stochastic multiplication of two bitstreams with corresponding to $\frac{4}{8}$. The exact product is $\frac{2}{8}$, see Figure 2.5, but the possible product value ranges from $\frac{0}{8}$ to $\frac{4}{8}$, see Figure 2.6 and 2.7. For example, the streams $S1 = (1, 0, 1, 0, 1, 0, 1, 0)$ and $S2 = (0, 1, 0, 1, 0, 1, 0, 1)$ would lead to $\frac{0}{8}$, while $S3 = (1, 0, 1, 0, 1, 0, 1, 0)$ and $S4 = (1, 0, 1, 0, 1, 0, 1, 0)$ leads to $\frac{4}{8}$. This example illustrates that the accuracy of SC computation for a given precision is dependent on random fluctuations and correlation. Correlation induced inaccuracies can become quite significant for SC datapath with high arithmetic depth. The arithmetic depth is the number of cascaded or serial operations performed in the stochastic number domain. For example, a single multiplication has an arithmetic depth of one. While this equation: $p1 \cdot p2 + p3$, has an arithmetic depth of two because the circuit must first compute $p1 \cdot p2$ before adding $p3$. After each operation, the streams start to correlate with each other and as such circuits with high arithmetic depths need to make use of longer streams to compensate for correlation. The inaccuracies of SC significantly increase the number of computation cycles, since the conventional solution is to increase the length of the stochastic bitstreams.

Steps between 0 to 1	**Bits required in binary representation**	**Bits required in SC**
4	2	4
8	3	8
16	4	16
32	5	32
64	6	64
128	7	128
256	8	256
512	9	512
1024	10	1024

Table 2.1: Information efficiency for binary and stochastic representation respectively.

However, apart from inaccuracies, SC's inefficient number representation is another reason why SC

calculations require such a large number of computation cycles when compared to conventional methods. Stochastic numbers are usually between $[0, 1]$, but the number of bits in a stochastic bitstream determines the representation precision. Table 2.1 describes how many bits are required to represent a different number of steps or precision in binary and stochastic representations. For example, to have a resolution of 16 steps, the stochastic bitstream size must be 16 bits. This is in contrast to a binary number, which requires only 4 bits for the same resolution. The resolution efficiency can be generalised for SC as N bits to represent N steps between 0 and 1, while a binary number requires only $\log_2 N$. In summary, the long latency of SC is related to three factors: random fluctuations, correlations between streams, and inefficient information encoding. Addressing these issues would improve SC multiplier efficiency and make SC more viable for CNN applications.

2.1.2 Handling Signed Stochastic Numbers

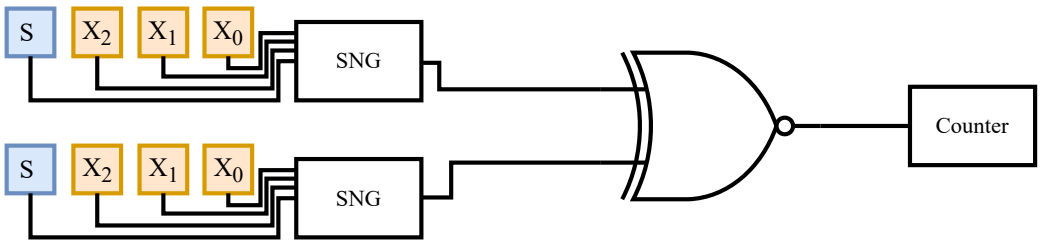


Figure 2.8: *SC bipolar multiplication.*

SC numbers usually fall in the range of $[0, 1]$ and this range is called unipolar encoding, but many applications including NNs operate on negative numbers. Therefore, bipolar encoding was introduced to represent numbers between $[-1, 1]$ [26]. Bipolar encoding uses the function $y = 2x - 1$, to rewrite values from $[0, 1]$ to $[-1, 1]$. The steps between consecutive numbers also change from $\frac{1}{N}$ to $\frac{2}{N}$. Half of the steps are reserved for the negative range. The processing of the bitstream also changes, because now the arithmetic logic and the stochastic-to-binary converter must process $0s$ as well. Each 1 contributes $\frac{2}{N}$ and each 0 contributes $-\frac{2}{N}$. Figure 2.8 illustrates a bipolar multiplication that takes two input operands encoded in 2's complement representation. To implement multiplication with bipolar encoding, an XNOR-gate is used because multiplication with the same sign outputs 1, while multiplication with different sign outputs 0 which decreases the number. Unipolar encoding has several advantages over bipolar encoding. The first advantage is that the number 0 is free of random fluctuations, while in bipolar coding, half of the bits must be $1s$ and the second half must be $0s$, which means that in this case multiplying by 0 can lead to inaccurate results. As such, the processing of a typical image recognition CNN input that includes many zero-valued pixels may result in many unnecessary errors. These additional errors occur because the 0.5 in the unipolar encoding has been rewritten to 0 in the bipolar encoding. The second advantage of unipolar encoding is that $0s$ does not contribute anything to the output values, and thus it is possible to skip $0s$ and terminate the operation early while bipolar encoding requires a complete evaluation of the bitstream since both $0s$ and $1s$ are contributing to the value.

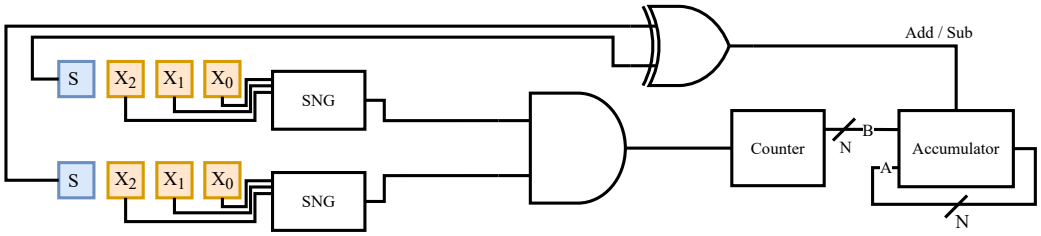


Figure 2.9: *SC unipolar multiplication with sign extension.*

However, it is possible to implement signed multiplication with unipolar encoding, but this requires logic outside the stochastic domain [46]. Figure 2.9 depicts a unipolar SC circuit with sign extension. The binary values are in sign-magnitude representation. The sign bit represents the polarity of the number i.e., if the sign bit is 1, the number is negative and if the bit is 0 the number is positive. The remaining bits are the magnitude or absolute value. This is different from 2's complement and 1's complement, where additional decoding is required to get the magnitude because the negative number is a complement of the positive number. An SC multiplier operates on the magnitude of each operand. The XOR-gate processes the signs of the two operands. If the signs are the same, the result is positive, otherwise, it is negative. The accumulator is fed with the magnitude of the product and the sign of the product determines whether the accumulator should add or subtract the magnitude. This solution retains the advantages of unipolar encoding and allows for negative number representation and processing at the expense of insignificant (one XOR gate) hardware overhead, while providing a 10x times better accuracy than the conventional bipolar SC multiplication [46].

2.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) belong to a computational approach that is loosely based on brain neuron behaviour [1, 11]. Figure 2.10a illustrates a biological neuron with dendrites receiving signals, a cell body to process signals, and an axon to transmit the result. Equivalent parts can be seen in the artificial neuron illustrated in Figure 2.10b, where there is an input stage similar to dendrites, a processing stage, and an output stage. The biological neurons are connected via synapses that can pass electrical or chemical signals in a network as seen in Figure 2.10c. These signals are modelled similarly in ANNs, where each connection between artificial neurons can send signals to other neurons as suggested in Figure 2.10d. ANNs are used in a wide range of applications and are typically utilised for problems for which it is difficult to define exact mathematical solutions [1]. ANNs can approximate such a mathematical solution via a training process, where the synaptic weights are adjusted for each connection [34]. The training dataset contains input and corresponding output pairs. For example, the input can be an image and the output a label. Through the training process, weights are set to initial values, which are usually randomly, and are further adjusted such the network's response corresponds to the expected one. The weights are refined based on the difference between the network's prediction and the correct output. The difference represents the error and after a certain number of iterations, the network response is similar to the expected one. This type of training is called supervised learning.

Another benefit of ANNs is the ability to learn from examples without modifying the topology of the network. To explain this further, consider an image recognition task that classifies whether an image contains a dog or a cat as depicted in Figure 2.11a and 2.11b. Here the weights of the network have been trained to provide the correct response for a dataset that contains cat and dog images. These images from the data set have been manually labelled "cat" or "dog". During the training process the network automatically generates characteristics from the examples, without any prior knowledge of cats or dogs.

As a consequence, the same network can be reused for different image recognition applications by training it on a different data set, such as images of cars or birds.

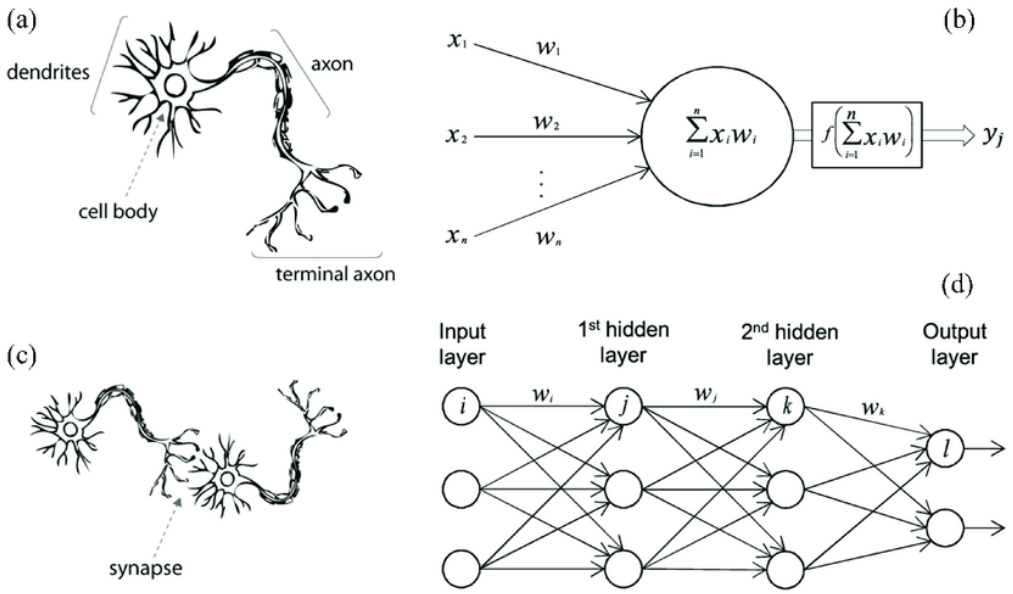
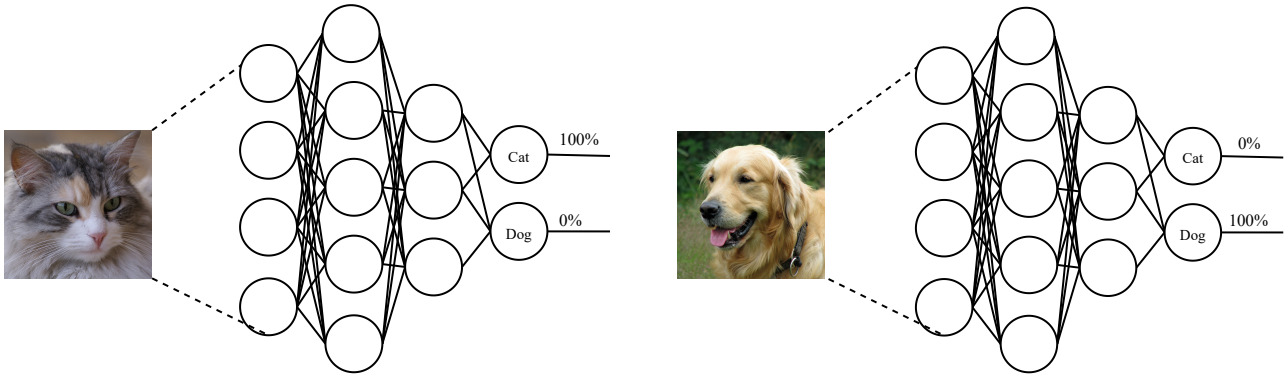


Figure 2.10: *An illustration of biological neurons and artificial neuron: a) human neuron; b) computational model of a neuron; c) biological synapse; d) Artificial Neural Network with four layers, adapted from [33].*

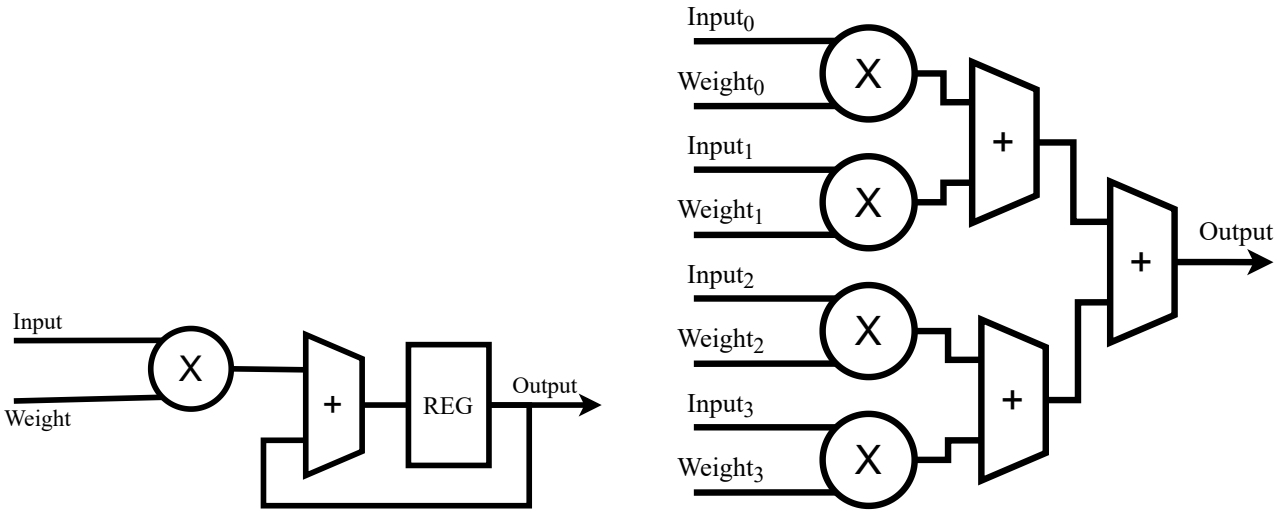


(a) *Trained ANN for predicting images with cats.* (b) *Trained ANN for predicting images with dogs.*

Figure 2.11: *ANN used for image recognition of cats and dogs.*

Once trained, the synaptic weight values are fixed and in order to derive ANN's reaction to a new not in the training dataset, input one has to evaluate the response of all artificial neurons in the network. This requires the calculation of the weighted sum of all neuron inputs and the application of an activation function, which is usually a nonlinear function such as Rectified Linear Unit (ReLU), Sigmoid or hyperbolic functions such as tanh (see Figure 2.10b). This so-called ANN inference can be quite computationally intensive depending on the size of the network. The required number of multiplications and additions is depending on the number of layers, the number of neurons in each layer, and each neuron fan in (number of inputs).

There are several ways to accelerate the ANN inference process by making use of neuron, layer, and inter-layer parallelism [5]. First, there is available parallelism in the neuron response calculation, as one can observe in Figure 2.12, where two neuron response evaluation hardware implementations are illustrated. Figure 2.12a depicts a serial implementation, which requires N multiply and accumulate cycles for the



(a) *Serial computation of a neuron.* (b) *Parallel hardware of a four-input neuron.*

Figure 2.12: *Serial and parallel implementation of the artificial neuron.*

evaluation of an N input neuron, while Figure 2.12b shows a parallel implementation that can evaluate a 4-input neuron in one cycle. In the more general case, N-multipliers and N-1 adders can evaluate the response of one N-input neuron in one cycle. The second way to speed up is to exploit parallelism between neurons. The neurons in a layer are independent of each other thus, it is possible to parallelly evaluate multiple neurons in a layer by replicating the neuron computation hardware. The exploitation of both neuron and layer parallelism requires hardware replication and as such their application is limited by the available chip real estate budget. In view of our previous discussion, it is clear that the low-cost SC hardware allows for higher utilisation of the available parallelism at both neurone and layer level. The last parallelism resort can be enabled by computation platform pipelining, which allows layers to operate in parallel on different input data. An ANN accelerator developer can choose different levels of hardware unrolling for different levels of parallelism. For example, a developer may choose not to unroll the internal neuron computation, but completely unroll the computation within a layer, and so on. This is a balancing act between hardware requirements (available resources) and runtime performance. A fully parallel implementation would be the fastest but would require a lot of hardware. In practice, due to the high degree of available parallelism, many ANNs are mapped on parallel hardware such as GPUs [29]. This is done to reduce latency and increase throughput, which allows for practically acceptable training and inference times.

2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a particular type of ANN that have provided excellent results for Computer Vision (CV) applications [43]. The architecture of a CNN is similar to that of a generic ANN, but it includes convolutional layers and pooling layers [19]. The convolutional layer performs a dot product, see Figure 2.13, which is subsequently fed into an activation function. The convolutional layer is usually followed by downsampling, see Figure 2.14. Downsampling involves selecting multiple inputs and attempting to approximate them by a single output value. The most common methods for downsampling are average pooling and max pooling. In average pooling, the average value is calculated for the input matrix and reported for output. Max pooling takes the maximum value for the input matrix and sets it to the output value.

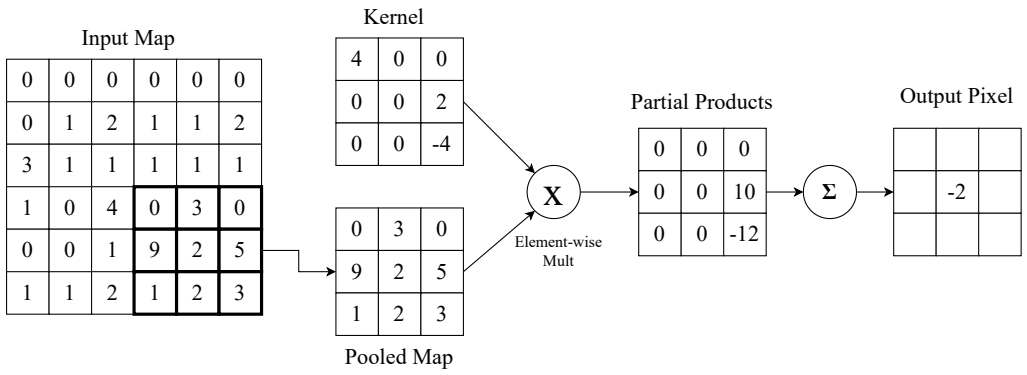


Figure 2.13: *Visual representation of the convolutional layer.*

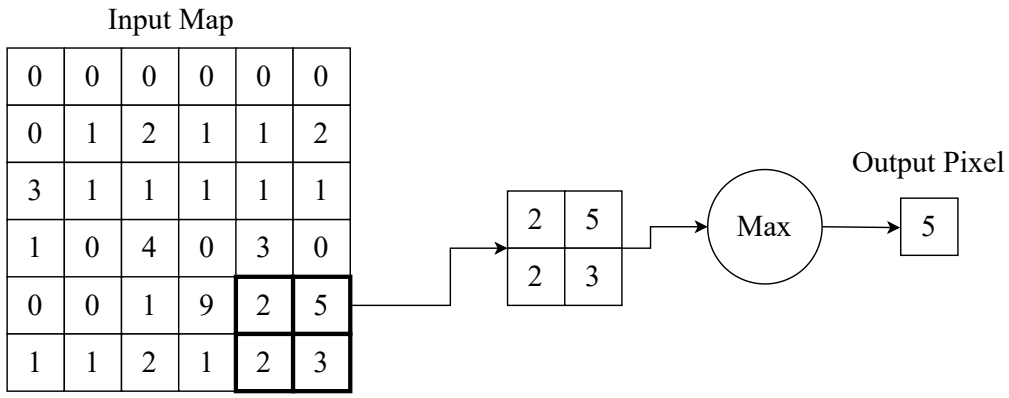


Figure 2.14: *Visual representation of the downsampling or pooling layer using max pooling.*

CNN's response evaluation is also computationally intensive. The convolutional layer takes over 90% of the computations [13] and thus it is important to identify ways to accelerate it. A convolutional layer is slightly different from an ordinary NN layer, which is also called the Fully Connected (FC) layer. The input of a convolutional layer is an input image, however, it is also called an input feature map, and the output is the convolved output feature map. A smaller matrix of weights, usually called a kernel or filter, is multiplied by a smaller same size section of the input map. The weighted inputs are then summed. A bias term is then added to the weighted sum and passed through an activation function. The output of the activation function becomes the convolved pixel. The process is repeated as the kernel slides over the input map at a given stride. Stride defines how the kernel moves over the input feature map. A stride of one means that the kernel moves one unit after each kernel computation. If the stride is three, the kernel moves three units at a time. Padding is used when part of the kernel is outside the input map to ensure that the computation provides no errors. It is also common to have multiple filters, which means that the process is repeated, but with different weights. If the input map has a depth greater than 1, the kernel process must also be repeated for that depth. Map depth can refer to the colour channels of an image or the number of channels produced after a convolutional layer, for example, if a convolutional layer has 32 filters, then the output feature map has a map depth of 32. Because of all these factors, convolution is quite expensive to compute when compared to other layers, such as fully connected or downsampling layers. There are also many different ways to accelerate the convolutional layer. For example, the kernel computation itself can be accelerated by fully unrolling it out in hardware. This is usually implemented by an array of multipliers and an adder tree. The second type of acceleration is to compute multiple output feature values simultaneously. The third type of speedup is to process multiple input feature maps simultaneously. A fully parallel version of a convolutional layer is quite expensive in terms of hardware

and to get a grasp on

Convolutional parameters			
Parameter	**Description**	**Parameter**	**Description**
X	Width of input feature map	KW	Width of kernel
Y	Height of input feature map	KH	Height of kernel
Z	Depth of input feature map	S	Stride of kernel
C	Width of output feature map	TC	Width of tile output feature map
R	Height of output feature map	TR	Height of tile output feature map
M	Depth of output feature map	TM	Depth of tile output feature map

Table 2.2: Convolutional layer parameters.

the number of computations a CNN evaluation requires, we consider a LeNet-5 CNN modified for the CIFAR-10 dataset, which stands for Canadian Institute For Advanced Research, see Figure 2.15. The CIFAR-10 dataset consists of $60,000$ images, with $40,000$ images used for training and the rest of them for testing [6]. CIFAR-10 consists of 32 by 32 pixels with three different colour maps and the goal of the network is to classify these images into ten different classes, namely: aeroplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The LeNet-5 has two convolutional layers, two downsampling layers, and three fully-connected layers. The number of operations for the different layer types are defined as:

$$Convolutionallayer = KW \cdot KH \cdot Z \cdot R \cdot C \cdot M$$
$$Pooling\ layer = KW \cdot KH \cdot R \cdot C \cdot M$$
$$Fully\ Connected\ layer = Neuron\ Inputs \cdot Neuron,$$

where the convolutional layer parameters are defined in Table 2.2. The parameters and computation intensity for each layer are presented in Table 2.3. The size of the input maps is written as (X, Y, Z), and the size of the output maps is written as (C, R, M). The parameter column represents the kernel parameters and is written as (KW, KW, M). The fully connected system has no additional parameters and is calculated by multiplying the number of inputs by the number of outputs. The first layer in Figure 2.15 has the following input parameters $(32, 32, 3)$, and the size of the filter is $(5, 5, 6)$. The width and height of the output map are equal to $32 - KH - 1$, so the size of the output map is 28 times 28, since the kernel size for this layer is 5 times 5. The depth of the output map is 6 since there are six filters.

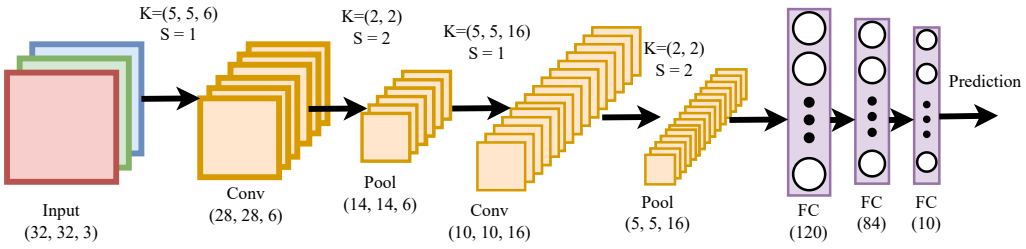


Figure 2.15: *Modified version of LeNet-5 CNN.*

Layers	Input Map Size	Output Map Size	Parameters	Number of Operations	Ratio
Conv 1	(32, 32, 3)	(28, 28, 6)	(5, 5, 6)	352800	53.62%
Pool 1	(28, 28, 6)	(14, 14, 6)	(2, 2)	4704	0.71%
Conv 2	(14, 14, 6)	(10, 10, 16)	(5, 5, 16)	240000	36.47%
Pool 2	(10, 10, 16)	(5, 5, 16)	(2, 2)	1600	0.24%
FC 1	(5, 5, 16)	(120,)	N/A	48000	7.29%
FC 2	120	84	N/A	10080	1.53%
FC 3	84	10	N/A	840	0.13%
All Conv	N/A	N/A	N/A	592800	90.09%
All Pool	N/A	N/A	N/A	6304	0.96%
All FC	N/A	N/A	N/A	58920	8.95%

Table 2.3: Computational analysis of LeNet5 with CIFAR-10 dataset.

The number of operations for the first convolutional is $5 \cdot 5 \cdot 3 \cdot 6 \cdot 28 \cdot 28 = 352800$. Downsampling layer requires $4 \cdot 4 \cdot 14 \cdot 14 \cdot 6 = 18816$ operations. The number of operations for the third layer is $5 \cdot 5 \cdot 6 \cdot 10 \cdot 10 \cdot 16 = 240000$. The next downsampling takes $4 \cdot 4 \cdot 5 \cdot 5 \cdot 16 = 6400$ operations. The fully connected layers take in total is equal to $48000 + 10080 + 840 = 58920$ operations. It can be observed that the convolutional layers account for over 90.09% of the computations. A fully parallel solution for the first layer would require 352800 multipliers, which is prohibitively expensive to implement. From this analysis, we can conclude that CNN accelerators could substantially benefit from a large number of hardware multipliers. However, the amount of multipliers is confined by chip real estate constraints and as such alternative multipliers are of interest. As by their very nature, CNNs do not have strict precision requirements, the utilisation of SC multipliers constitute an attractive alternative due to their extremely low area, which enables the utilisation of a large amount if not all the CNN application available parallelism.

```
for(int m = 0; m < M; m++)
for(int r = 0; r < R; r++)
for(int c = 0; c < C; c++)
for(int z = 0; z < Z; z++)
//UNROLL IN HARDWARE
for(int y = 0; y < KW; y++)
for(int x = 0; x < KH; x++)
{
  om[m][r][c] += im[z][Sr + y][Sc + x]  *
  ↪  w[m][z][y][x];
}
```

(a) *Six loop implementation of a convolutional layer in C.*

```
for(int m_t = 0; m_t < M; m_t += Tm)
for(int r_t = 0; r_t < R; r_t += Tr)
for(int c_t = 0; c_t < C; c_t += Tc)
for(int z = 0; z < Z; z++)
for(int y = 0; y < KW; y++)
for(int x = 0; x < KH; x++)
//UNROLL IN HARDWARE
for(int m = m_t; m < min(m_t+Tm, M); m++)
for(int r = r_t; r < min(r_t+Tr, R); r++)
for(int c = c_t; c < min(c_t+Tc, C); c++)
{
  om[m][r][c] += im[z][Sr + y][Sc + x]  *
  ↪  w[m][z][y][x];
}
```

(b) *Tiled version of a convolutional layer.*

Figure 2.16: *Convolutional layers implemented in C.*

The usual method for accelerating the convolutional layer is to unroll the kernel width and kernel weight. The hardware setup for such acceleration is depicted in Figure 2.17, while the C implementation of the layer is presented in Figure 2.16a. The variables; `im`, `om`, and `w` stand for input map, output map, and weights, respectively. The two kernel loops are unrolled in hardware and compute one output pixel each. The number of operations for the conventional configuration is equal to $M \cdot R \cdot C \cdot Z$.

An alternative configuration is a tiling system, presented in Figure 2.16b. Instead of computing one pixel at a time, a tile of the output map is computed instead [38] and a single weight is multiplied with

multiple input values and the calculation continues until all weights are utilised. The tile is then stored in memory and the next tile can be processed. This configuration is better suited for multipliers with weight value-dependent latency, such as BISC-MVM [40]. Only one weight is used at a time instance and therefore no multiplier has to wait for another multiplier to complete. The number of operations for a tile configuration is equal to $\frac{M}{TM} \cdot \frac{R}{TR} \cdot \frac{C}{TR} \cdot Z \cdot KW \cdot KH$. From these two configurations, we can observe that tiling can achieve a smaller runtime if the tile size is large. If the tile size is equal to the output map size then the only terms that influence are Z, KW, and KH, Z, KH, and KW are often smaller than M, R, and C, therefore the tiling configuration can achieve higher performance as it only needs to iterate through Z, KH, and KW. However, it depends on how many multipliers units can be instantiated on the same hardware platform. SC multipliers have significantly less hardware and energy consumption and therefore can provide inexpensive, but highly parallel hardware support.

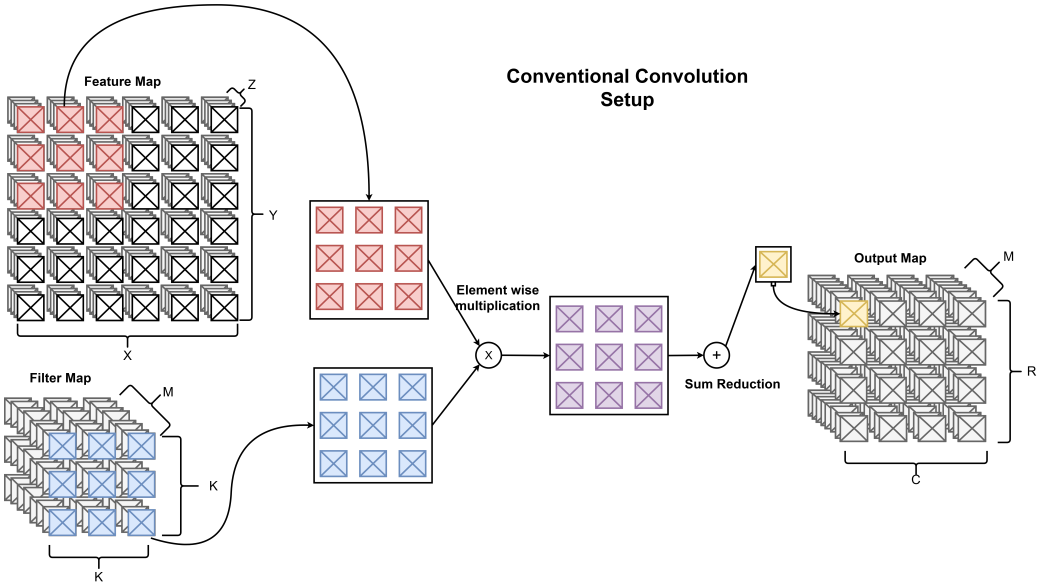


Figure 2.17: *Conventional hardware acceleration of a convolutional layer.*

2.4 Conclusion

This chapter covers background information on Stochastic Computing (SC), Artificial Neural Networks (ANNs), and Convolutional Neural Networks (CNNs). SC is an alternative computing technique that is used for providing low-cost arithmetic for applications that have no strict precision requirements. ANN and CNN do not require the exact calculation of the network's response and can tolerate some level of computational errors. Due to not requiring exact computation, SC can be attractive for ANN and CNN training and inference, because its low hardware cost enables the utilisation of the available parallelism. However, SC suffers from a large number of iterations, which limits the wider proliferation of SC arithmetic on CNNs accelerators. There are three main sources for a large number of iterations. The first source is random fluctuations in generating the numbers, the second is the correlation of input values, and the last is inefficient storage of stochastic bitstreams. The next chapter discusses previous SC related work that tries to reduce the number of iterations by targeting one or multiple of these sources.

Chapter 3

Related Work

This chapter has two purposes. First, we provide an overview of the state-of-the-art attempts to reduce latency of Stochastic Computing (SC) multiplication. The attempts target recognised problems such as random fluctuations in the Stochastic Number Generator (SNG), correlation between bitstreams, and inefficient number representation in SC. Secondly, we compare the different designs in terms of hardware usage, latency, and energy consumption in the context of a CNN application. We identify the strengths and weaknesses of each proposed solution and which design is most suited in a convolutional layer accelerator.

3.1 Reducing Inaccuracies in SNGs

A vital component in an SC multiplier is the SNG, and the accuracy of the SNG directly impacts the accuracy of the SC multiplier. As mentioned in the background chapter, almost all of the non-idealities or errors occur in the Stochastic Number Generator (SNG). The ad-hoc solution to non-idealities is simply increasing bitstream length, which reduces computation errors. However, the number of cycles can become excessive and leads to a longer runtime. Thus, one of the main challenges in SC is improving the SNG in terms of accuracy, as improving accuracy leads to a short stream and faster computation [23].

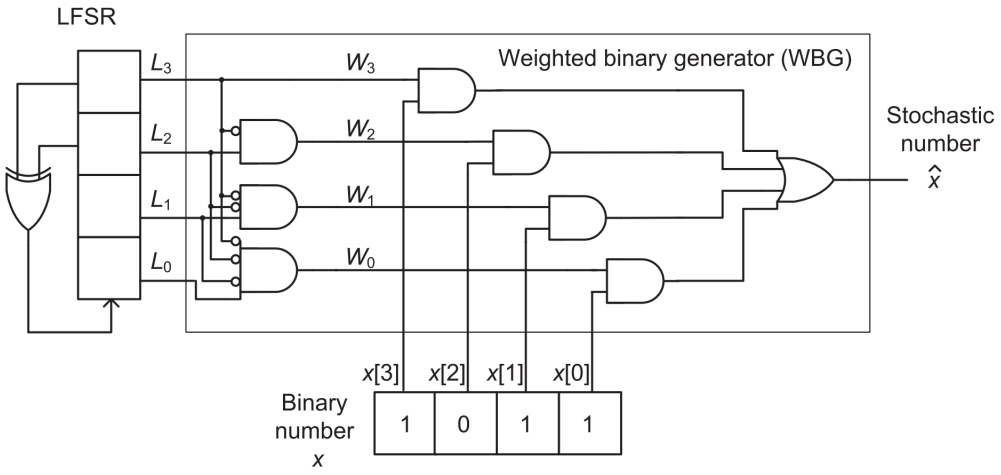


Figure 3.1: *Weighted Binary Generator, adapted from [3]*

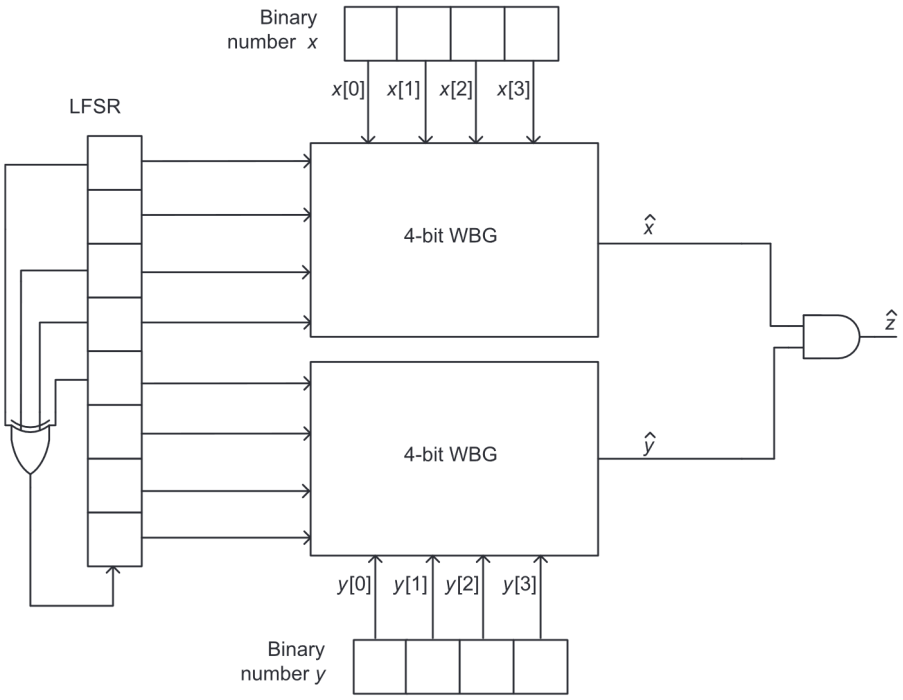
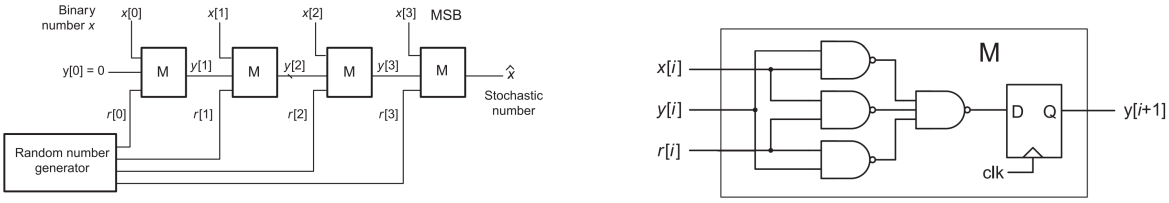


Figure 3.2: *4-bit SC multiplier created using Weighted Binary Generator as SNGs, adapted from [3]*

Weighted Binary Generator (WBG) is an SNG design that was first proposed by Gupta and Kumaresan [20] in 1988. It uses a 4-bit LFSR and replaces the comparator with a weighted binary circuit. Each output bit of the LFSR is connected to a circuit of AND-gates and inverters to ensure that there are no overlapping 1s from the LFSR. The non-overlapping output of the LFSR is then connected to the binary input value using a bit-wise AND-gate. The ANDed outputs are then fed into an OR-gate, which outputs the stochastic bitstream. This solution generates an exact bitstream, which is due to the non-overlapping 1s from the LFSR. However, a drawback of this design is the increase in circuit complexity when increasing bit accuracy. The weight binary circuit is more complex than a simple comparator, which a conventional SNG uses. Thus, a 4-bit SC multiplier constructed built utilising WBGs, as shown in Figure 3.2 uses 348 gates, compared to 228 for a conventional SC multiplier. This is a 52 % increase of gate count, however, the WBG-based multiplier always returns exact results in the 4-bit case.



(a) *The whole SNG.* (b) *Inside a M unit.*

Figure 3.3: *Bit modulator SNG, adapted from [3]*

Bit Modulation Sequence is an alternative SNG design proposed by Van Daalen et al [14], that replaces the comparator with a pipeline of bit modulators, as depicted in Figure 3.3a. The number of bit modulators used in the pipeline is equal to the input bit width. Each bit modulator has three inputs and one output as displayed in Figure 3.3b. The three inputs are modulation bit, pre-stage, and carrier stage. If the modulation bit is 1, then the probability of the output bit set to 1 is $0.5p + 0.5$. However, if the

modulation bit is 0, then the probability is set to $0.5p$. To generate a stochastic bitstream, a sequence of bit modulators is used. The modulation inputs are connected to the binary input number and the carrier inputs are connected to an LFSR. The probability of each bit from the LFSR is set to 0.5 and is uncorrelated. For every clock cycle, a new bit of the stochastic stream is generated by the bit modulation sequence. This design can be used to eliminate the comparator in a conventional SNG design. This proposal only deals with correlations and random fluctuations. To create a 4-bit SC multiplier using the Bit modulator SNG requires 332 gates, which is fewer than an equivalent 4-bit SC multiplier using WBG. However, it is a 46 % increase of gates compared to a conventional 4-bit SC multiplier using conventional SNGs.

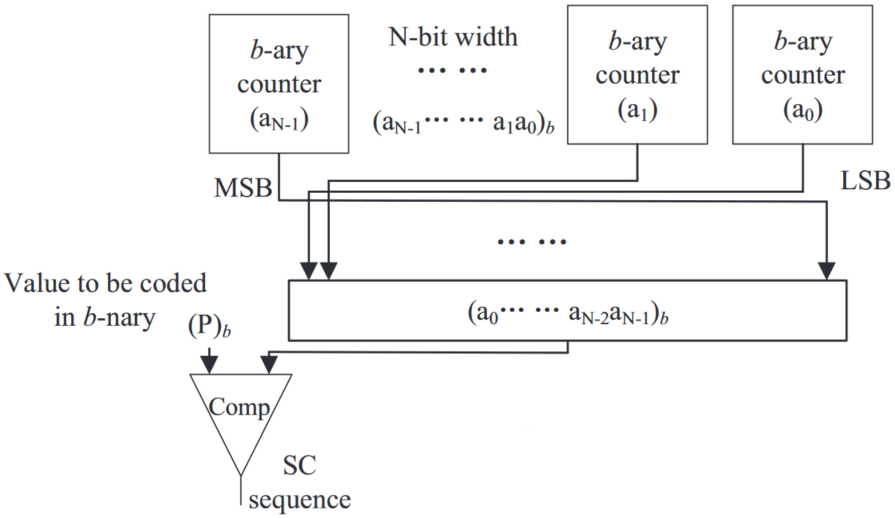


Figure 3.4: *A Halton SNG, adapted from [32].*

Halton Sequence SNG was first proposed by Alahi and Hayes in 2014 and the main idea behind the proposal is to exploit Progressive Precision (PP) in stochastic numbers [2]. This means that if the stochastic computation has a sufficient approximation during the computation, then the computation can end early. The paper also identifies that good PP is dependent on reducing random fluctuations. Thus, they propose to use Low-Discrepancy (LD) sequences because of the uniform spacing of $1s$ and $0s$. The Halton sequence was chosen because the sequence is less complicated and has good performance for similar problems to SC.

Before explaining the hardware for the Halton sequence generator, it is beneficial to understand how the Halton sequence works. The Halton sequence uses coprime numbers for the base of the sequence. For example, one dimension of the Halton can be based on 2. We can generate the sequence for base 2, by dividing the interval 0 to 1 in half, fourth, eighths, etc, which generates the sequence $\frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{5}{8}, \frac{3}{8}$.... To add another dimension to the sequence, we need to use a different which is 3. This sequence is generated by the same procedure as 2, but the interval is divided in thirds, ninths, twenty-sevenths, etc., which generates $\frac{1}{3}, \frac{2}{3}, \frac{1}{9}, \frac{4}{9}, \frac{7}{9}, \frac{2}{9}$.... We can pair up these to get a two dimensional sequence.

The proposed Halton sequence generator as shown in Figure 3.4, implements the sequence by first writing N-th number in the sequence in binary representation. The number is then reversed, which becomes the N-th element in the sequence. For example, the sixth element in the base 2 sequence is $\frac{3}{8}$. We can also derive it by first writing 6 in binary form, which is 110. If we reverse the order and place a decimal point at the start of the number, it number becomes .011, which is equal to $\frac{3}{8}$. Thus, counters are used to step through the Halton sequence. The reversed number is then compared to the binary input

value, which outputs the stochastic bitstream. By utilizing PP, the applications was 225% faster when compared to SC without PP. This proposal tries to solve the two mentioned issues by utilizing LD codes to reduce random fluctuations and using PP to mitigate the issue of information efficiency in stochastic numbers. However, there is a significant area overhead due to the circuit required for the Halton sequence generator. A 2-input 4-bit multiplier with Halton SNGs requires 304 gates, which is a 33 percentage increase of area compared to conventional SNG.

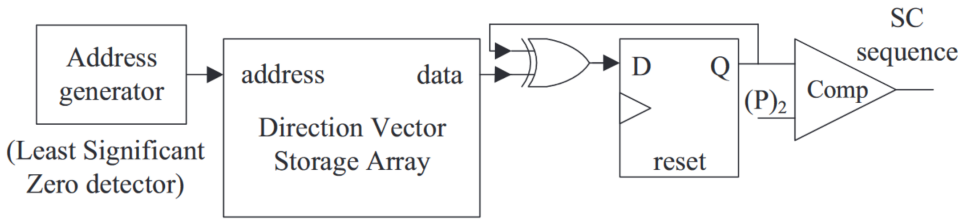
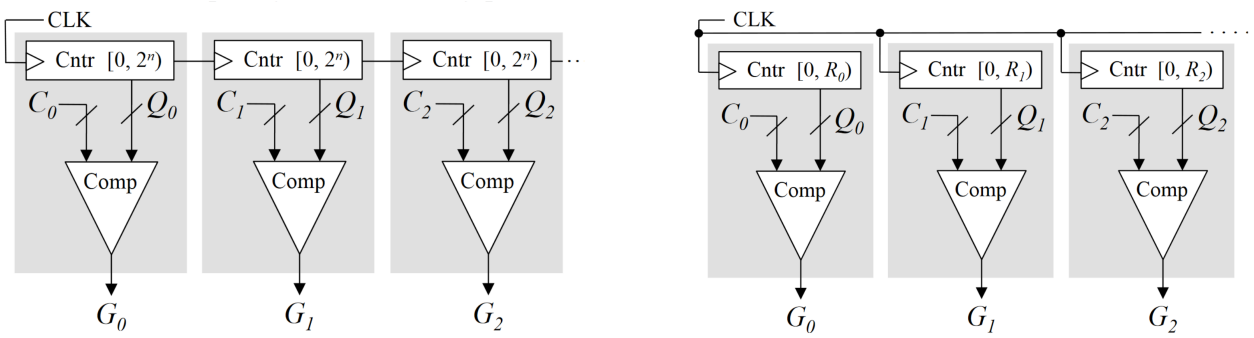


Figure 3.5: *A Sobol SNG, adapted from [32]*

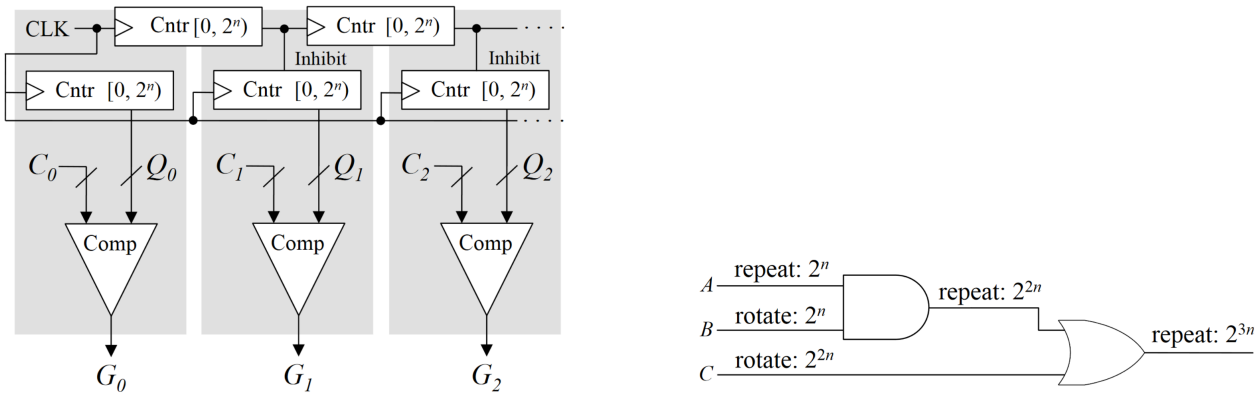
Sobol Sequence SNG is proposed in a paper published in 2016 by Liu and Han [32]. The paper proposes the use of a Sobol sequence generator instead of a Halton sequence generator as it reduces the hardware cost. The SNG consists of three main components, see Figure 3.5. The first component is an address generator that selects the address based on the position of the least significant zero. The generated address is then used to select a direction vector from a Look-Up Table (LUT). The vector is then recursively processed through an XOR-gate and the previous value stored in a register. The value from the register is then compared to the binary value. Finally, the comparator outputs the stochastic bitstream. In the paper, it was shown that the Sobol version has a smaller Root-Mean-Squared-Error (RMSE) than Halton in a multiplier circuit and Bernstein polynomial circuit. Sobol also had better energy per operation than Halton. The disadvantage of Sobol SNG is the need for a LUT for the direction vectors. Overall, a 4-bit SC multiplier using the Sobol SNG requires 296 gates.



(a) *Clock division.* (b) *Relative prime.*

Figure 3.6: *Deterministic SNGs, adapted from [24].*

Deterministic SC is a collection of techniques that uses a deterministic approach for generating stochastic numbers. A paper by Jenson and Riedel [24] describes the reasoning behind Deterministic SC and gives three examples on how to generate and compute with deterministic SC. The SNG can be seen as a bit-selection process from a collection of bits. The binary input value is considered as a collection or array of bits and a random number is used to select the bit. The selected bit is then inserted into the stochastic bitstream. However, it is also possible to use a deterministic source for selecting bits, which means that any arbitrary selection can be created. The paper suggests that the selection $1s$ and $0s$ in



(a) *Schematic of rotation and repeat SNG.*

(b) *Exponential growth of bits.*

Figure 3.7: *SNG based on rotation and repeat method, adapted from [24].*

the bitstream should be proportional to the selection of binary numbers. Thus, the selection index must point to each bit in the binary value for an equal number. Based on this reason, the paper proposed to use Clock Division, Rotation, and Relatively prime bit lengths. Clock division works by repeating one bitstream, while the other bitstream uses a divided clock. The divided clock stream will repeat each bit for the second stream. The generated bitstream can be viewed in Figure 3.6a. The second method rotation involves rotating the bits in a stream, see Figure 3.7a. The rotated bitstream is then computed using a repeated bitstream. This means that each bit in the rotating stream is compared to each bit in the repeated stream. Relatively prime bit lengths are similar to rotation, but other stream lengths are used that are relatively prime as shown in Figure 3.6b. The hardware is relatively inexpensive compared to a conventional SC and the latency is lower, due to fully accurate computations. Relative Prime SNG and Clock Division SNG need 72 gates for a 4-bit multiplier, which is significantly lower than conventional SC. The rotation SNG needs 96 gates for a similar design, which is still significantly fewer than conventional SNG. However, there is one major drawback, which is the expansion of stream lengths after each operation, which Figure 3.7b depicts. For example, a multiplication with two 4-bit- streams grows to 16-bit streams, but a stochastic bitstream generated with a random source does not grow after each computation and instead converges to the values. Downsampling could be a way to manage stream sizes, but deterministic methods such as truncation significantly reduce accuracy. Some researchers have tried using a random source for downsampling [36]. This type of downsampling did not affect the accuracy as much. However, the area reduction is lost due to the hardware required to generate random numbers. This approach reduces random fluctuations due to the use of deterministic methods to generate bitstreams, but due to the mathematical requirement to increase the stream after each multiplication, the latency also grows with it.

3.2 Parallel Processing

While reducing inaccuracies in the SNG enables the use of smaller bitstreams, there are some limitations to this approach. Even if a SC multiplier is exact, the number of computation cycles for larger bitsizes is still significant, which is due to the SC's number representation. Thus, an approach such as parallel processing can further reduce the number of computation cycles by processing multiple bits of the stochastic bitstream at the same time. For example, 8-bit SC multiplier need 256 (2^8) cycles to process the stream, however, with parallel processing we can divide the number of cycles by 2 or more, depending on the level of parallelism utilised.

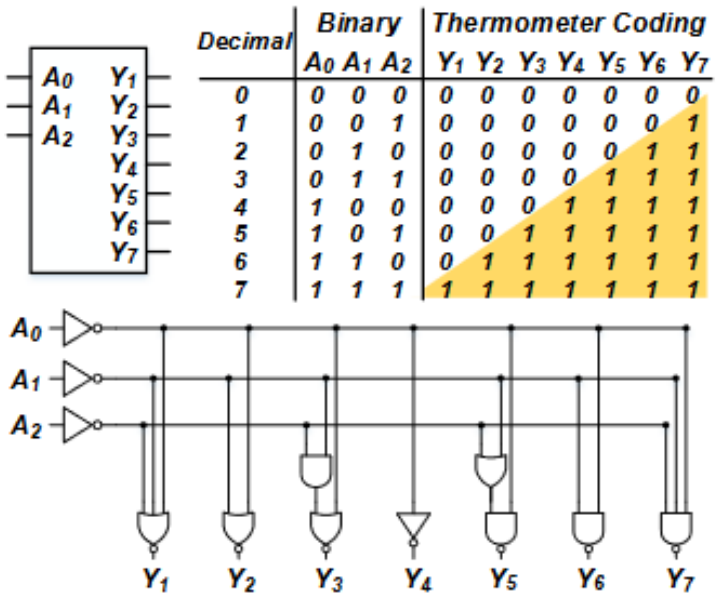


Figure 3.8: *Parallel SNG, adapted from [48]*

Parallel processing is the idea of splitting a stochastic stream into smaller streams and processing them in parallel [48]. Thus, the SNG unit needs to generate at least two-bit streams for one binary input. It also needs many arithmetic processing units, but this is usually insignificant. However, the stochastic-to-binary converter gets a little more complicated because it requires a parallel counter. This significantly increases the hardware requirements. Parallel processing mainly focuses on solving the inherent information efficiency of stochastic numbers. The other approaches try to solve the issue of inaccuracy in the generation to be able to shorten the bitstream, in contrast, parallel processing is mainly focused on increasing the throughput, by processing multiple sections of the stochastic number at the same time. For a smaller number of bits, the SNG design results in a more hardware efficient implementation, as there is no random element. One way to convert a binary number to a parallel stochastic number is to use a binary-to-thermo-encoding as displayed in Figure 3.8, which takes binary input and converts it to a uniform number. Then the number is fed into an AND-gate matrix and the product is then fed into a parallel adder. A 4-bit SC multiplier using a thermo-based SNG requires 196 gates, which is less than conventional SNG. However, the thermo-based SNG can not be shared in a multiplier array, which means that each multiplier needs to duplicate the SNG for every multiplier operand. In contrast to the previous mentioned SNGs, where some subcomponents, such as the LFSRs can be shared over multiple multipliers. Thus, a parallel approach uses fewer gates than a conventional SC for a single multiplier, however, in the context of a parallel application, the hardware cost of the subcomponents in the SNG such as the LFSR can be shared over multiple inputs, which is not possible for a parallel thermo SNG.

3.3 Early Termination

Another technique to speedup SC multiplier computation is early termination. Both early termination and parallel processing are focused on the information inefficiencies, however, early termination approach tries to decrease the number of bits requires to process, while a parallel processing approach tries to increase throughput.

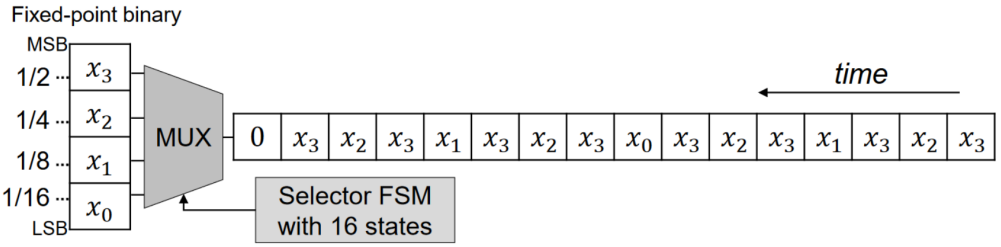


Figure 3.9: *Bit-shuffle SNG, adapted from [40]*

Early termination is the idea of terminating the operation early and saving computation time. There are several criteria for early termination, Halton and Sobol SC use PP to determine when to terminate the operation. However, it is possible to use a mathematical axiom that states that multiplying by two numbers that lie between 0 and 1 yields a product that lies between 0 and the minimum of the inputs. This can be proved for unipolar multiplication by inserting all 1s at the beginning of the stream, which also gives the minimum. Therefore, it is possible to replace one of the SNGs with a down counter. The AND-gate is removed and a single stochastic bitstream is connected to the stochastic to binary converter. This type of early termination is proposed in a paper written by Sim and Lee in 2017 [40]. The multiplier design is called BISC-MVM, which stands for Binary Interfaced Stochastic Computing-Matrix Vector Multiplication. For this design, a conventional SNG can be used, however, the designers proposed an alternative SNG design, which is a Finite State Machine (FSM) that selects one bit from the binary input number for every clock cycle, which Figure 3.9 illustrates. The FSM uses a deterministic pattern that distributes the different weights evenly. For example, the Most Significant Bit (MSB) in a fractional binary number is 0.5, which means that half of the bits in the bitstream are allocated to the MSB. This then applies to the other bits as well, 0.25 gets a quarter of the bitstream and so on. The idea is that only the distribution of 1s affects the accuracy and not the order of the bits. BISC-MVM was tested in a CNN application and was shown to outperform conventional SC computation in terms of latency and CNN accuracy. The disadvantage of the design is that the impact of early determination decreases for larger numbers as these computations require more iterations. Another disadvantage is that this type of technique can only be used for multiplication and not for other arithmetic operations. Nevertheless, it is suitable for CNN applications since an overwhelming part of their computations are multiplications. The idea behind BISC is that certain arithmetic operations are more efficient in SC than others. For example, SC multiplication is more efficient than SC addition. Sim and Lee also used an alternative configuration to speed up the convolutional layer. Instead of speeding up the kernel loops, they instead choose to speed up along the dimensions of an output map tile. This creates a general convolutional layer accelerator that can be used for arbitrary dimensions. This is often not true of other CNN acceleration designs that target specific convolutional layer parameters. Due to its accuracy rate and latency reduction, this work identifies BISC-MVM as the state-of-the-art for multipliers used in the convolutional layer. A 4-bit multiplier using BISC-MVM FSM SNG requires 288 gates, which is a 26 % increase over the conventional SNG cost.

3.4 Conclusion

This chapter covers related works that focus on reducing SC latency by targeting one of the non-idealities of SC arithmetic. In many of the presented designs, an accuracy increase is traded against an area increase. This trade-off happens in many of the designs. The only designs that do not have such a trade-off between accuracy and area, are the deterministic based designs, such as clock division, however, they suffer from

exponential growth in terms of stream length after each computation. Truncation of the results is possible to shorten the length, however, deterministic truncation methods result in low accuracy. BISC-MVM is a design that utilises early termination of the operation to reduce the processing time, by exploiting the redundancy of stochastic unipolar numbers. This reduction is quite significant and made it possible to increase the SC performance. While all of the mentioned designs improved SC multiplication in terms of latency, these had a quite significant hardware cost or other problems such as exponential growth or inaccurate truncation such as in the deterministic approaches. The next chapter proposes a new SC multiplier design called StoHej, which utilises early termination and complementary events.

Chapter 4

StoHej - Proposed Multiplier

This chapter describes the proposed multiplier, StoHej. Firstly, the chapter presents an analysis of a conventional SC multiplier and the BISC-MVM multiplier. The result from the analysis is that SC multiplication can be terminated early due to the redundancy of $0s$ in the unipolar encoding. Early termination is achieved by using an uncorrelated stream and a correlated stream. The correlated stream packs all the $1s$ at the beginning. If the correlated stream does not contain any $1s$ then the operation is terminated. Exploiting the redundancy in unipolar encoding was the main idea behind BISC-MVM, which was first proposed by Sim and Lee [40]. Secondly, the chapter provides a new multiplier design called StoHej, which extends BISC-MVM with complementary events, which enable reduces computation cycle number reductions for input operands that are larger or equal to 0.5. Thirdly, we describe how to use StoHej in a CNN context and how the acceleration configuration affects CNN performance and hardware efficiency. Finally, we discuss special optimisations techniques such as scaling values, multiple Search Areas (SA) for values smaller than 0.5, and multiple reductions, which cascades the setup cycles. StoHej can make use of these techniques to improve its performance in some specific cases.

4.1 Theory Behind the Proposed Multiplier

Consider an SC unipolar multiplier operating on 4-bit inputs as displayed in Figure 4.1. The blue coloured $1s$ in the figure indicate 1s occurring in both bitstreams at the same position, while red coloured $1s$ represent 1s occurring in only one bitstream. N_0 is the number of $0s$ in a stream and N_1 stands for the number of $1s$ in a stream. The circuit includes two SNG units, an AND-gate, and a counter. Each SNG process the 4-bit input operands A and B, and for every clock cycle, each SNG randomly generates a number and compares it to each respective input operand. If the input value is greater than the randomly generated number, then the SNG outputs a 1, else it outputs a 0. The entire generation of the stochastic bitstream takes 16 cycles since 16 bits are needed in the stream to represent all possible values in a 4-bit binary value. The AND-gate processes the two generated bitstreams and outputs the probability PQ, which is equal to $PA \cdot PB$. Finally, the counter adds every incoming $1s$ and completes the conversion from the stochastic bitstream to a binary number. The entire multiplication takes 16 clock cycles, and therefore, the latency for such a design is $O(N)$, where N is the length of the stream. To get a better grasp on the conventional SC multiplication, we demonstrate the computation with an example where A is $\frac{6}{16}$ and B is $\frac{10}{16}$. During the computation, the two SNGs generate a stream with 1s and 0s corresponding to the respective input values. When no random fluctuations occur, the A stream has six $1s$ and ten $0s$, and the B stream has ten $1s$ and six $0s$. However, depending on the bit order, the multiplication result can be between 0 and 6. To get 0, all $1s$ for A can be at the beginning, and B can have all $1s$ at the end.

This bit order results in no overlapping 1s, and therefore, the product is 0. However, to get 6, all the 1s in the A stream must overlap with all 1s in the B stream. From these two examples, we can deduce that the product value spans between 0 and the minimum of A and B and as such, the operation does not need to run for more than six clock cycles. However, both streams contain random bits and therefore ending the operation after six clock cycles may result in severe accuracy loss.

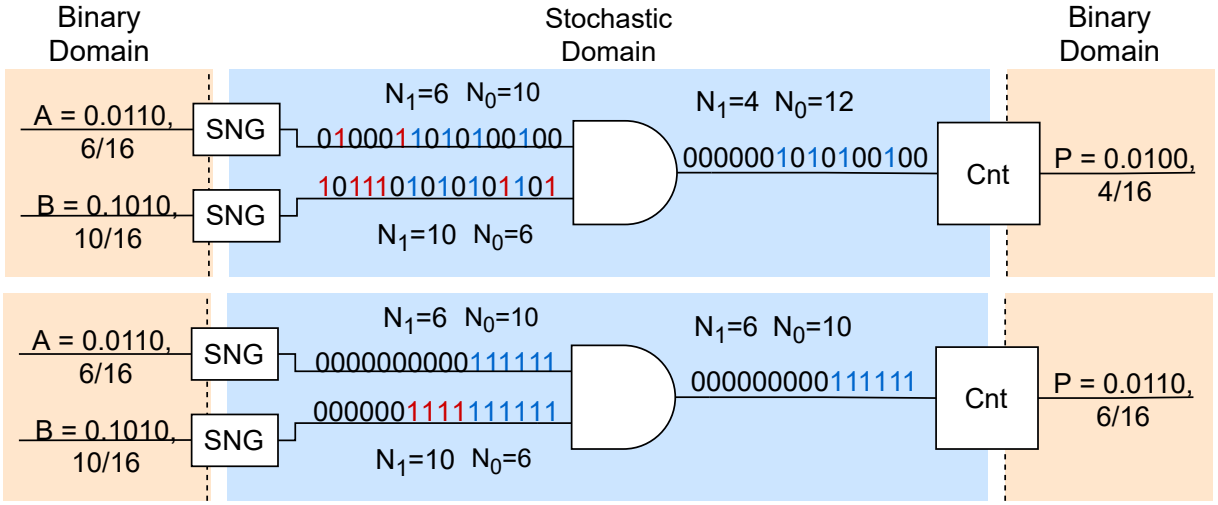


Figure 4.1: *Upper diagram illustrates a conventional SC multiplier, Lower diagram shows the effects of correlation.*

Now, rearrange both streams so that all 1s are at the beginning of the stream, followed by the remaining 0s. The final result of both streams is the minimum of both streams since in order to produce a 1 bit both input bits must be 1. If a 0 appears in either stream, the multiplier terminates the operation without affecting the result. Therefore, it is possible to use the input operands to determine how many iterations are required. However, this stream configuration results in a minimum function instead of a multiplication function, which is due to the correlation between the two streams.

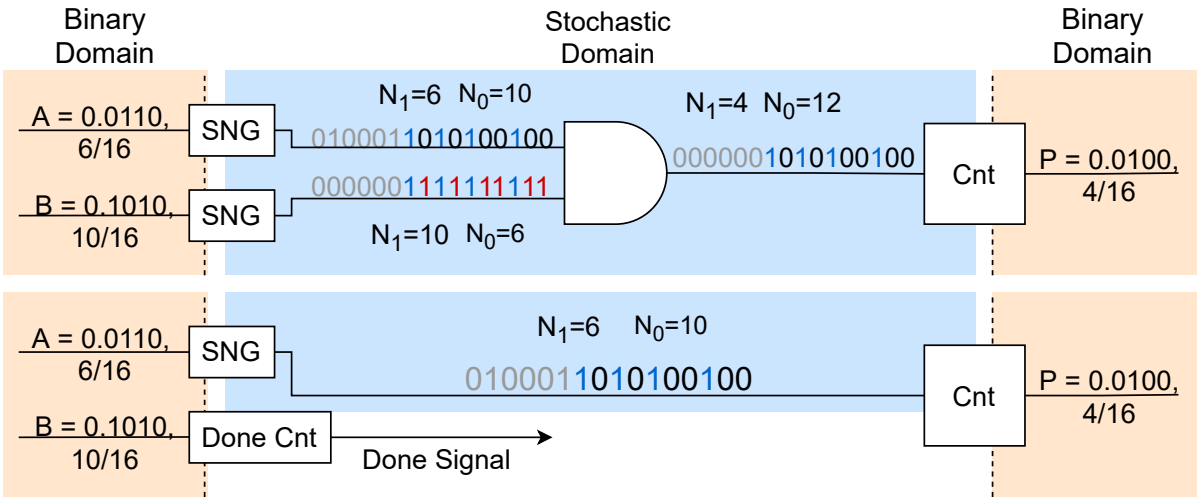


Figure 4.2: *The upper diagram shows the effects of combining streams. The lower diagram is BISC-MVM.*

To obtain multiplication while making use of a reliable termination mechanism, we can feed A into a regular SNG and B on a custom SNG that places all the 1s at the start of its output stream, as depicted in Figure 4.2. The design illustrated in the figure is called Binary Interfaced Stochastic Computing Matrix

Vector Multiplication (BISC-MVM) was first proposed by Sim and Lee in 2017 [40]. From the figure, we can conclude that the operation only needs 10 clock cycles instead of 16, because at this point the B stream has no more 1s. Continuing the operation after this point would not change the final value, as to report a 1 for output both input streams have to be 1. The multiplication circuit is simplified further by removing the AND-gate and connecting A directly to the stochastic-to-binary converter. The custom SNG is not present and a down-counter initialised with B is utilised instead. The average number of iterations in this design is less than $O(N)$. However, when B is close to 1, the latency is almost the same as the one of a conventional SC multiplier.

4.2 StoHej

This section describes this thesis proposal which is StoHej, an SC multiplier that uses complementary events to find the least number of iterations required to compute the product, as depicted in Figure 4.3.

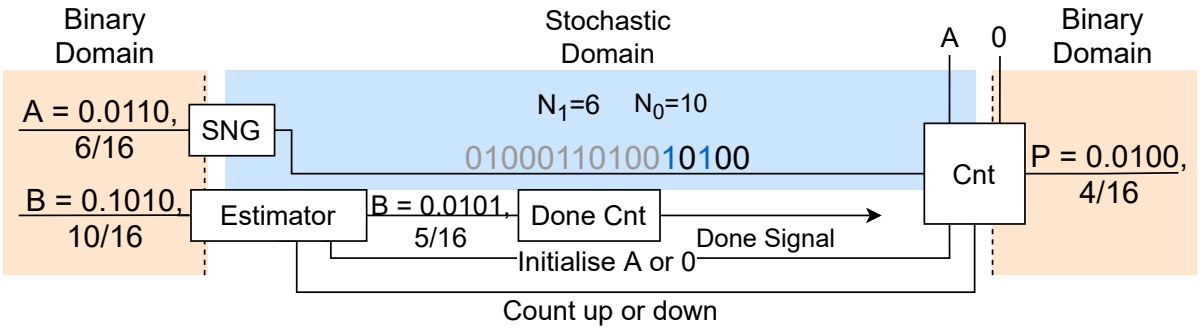


Figure 4.3: *Proposed multiplier StoHej.*

As discussed in the previous section, we can improve SC multiplication by rearranging one of the bitstreams and terminating the multiplication when that stream has no 1s left. This mechanism provides significant speedup when B is close to 0. However, when B is close to 1, the latency is similar to conventional SC. We can further improve upon the design in Figure 4.2 by observing that the product is between 0 and the minimum among A and B. Here we assume that the absolute value of A and B is between 0 and 1, which means that the computation does not need to operate more than the minimum of A and B cycles. This axiom limits the range where the product can be found from [0, 1] to [0, minimum(A, B)]. An SC multiplier starts from 0 and obtains the result after N cycles, where N is the bitstream length. BISC-MVM uses operand B to decide when to terminate the operation, which only provides a cycle reduction if B is not too close to 1. Given that $0 \leq P \leq A$ we can observe that the SC multiplier can reach P by starting from 0 and counting upwards or starting from A and counting downwards. Depending on the distance between P to A and P to 0, it might be more advantageous to start from A instead of 0, which is Stohej's main point. However, we can not use the difference between A and the product to decide what point to start counting from because we only know A and B and not the product.

The second point we can observe is that B can be utilised to decide if it is more advantageous to start from A or from 0. As B decides the number of cycles a multiplication takes, we should try to minimise B as much as possible, without reducing the accuracy of the computation. We can achieve this goal by observing that it takes B cycles to reach from 0 to the product and it takes the complement of B cycles to reach the product from A. To select the smallest B, we can check if the Most Significant Bit (MSB) is 1. If MSB is 1, then the complement of B will be smaller than B, as the MSB signifies that B is larger or equal to 0.5. If MSB is 0, then computing B is more advantageous as the complement of B will be larger than B. These two statements can be confirmed by observing that the complement of an event $1 - p$ and

if p is larger than 0.5, then $1 - p$ will be smaller than p. However, if p is smaller than 0.5, then $1 - p$ will be larger than p.

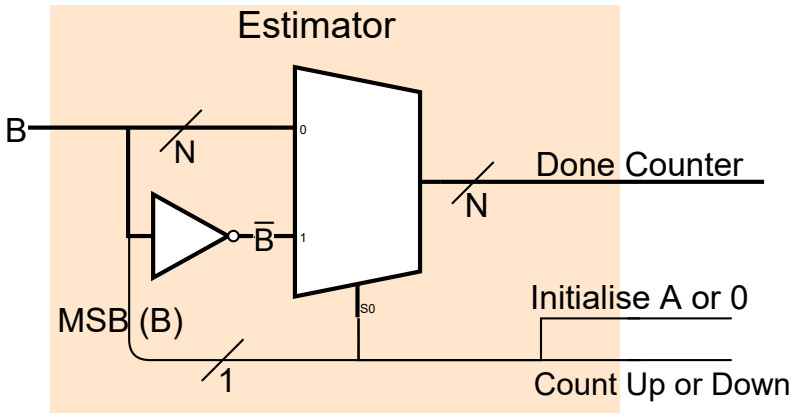


Figure 4.4: *The estimator circuit for StoHej.*

When computing with the complement, the multiplier needs to do two things: the first is to initialise the counter with A and the second is to count down for every incoming 1s. If these two steps are not performed, then the calculation results in a highly inaccurate product. To further explain, a stochastic bitstream is a Bernoulli process, which is a sequence of binary variables. For example, a sequence of coin tosses can be treated as a Bernoulli process, where the two outcomes are heads or tails. A stochastic bitstream can also be considered a sequence of coin tosses, but the outcomes are increment value or not (in a unipolar encoding). However, if the multiplier computes with the complement, then the event is also redefined from incrementing to decrementing. A hardware implementation of this mechanism is depicted in Figure 4.4 and here one can observe that the circuit only need to check the Most Significant Bit (MSB) of B. If MSB is 1, then the value is at least 0.5, and the inverted B is smaller than B. The MSB of B also selects if the stochastic-to-binary converter increments or decrements for every 1 in the stream and if the converter is initialised with 0 or A.

While a conventional SNG works for this method, it is also possible to use a deterministic SNG. The solution uses only one stream and therefore we do not need to consider the interactions between the two streams. In this case, only the distribution of 1s and 0s matters and not the order. The proposed multiplier uses the same deterministic SNG as BISC-MVM, which is more accurate and requires less hardware when compared to an LFSR-based SNG. BISC-MVM uses an alternative SNG design that is optimized for a stream configuration. Instead of a random SNG, a Finite State Machine (FSM) is used to select different bits from the binary SNG input. The pattern selection is based on the different weights of the fractional binary number. Weight 0.5 is selected half the time, weight 0.25 is selected a quarter of the time, and so on. The number of states in the FSM is equal to the bitstream length. Another advantage of this SNG is that it can be shared for multiple A inputs, which is beneficial for a Multiply Accumulate (MAC) design.

4.3 StoHej in CNNs

This section describes how the proposed multiplier can be used to accelerate the convolution calculation and possible parallel configurations to increase hardware efficiency.

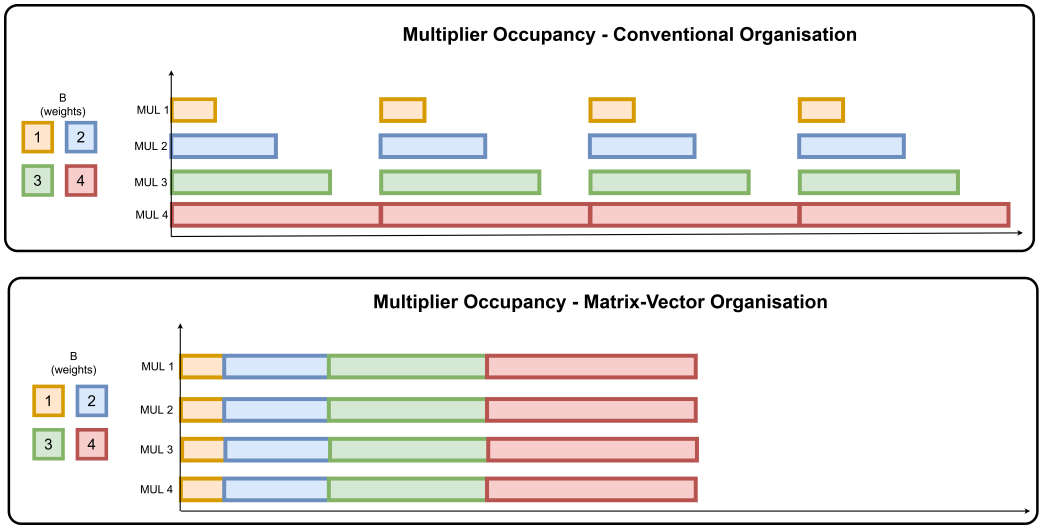


Figure 4.5: *The upper timeline is for conventional kernel acceleration and the bottom timeline for matrix vector multiplication configuration.*

The conventional way to accelerate the convolutional layer of a CNN is to accelerate the kernel computation as shown in Figure 4.5. The kernel engine in the figure has 4 multipliers and computes a $(2, 2)$ convolution operation in parallel, which has 4 A inputs for the feature map and 4 B inputs for the weights. The adder tree is not depicted in the figure, as the focus is the utilisation of each multiplier. For more information about the convolution operation, see Section 2.3. The weights in the kernel are $(1, 2, 3, 4)$ and the last weight in the set takes the most computation cycles since the weights determine the latency. When the computation starts, $1, 2, 3$ will finish before weight 4. The multipliers for $1, 2, 3$ have to wait until 4 is finished because the products connect to an adder tree, and starting the calculation of the next pixel results in an incorrect answer. While the last multiplier computes, the other multipliers are not used and thus results in lower utilisation of the multipliers. In a CNN application, the utilisation of the multipliers is vital, as it improves throughput and reduces hardware usage. One way to solve the scheduling issue is to broadcast the same weight to all of the multipliers because then all the multipliers end their computation at the same time. This configuration is called matrix-vector multiplication and is depicted in Figure 4.6. For each iteration, a new section of the feature map is loaded and multiplied by a single B or weight value. The products are then accumulated in the output feature map. The number of accumulations is equal to the number of weights. This design speeds up along the dimensions of the output feature maps rather than speeding up the kernel computation. This configuration is convenient for smaller output feature maps, but for larger maps, an impractical number of multipliers and accumulators are required. Therefore, only one tile of the output map is computed instead, which is more practical for larger maps. Here, we have three tile parameters, namely TM, TR, and TC. TR and TC only need to be incremented with A inputs, as this increases the height and width of the computation tile. However, TM increases with B inputs which require more SNG units and down-counters. For more information about the matrix-vector multiplication configuration, see Section 2.3.

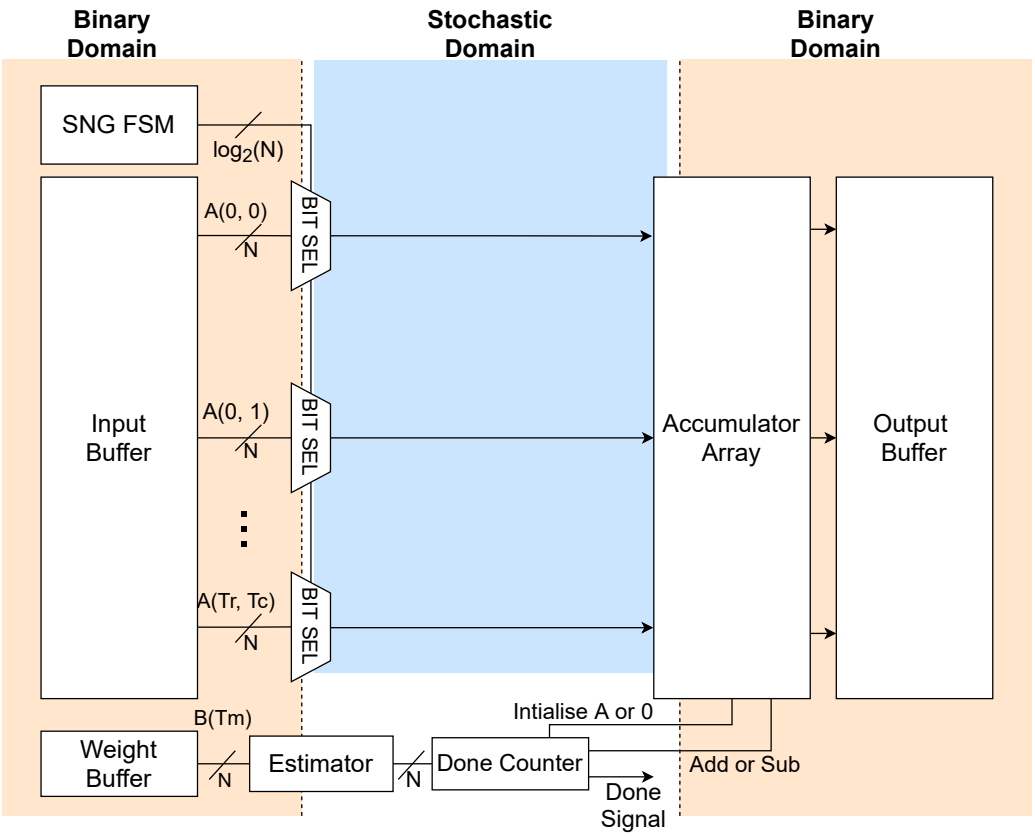


Figure 4.6: *Micro-architecture of a MAC unit using StoHej multipliers.*

4.4 Additional Techniques

This section describes additional techniques that can be applied to Stohej to further optimise for certain CNN applications. All of the techniques mentioned in this section are contributions of the thesis and were developed on their own.

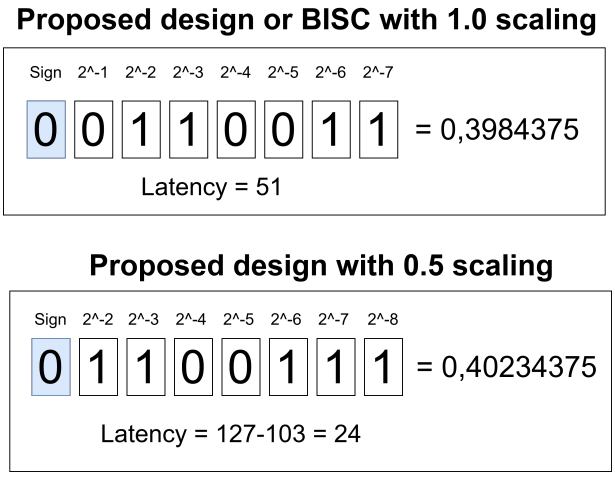


Figure 4.7: *Effects of scaling on representation and latency.*

Some special techniques can be applied to StoHej, which can increase the computation latency reduction in specific cases. These techniques include scaling the B-input (weight value), multiple search ranges, and multiple reductions. The argument for scaling is that StoHej's latency reduction only activates when

the B values are above 0.5. However, in a trained CNN, only a few weight values are above 0.5, therefore there is no significant benefit of using StoHej in this situation [40]. Nevertheless, we can trigger the latency reduction more often by scaling the weight values, so that a majority of them are above 0.5. This is not a conventional scaling where the value is multiplied with a scaling value, the weights of the bits are shifted instead. We assume that the CNN is trained on a platform with higher precision, which is then converted to a smaller precision. In the conversion process, many weights will lose some information. However, if a majority of the weights are smaller than 0.5, then scaling for those value results in higher resolution. However, weights over 0.5 will saturate at 0.5 as MSB is 0.25 instead of 0.5.

Figure 4.7 also depicts the example of scaling weight value $\frac{51}{128}$ by two, which becomes $\frac{103}{256}$. If we do not use any scaling, the number of computation cycles for both StoHej and BISC-MVM is the same. However, scaling, in this case, results in a significant difference in computation cycles. BISC-MVM computes the product in 103 cycles, while StoHej computes the product in 24 cycles, due to the complement mechanism described earlier in Section 4.2. The implementation of scaling hardware depends on the scale factor. If the factor is a power of two, then simple rewiring is sufficient, however, scaling with non-powers of twos requires more hardware and therefore are not commonly implemented.

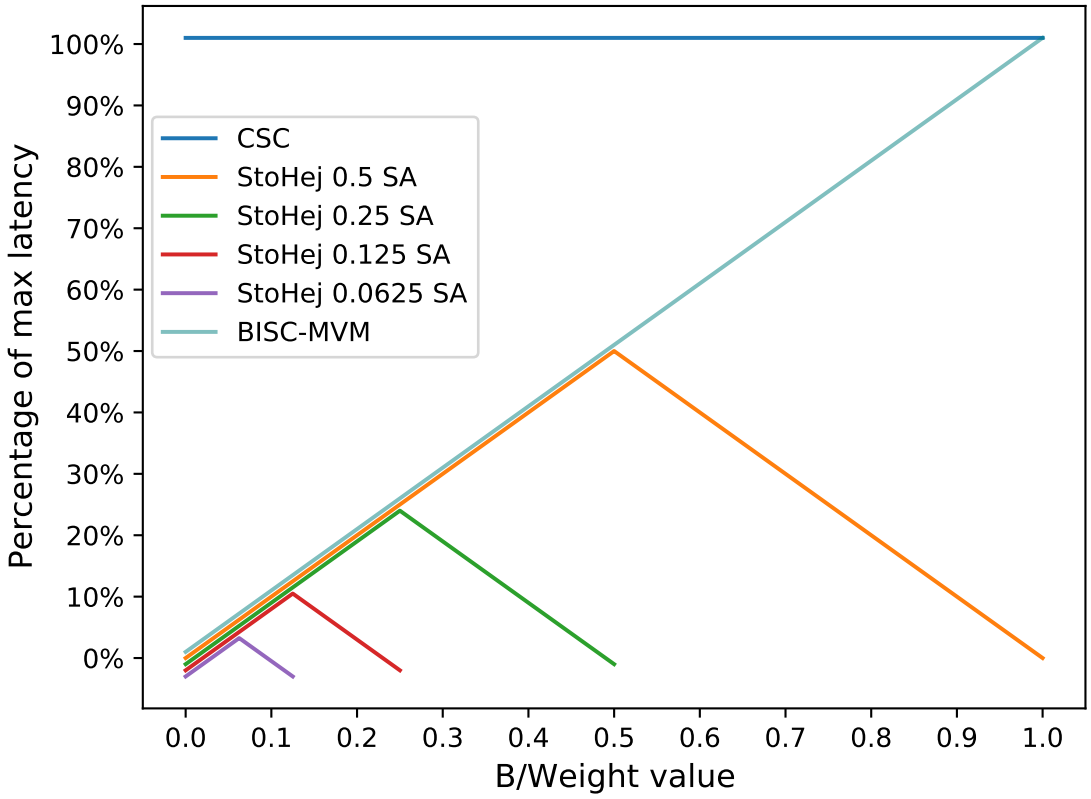


Figure 4.8: *Latency against different weight values.*

However, scaling might not be practical for all applications, which leads to the second technique of multiple search areas. A Search Area (SA) refers to the check that is performed first in StoHej. For example, the checking of 0.5 is called the 0.5 search area or SA. It is also possible to check any powers of two, such as $0.5, 0.25, 0.125$, and so forth. The purpose of using multiple SA is that a significant portion of the weight values are below 0.5 and do not gain any benefit from the complement mechanism. By

using a smaller SA, the multiplier can reduce latency for these weights, which further reduce the average latency of the CNN. Figure 4.8 depicts B value vs percentage of max latency. Conventional Stochastic Computing (CSC)'s latency is independent of the B value, which results in a flat line. BISC-MVM's latency is proportional to the B value. However, StoHej's different SAs results in max points, where latency starts decreasing again. Where the max point appears depends on the SA. A smaller SA means that the complement mechanism can be triggered for smaller values and reduce latency. This technique is quite useful for applications that have a majority of weight values below 0.5. The computation of smaller SAs is a bit different as the circuit check if values are in a certain range, which means the multiplier need to check at least two bits, while the 0.5 search area only needs to check MSB. The initialisation of the stochastic-to-binary counter is also changed when computing with smaller SAs, by shifting A to the appropriate range. For instance, 0.25 SA needs to shift the A input by 2, the 0.125 SA needs to shift by 4 and so on.

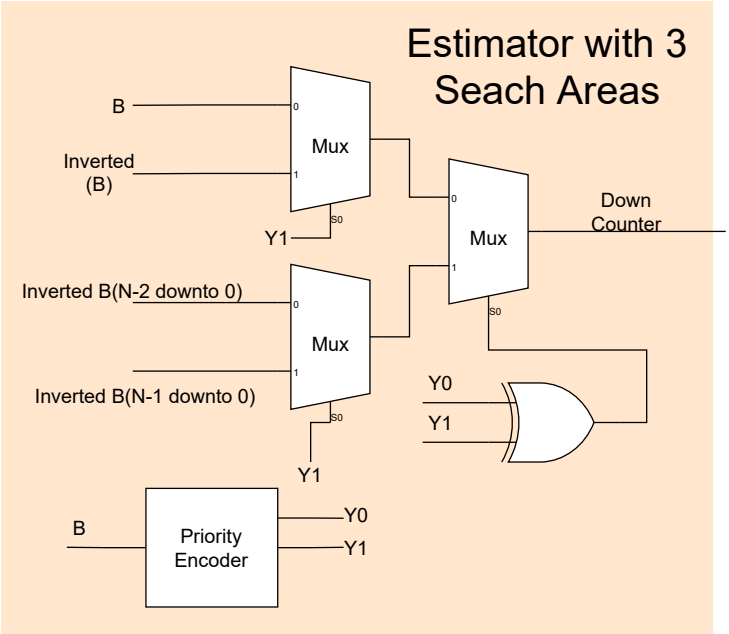


Figure 4.9: *StoHej estimator with 3 Search Areas (SAs).*

The hardware also needs to be changed to use smaller SAs. Figure 4.9 shows StoHej estimator with 3 SAs, which check 0.5, 0.25, and 0.125. The priority encoder checks the MSB position and selects the correct B. To compute using 0.25 SA, the hardware checks whether B is greater than or equal to 0.25 and less than 0.5. If so, the algorithm inverts all bits except the MSB of the magnitude part. If the MSB is also inverted, then the value is at least 0.5, resulting in an inaccurate calculation. The product counter must also be initialised with $\frac{A}{2}$, which is realised by simple rewiring. It is possible to check several SAs at the same time since the weight value only falls into one SA. This can be used as an alternative solution to scaling, for applications that have a majority of values below 0.5.

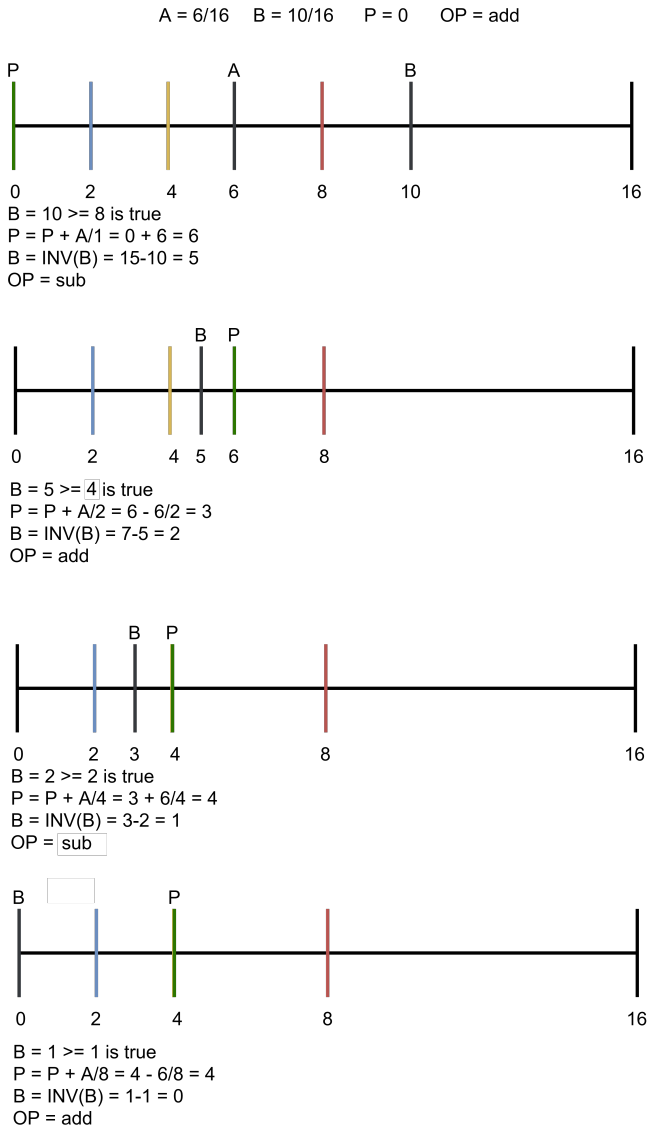


Figure 4.10: *A fully unrolled reduction.*

The third and final method is called multiple reductions. The main idea is to cascade the complement mechanism for at least two iterations, which means that the result from the previous iteration is fed into the next one. Iteration in this context refer to utilising the complement mechanism, so two iterations, means that the multiplier has utilised the complement mechanism twice. For each iteration, the multiplier checks if B is larger than 0.5. If it is larger, B is inverted and A gets added to the stochastic-to-binary converter, which is also called product counter. The next iteration checks a smaller SA, which in this case is 0.25, if true, then the circuit inverts all bits of B except for the previously checked. The stochastic-to-binary converter shifts A by 2 to the right and subtracts from the accumulated value. For each successful check, the stochastic-to-binary changes the operation from addition to subtraction, due to the estimation overshooting or undershooting the result. To further clarify, consider the multiplication: $A = \frac{6}{16} \cdot B = \frac{10}{16}$, as shown in Figure 4.10. The multiplier first checks if B is larger or equal to 0.5, which is true. Thus, B is set to 5, as the bit-inversion of 10 is 5. The product counter sets its value to A and sets that the next operation should be subtraction. The current estimated product is 6, which is currently overshooting the final product and therefore, we need to subtract for the next successful check. Now, the multiplier checks if B is larger or equal to 0.25, which is also true. B has inverted again and now is 2, but now excluding MSB, as inverting MSB again results in a larger number. The product counter uses this equation to

compute the current estimation: $Product\ Counter = Product\ Counter - \left\lfloor \frac{A}{2} \right\rfloor = 6 - \left\lfloor \frac{6}{2} \right\rfloor = 3$. Now the multiplier is underestimating the product, thus the next operation for the product counter should be addition. For the third iteration, the multiplier checks if B is larger or equal to $\frac{2}{16}$ or 0.125, which is also true. B has inverted again and results in 1. The product counter updates its estimation which is now equal to $Product\ Counter = Product\ Counter + \left\lfloor \frac{A}{4} \right\rfloor = 3 + \left\lfloor \frac{6}{4} \right\rfloor = 4$. As B is not zero, the operation continues and is now checked against 1, which is true. B gets inverted again and becomes 0. The product counter's value is now: $Product\ Counter = Product\ Counter + \left\lfloor \frac{A}{8} \right\rfloor = 4 + \lfloor \frac{6}{8} \rfloor$. The operation has ended and the final product is $\frac{4}{16}$. Using multiple reductions, require more hardware and requires additional estimation cycles. However, it can be a useful technique for applications that need a higher precision with a smaller number of computation cycles. The number of estimation iterations is flexible, but should be chosen with care, as it affects hardware usage and can diminish efficiency.

4.5 Conclusion

This chapter covers the proposed multiplier StoHej and the potential benefit of its design in terms of reducing the number of computation cycles. It also covers the reasoning behind StoHej by comparing it with a conventional SC and BISC-MVM, which is the state-of-the-art for SC multipliers. While the theory behind the multiplier is promising, it also vital to test the multiplier design empirically. By testing the multiplier, we can better understand the benefits and potential drawbacks and identify the situations when StoHej is preferable to BISC-MVM. The next chapter documents a set of experiments that compare the performance of StoHej, BISC-MVM, and other multipliers.

Chapter 5

Experiments

This chapter documents the results of experiments performed with StoHej, Conventional SC (CSC), and BISC-MVM. The purpose of the experiments is to determine the performance of StoHej in a general and in a CNN context. Firstly, we tested the multipliers in a general context, by using a programme written in C. All multipliers in the programme were tested with every possible input combination from 3-bit to 8-bit precision. The software simulation measures two metrics, latency, and accuracy. Secondly, we performed an analysis of how the B/weight value affects latency and how it affects the choice between Stohej and BISC-MVM. Thirdly, we evaluated metrics such as hardware usage and energy consumption in an FPGA experiment in a CNN inference context.

5.1 Stand-alone Multiplier Performance

This experiment obtained data related to accuracy and latency, which helps to make further analysis of the multipliers in a CNN application. StoHej, CSC, and BISC-MVM were implemented in a C programme, where each multiplier is a function. The function has two arguments which are the input operands and returns the product. The simulation allowed for variable bit width, which was used to observe how latency and accuracy change over bitstream size. Each simulated multiplier was tested with a uniform dataset. A uniform data set refers to a dataset where every input combination is used. Accuracy was measured in Mean Absolute Error (MAE), which is computed by the absolute difference between the correct product and the generated product from the multiplier. This was done for every input combination for all multipliers. CSC requires random SNGs and thus more samples are required per input combination, as the product might vary, but StoHej and BISC-MVM use a deterministic SNG, thus only one sample is needed for every input combination. Latency is computed by the number of iterations required for each input combination. CSC latency is independent of input combinations and is equal to the bitstream length. BISC-MVM latency is equal to the B input value, which means that for every bitstream size an average is taken for all input B values. StoHej's latency is computed by using the initialising cycle, which means that if B is larger or equal to 0.5 then the latency is equal to $bitstreamsize - 1 - B$. For values below 0.5, the latency is identical to BISC-MVM.

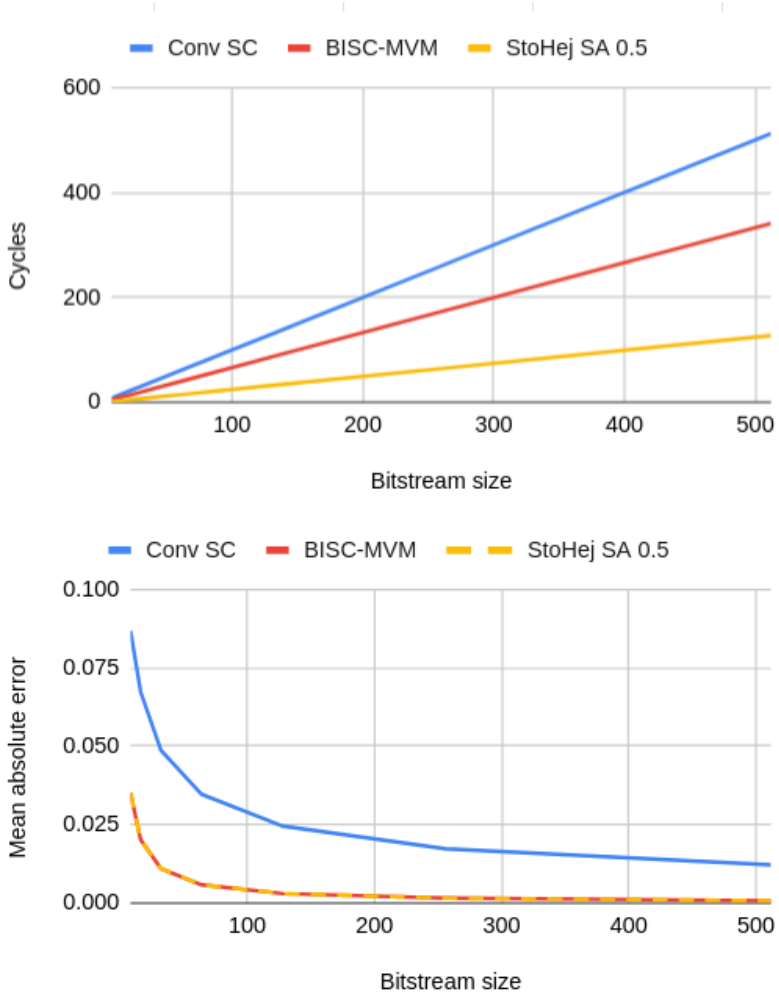
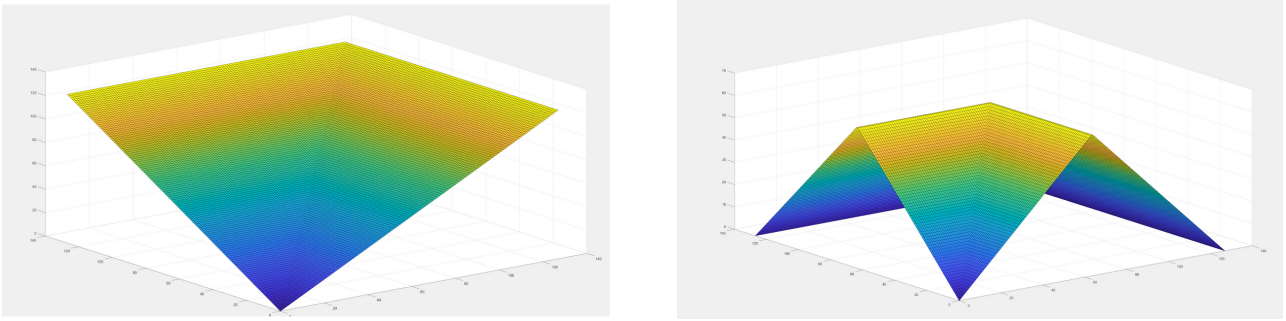


Figure 5.1: *The upper graph shows latency and lower graph shows accuracy.*

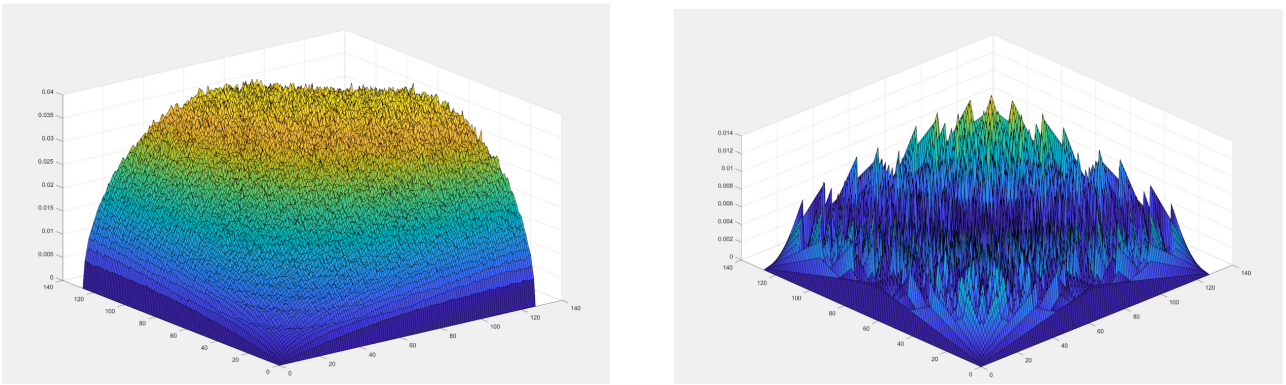
The upper section of Figure 5.1 shows the average latency for all input combinations in cycles for StoHej, BISC-MVM and CSC for bit sizes of 8 to 512. CSC requires the largest number of cycles when compared to StoHej and BISC-MVM. CSC does not use any special case for early termination and always processes the entire stream. BISC-MVM and StoHej's latency is dependent on the B operand value, while CSC's latency is fixed for all input operands. The latency of BISC-MVM is smaller than that of CSC, mainly due to the use of early termination based on the B input. The number of iterations used for the computation is equal to B, exploiting the fact that the product will be less than or equal to B. For any bitstream size, BISC-MVM's latency is half of CSC's latency. The latency of StoHej half and quarter of BISC-MVM and CSC latency, respectively. This is due to the use of complementary events, which reduces the latency for B values above 0.5. This changes the worst-case latency from $O(N)$ to $O(\frac{N}{2})$. The BISC-MVM latency for values above 0.5 increases, but not for StoHej due to the complementary events utilisation. It can also be observed in the upper section of Figure 5.1 that the latency of all multiplier techniques grows linearly with the bitstream size, but that BISC-MVM and StoHej grow at a shallower angle. Nevertheless, StoHej has the smallest latency compared to the other multiplier designs. Figure 5.2a and 5.2b show two 3D graphs where X and Y are input operands the Z is the latency for BISC-MVM and StoHej, respectively. CSC has not been plotted as the latency is fixed for all input combinations and hence will be drawn as a flat plane. BISC-MVM latency is drawn as a slope, with latency increasing linearly with input values. However, StoHej has a similar shape until halfway through the graph, where it is convolved in half. This also confirms that the worst case for StoHej is $O(\frac{N}{2})$ since the largest latency

is in the middle. For BISC-MVM, the worst-case latency is at the end of the graph, which is equal to $O(N)$.



(a) *Latency for BISC-MVM.* (b) *Latency for StoHej.*

Figure 5.2: *3D graphs of BISC-MVM and StoHej in terms of latency.*



(a) *Mean Absolute Error for CSC.* (b) *Mean Absolute Error for BISC-MVM and StoHej.*

Figure 5.3: *3D graphs of MAE.*

The second metric used in the multiplier evaluation is accuracy. Accuracy, as mentioned earlier, is defined as MAE and is computed for all input combinations. The lower section of Figure 5.1 shows a graph, where the X-axis is the bitstream size from 8 to 512 bits and the Y-axis is the average MAE for all input combinations. In the graph, we can observe that CSC (Written as Conv SC in the graph) has a higher MAE for all bitstream sizes, which is due to several sources of errors, such as random fluctuations, correlation, which are described in more detail in Section 2.1.1. BISC-MVM and StoHej have the same accuracy for the same bitstream size, which is due to the fact that both designs make use of the same deterministic SNG design. This result also indicates that the complement mechanism in StoHej does not degrade accuracy. Figure 5.3 illustrates two 3D graphs where X and Y are input operands and Z is MAE. For CSC it forms a rough sphere, where MAE is the largest, in the middle of the two axes. This can be explained by the number of possible arrangements the bits can have when there is an equal number of $1s$ and $0s$. The number of combinations decreases when it is closer to 0 or 1. Figure 5.3b presents MAE for BISC-MVM and StoHej, as they have identical accuracy. The graph has many spikes, which is due to the deterministic way streams are generated and the fact that certain computations have an exact representation. For example, $\frac{4}{8}$ multiplied by $\frac{4}{8}$ will have no errors, since it is computed to $\frac{2}{8}$. But certain results cannot be represented, for example, $\frac{4}{8}$ multiplied by $\frac{5}{8}$ cannot be represented accurately, so an approximation is performed, which in this case results in $\frac{2}{8}$. The rounding errors can be solved by using a bitstream size equal to the product of the two denominators. However, this leads to excessively long streams in most cases and may not be needed in CNN applications.

Software simulation results have shown that StoHej has a smaller average latency when compared

to BISC while providing the same accuracy. However, there is a potential hardware usage and energy consumption overhead that needs to be studied carefully. BISC-MVM is a simpler design but has a longer latency than StoHej for values above 0.5. StoHej requires slightly more hardware, but requires a smaller number of cycles for values above 0.5. This means that the performance gain of StoHej over BISC-MVM depends on the amount of above 0.5 numbers. If there are no values above 0.5, then the latency of StoHej is equal to that of BISC-MVM but with more area usage, which means a suboptimal design. However, if a significant amount of the numbers are above 0.5, then StoHej has significantly better performance. The next set of experiments will attempt to estimate the area overhead of StoHej and determine how many weight values larger than 0.5 are needed in order to be cost-effective when compared to BISC-MVM.

5.2 Weight Analysis

To obtain data regarding hardware usage, an FPGA implementation was created for BISC-MVM and StoHej. Xilinx Zynq 7000 SoC was used which contains an FPGA and an ARM processor. Both multipliers were implemented in Vivado 2018.3 using VHDL as the hardware description language and then implemented as an IP block with an AXI bus interface. The ARM processor was used to send and fetch data from the multipliers. The synthesis and implementation of the multipliers in Vivado show the number of Look-Up Tables (LUTs) and the number of slice registers. This is used to determine the hardware usage of each multiplier design. The next step is to determine what ratio of values above 0.5 is required for StoHej to perform better in terms of area-delay product. The evaluation dataset was created by randomly generating numbers below 0.5 and then numbers above 0.5. Then the ratio between numbers below 0.5 and above 0.5 was adjusted from 0% to 100% with 5% increments.

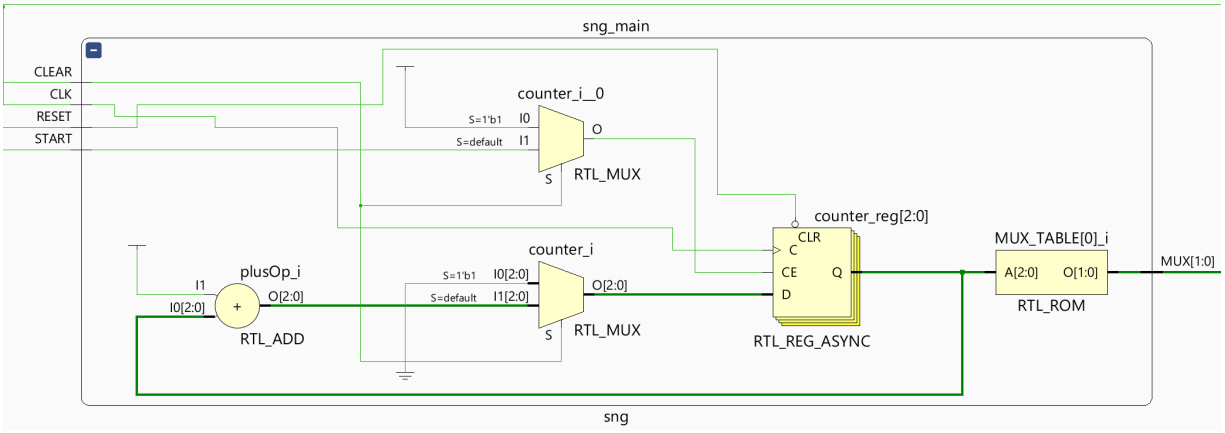


Figure 5.4: *RTL diagram of the Bit shuffling SNG.*

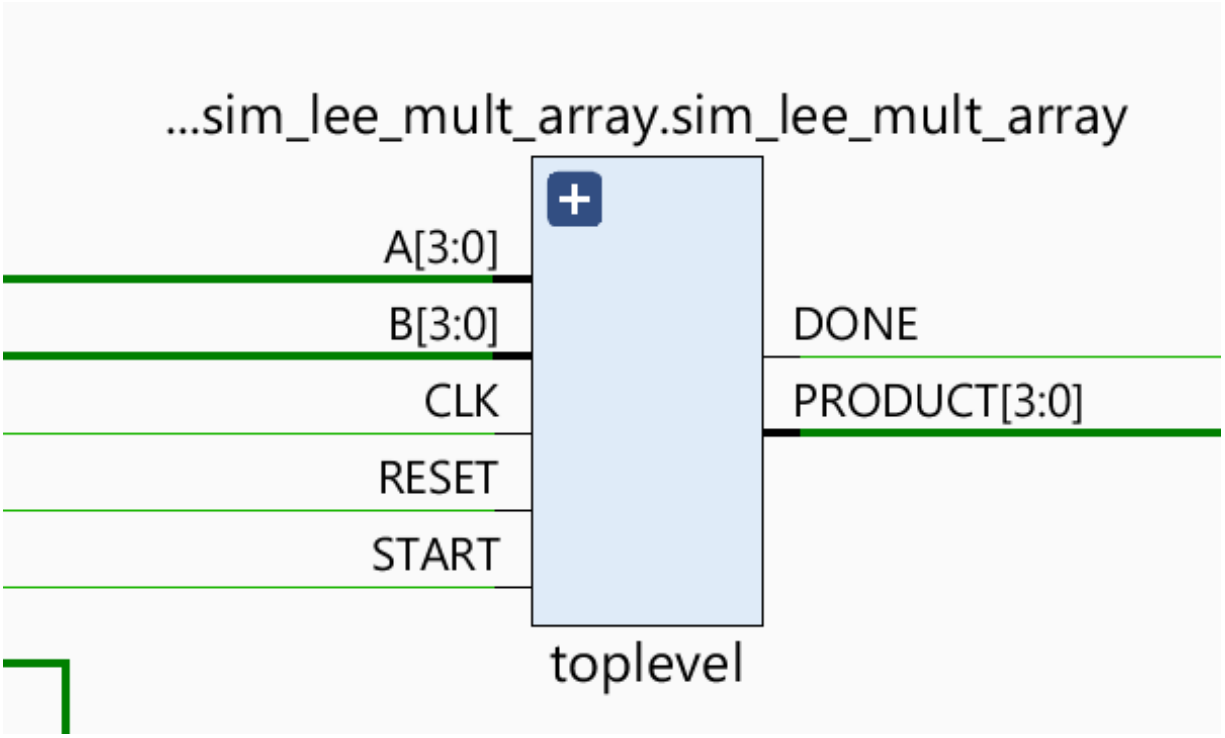


Figure 5.5: *RTL diagram of StoHej multiplier.*

The VHDL implementation of BISC-MVM and StoHej has been divided into three components. The first component is the SNG, which has three inputs, namely CLK, RESET, and START as depicted in Figure 5.4. The SNG output is the mux selector which controls the bit selection of A. The mux values are stored in a LUT and for each clock cycle, the counter is incremented to point to a different address of the LUT. When START is set, the SNG starts counting, when START is set again, the whole process starts over. The second component is the multiplier, which takes CLK, RESET, START, A, B, and outputs PRODUCT and DONE, see Figure 5.5. The multiplier makes use of an SNG to generate the a bitstream. The multiplier has an FSM with three states. The first state is an init state where the product counter and iteration counter are initialised. When the start is high, the iteration counter is set to B or inverse to B depending on whether the MSB of B is enabled. The product counter is set to A if MSB of B is enabled, otherwise, it is set to 0. BISC-MVM skips these steps. Then the actual calculation begins. At each clock cycle, the product counter is incremented or decremented depending on whether MSB of B is enabled and whether the bitstream has one. Also, the iteration counter is decremented until it reaches 0. When it reaches 0, the operation is complete. The third component is the AXI interface and the registers used to control the multiplier. Two registers are mapped to the input of the multiplier and one to the output of the multiplier. An additional register is used as a control register. The control register is used to start the operation and to check if the operation is complete. An onboard cycle counter was used to count the number of cycles for each operation. This was used to remove the overhead of communication during the measurement process.

BISC-MVM and StoHej with 25% of B values over 0.5								
Bits	Average Cycles		Energy (mJ)		Area (FPGA LUT)		Speedup	ACE PG
3	2.50	2.49	0.019	0.019	12	31	1.00	0.389
4	5.47	4.47	0.042	0.034	17	46	1.23	0.560
5	11.61	8.47	0.089	0.065	23	48	1.37	0.899
6	23.95	16.86	0.185	0.13	28	54	1.42	1.048
7	47.92	33.23	0.371	0.257	39	62	1.44	1.310
8	95.70	64.81	0.742	0.502	45	83	1.48	1.183
9	193.47	129.87	1.502	1.008	46	97	1.49	1.053

Table 5.1: The results from the weight analysis where 25% of the B values are larger than 0.5. The first column of Average Cycles, Energy and Area is BISC-MVM and the second column is StoHej.

The reported area figures refer only to the multiplier and the SNG itself, since there are many ways to implement an interface to the multiplier, and the focus of the thesis is on multiplier designs and their impact on arithmetic efficiency in CNNs. Next, to determine what ratio of values is required for StoHej to perform better than BISC-MVM, the onboard ARM processor executes a test program that generates a random set of data the adjustable ratio between values below and above 0.5. The same seed was used for both datasets to make the comparison as equal as possible. The ratio starts at 0% above 0.5 and increases by 5% at each step. Bit widths from 3 to 8 were tested in this experiment.

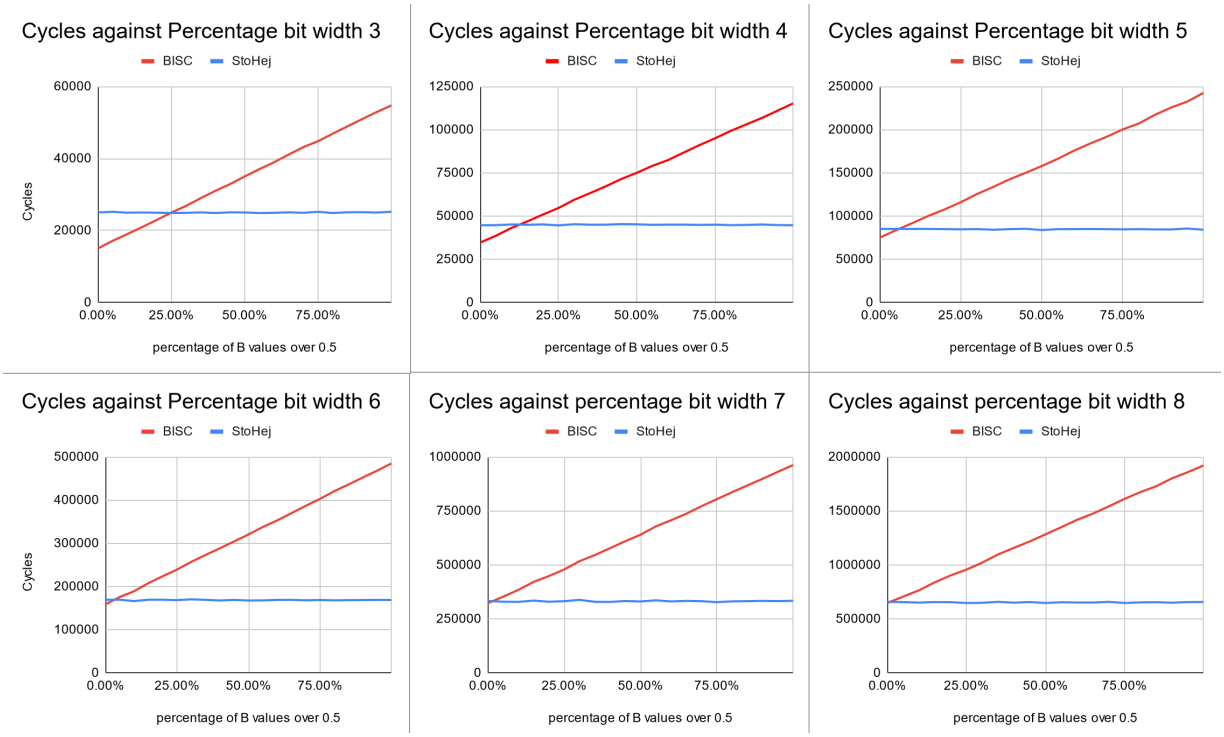


Figure 5.6: *Latency vs. percentage values over 0.5.*

The results of the experiment can be found in Figure 5.6. Here it can be seen that BISC-MVM has better latency when the amount of larger than 0.5 B values is close to 0%. This is due to the overhead cycle in StoHej to determine if computing with complementary events is beneficial. If the overhead cycle is excluded, then the latencies are identical when 0% of the values are above 0.5. As the ratio increases, StoHej stays on the same line while the BISC-MVM latency increases. At a certain point, BISC-MVM and StoHej latency becomes equal. As can be seen, this intersection point is at 25% when the bit width is 3 and shifts further to the left as the bit width increases. This is mainly due to the influence of the

overhead cycle. An overhead cycle has a greater impact on latency for smaller bit widths than for larger ones. For a bit width of 8, the intersection occurs quite early, around less than 1%. Tables 5.1, 5.3, and 5.2 shows the result from the weight experiment. In Table 5.1 only 25% of the B values are larger than 0.5, which has a clear impact on Speedup and Area-Cycles-Energy Product Gain (ACE PG). For the smaller bits, StoHej has fewer cycles, however, when considering the cost of Area, StoHej is not cost-effective for 3 to 5 bits. For the larger bit sizes, StoHej becomes cost-effective and outperforms BISC-MVM.

BISC-MVM and StoHej with 75% of B values over 0.5								
Bits	Average Cycles		Energy (mJ)		Area (FPGA LUT)		Speedup	ACE PG
3	3.50	2.50	0.035	0.019	12	31	1.78	1.268
4	7.51	4.53	0.073	0.035	17	46	2.11	1.625
5	15.80	8.39	0.154	0.065	23	48	2.36	2.684
6	32.07	16.78	0.312	0.131	28	54	2.39	2.946
7	64.01	33.10	0.624	0.254	39	62	2.45	3.792
8	128.33	64.84	1.250	0.503	45	83	2.49	3.349
9	256.14	129.23	2.488	1.000	46	97	2.49	2.934

Table 5.2: The results from the weight analysis where 75% of the B values are larger than 0.5. The first column of Average Cycles, Energy and Area is BISC-MVM and the second column is StoHej.

BISC-MVM and StoHej with 50 % of B values over 0.5								
Bits	Average Cycles		Energy (mJ)		Area (FPGA LUT)		Speedup	ACE PG
3	3.50	2.50	0.027	0.019	12	31	1.40	0.770
4	7.51	4.53	0.058	0.035	17	46	1.66	1.015
5	15.80	8.39	0.122	0.065	23	48	1.88	1.692
6	32.07	16.78	0.248	0.13	28	54	1.91	1.891
7	64.01	33.10	0.496	0.256	39	62	1.93	2.357
8	128.33	64.84	0.995	0.503	45	83	1.98	2.123
9	256.14	129.23	1.989	1.003	46	97	1.98	1.864

Table 5.3: The results from the weight analysis where 50% of the B values are larger than 0.5. The first column of Average Cycles, Energy and Area is BISC-MVM and the second column is StoHej.

Table 5.3 and 5.2 indicates that StoHej becomes more cost-effective as more of the B/weight values are over 0.5, which demonstrates the importance of analysing the trained weights. Here we can see that the speedup is around 2.3 when the 75 % values are above 0.5. The speedup and ACE PG increases when the number of bits increases. The area for StoHej is larger than BISC-MVM, but the iteration and latency reduction outweighs the increased area. These results demonstrate the importance of analysing the trained weight values before deploying StoHej as the weight values directly impact latency, and thus, decides if StoHej outperforms BISC-MVM. We note that if not enough weight values are over 0.5, other techniques can be utilised, such as multiple Search Areas (SAs) and scaling the weight values, which are described in Section 4.4. We observe a similar trend for energy consumption in Table 5.1 where both 3-bit StoHej and 3-bit BISC-MVM consumes the same amount of energy. When the number of bits increases, the difference in energy consumption also starts to increase for StoHej and BISC-MVM. This relationship is also true, when increasing the number of B-weight values over 0.5. In Table 5.3 and 5.2 3-bit BISC-MVM increases its energy consumption, while 3-bit StoHej keeps the energy consumption at the same level. Here we can observe that the number of bits and the number of B values over 0.5 impacts the potential performance gain StoHej can obtain when compared to BISC-MVM.

5.3 CNN Context

This section describes a series of experiments conducted with different multiplier designs in a CNN context. Instead of evaluating a generic input set, the multipliers are used in a trained CNN. The first section discusses an experiment conducted with different multiplier designs by means of a GPU-based simulation. The CNN application used in this experiment is LeNet-5 with the MNIST dataset [28]. LeNet-5 consists of two convolutional layers, two pooling layers, and three fully connected layers. The CNN was first trained using Python and Keras and achieved an accuracy rate of 98.92% for 10,000 MNIST test images. Only the convolutional layers are accelerated on the GPU, while the rest are computed on the CPU. The multiplier designs are applied only to the convolutional layer, as this is the main focus of this thesis. Five different multipliers have been implemented on the GPU: standard floating-point, fixed-point, CSC, BISC-MVM, and StoHej. The floating-point implementation relies on a 32-bit format and is used as comparison basis for CNN's accuracy rate. The bit width of the other designs is variable from 2 to 9 bits. Each multiplier was tested with 10,000 images from the MNIST dataset. The accuracy rate is calculated by dividing the number of correct predictions by the number of predictions.

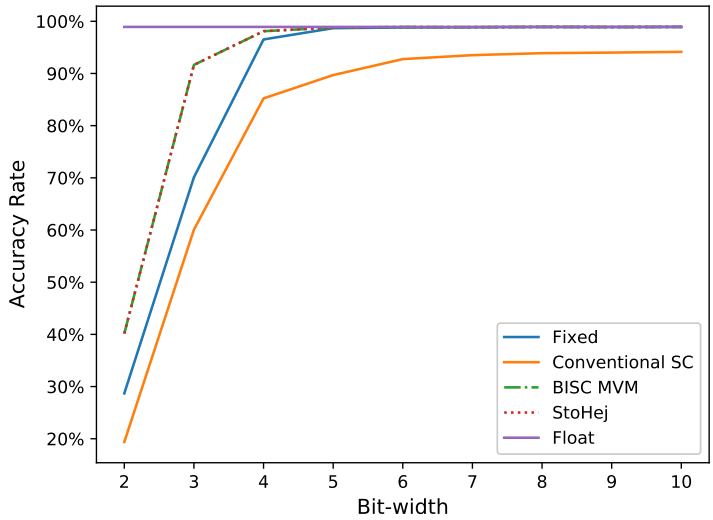
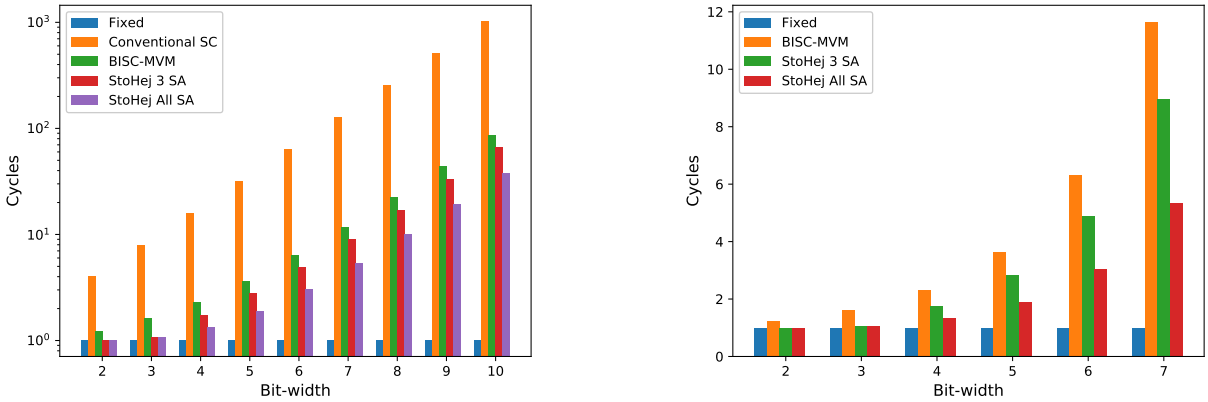


Figure 5.7: *Accuracy-rate for LeNet-5 and MNIST.*

Figure 5.7 shows the accuracy rate obtained when the five different multipliers, are utilised in the LeNet-5 implementation. Floating-point is drawn as a straight line and represents the baseline accuracy rate. Most designs will either be at the same accuracy rate or below. The fixed point starts at 29% and saturates at 98.92% when the bit size is 5. CSC starts at 19% and saturates at 92% when the bit size is 6. BISC-MVM starts at 40% and saturates at 98.92% when the bit size is 4. StoHej has an identical accuracy rate, which is due to the fact that it uses the same SNG design. We also need to consider the latency for each design. The latency for fixed-point, StoHej, BISC-MVM, and CSC was calculated. The fixed point has fixed latency if we assume that the multiplier is an array multiplier. CSC latency depends on the bitstream size. StoHej and BISC-MVM need to know the weights of the layer and CNN's third layer was used to determine the latency for both designs. Figure 5.8a depicts average cycles for Fixed, CSC, BISC-MVM, StoHej with 3 SAs, and StoHej with all SAs enabled. All SAs enabled means that the multiplier check all possible bits for possible complement. CSC have the most cycles, which is mainly due to not utilising any early termination. BISC-MVM is significant faster than CSC, which is due the early termination technique. Fixed point has the fewest cycles, which is expected. StoHej with

3 SAs significantly outperforms both BISC-MVM, however the fastest design out of the SC multipliers is StoHej with all SA. Figure 5.8b depicts the same graph with CSC excluded. In this graph we can see that StoHej All SA and 3SA provide 50% and 33% delay reduction over BISC-MVM, which demonstrates the importance of SAs when it comes to latency reduction.

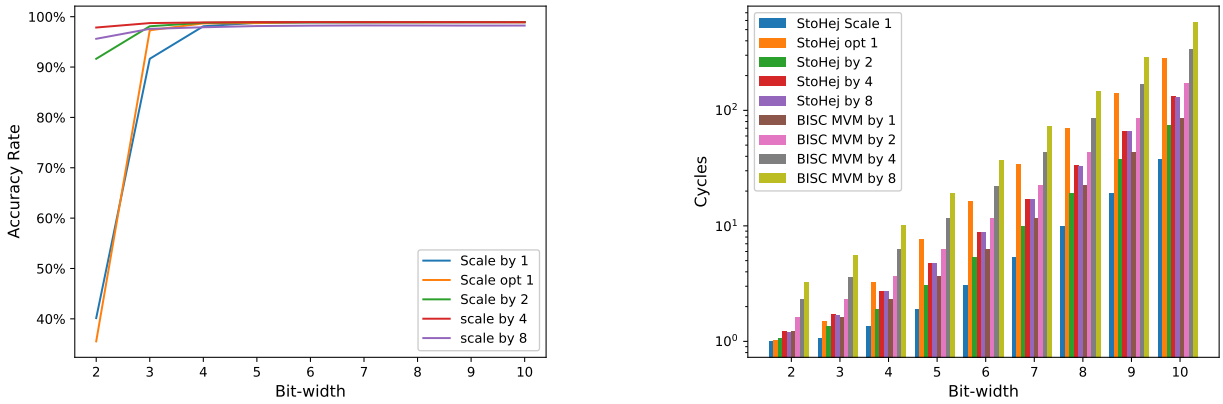


(a) *Latency for MNIST.*

(b) *Excluding CSC.*

Figure 5.8: *Average multiplier latency for MNIST layer 3.*

The next step of testing is to observe how scaling affects the accuracy rate and latency for BISC-MVM and StoHej. There is an opportunity to use scaling to achieve a better representation of numbers and latency with StoHej. Three different weight scales were used, ie., 2, 4, 8, and also a special strategy to optimise the scaling weights as close as possible to 1 with powers of two without exceeding 1. For more information about scaling, see Section 4.4. The idea of this strategy is to allow a more accurate representation of numbers without hurting the upper range, which happens when scaling overall values. Figure 5.9a shows the accuracy rate for different scales. Here it can be observed that scaling helps at the earlier bit sizes, but goes into saturation at lower accuracy than scaling with 1. Scaling with optimal value to 1 had a beneficial effect at the beginning, but then the effects are similar to scaling with 1. Scaling with 2 was the most beneficial. Figure 5.9b shows the latency for BISC-MVM and StoHej with different scaling. StoHej with scaling 1 has the best performance, overall designs, but other designs had a positive impact in some cases. Scaling optimal 1 had lower latency at the beginning compared to another scaling scheme, but this fell apart at large bit sizes. This is mainly because certain weights cannot come close to 1 when scaling with powers of 2. This means that the latency for these values increased instead of reaching the original value. The latency of BISC-MVM is quite high, which is mostly due to the fact that it does not make use of complementary events. This means that BISC-MVM is not suitable for designs that use scaling when compared to StoHej, but still has better performance than CSC.



(a) *Accuracy rates.* (b) *Latency for different weight scale.*

Figure 5.9: *MNIST and LeNet-5 with different scales.*

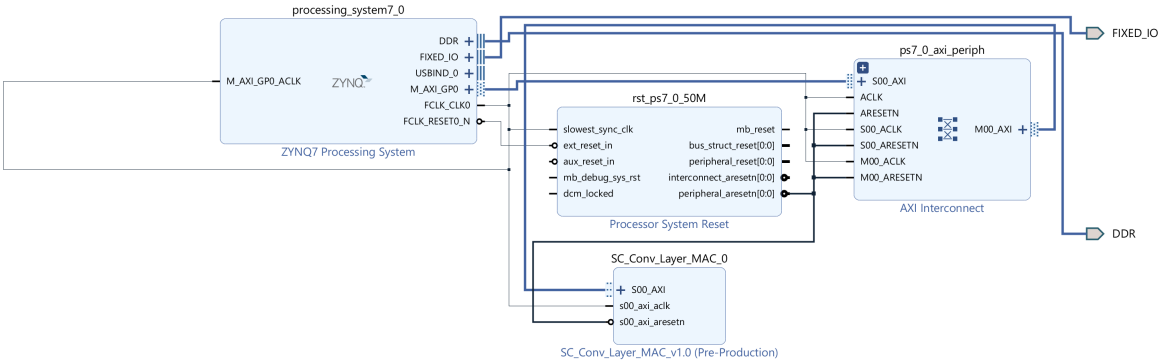


Figure 5.10: *Block diagram of the FPGA design.*

The final step in the CNN experiments is to evaluate BISC-MVM and StoHej on an FPGA implementation. This implementation is different from the earlier VHDL implementation of the multipliers. As in this case, the convolutional layer accelerator requires an array of multipliers and an array of accumulators. The configuration of the accelerator is such that it speedup the computation along the tile of the output map. The component hierarchy consists of a multiplier component, a saturated accumulator component, and an AXI bus component. The multiplier component takes multiple input feature map values and a single weight value. The output of the multiplier is then fed into the saturated accumulator. The final output values are stored in the AXI bus layer, which is read by the ARM processor. In the ARM processor, all layers are implemented in software, while the convolutional layers are computed on the hardware module. A block diagram of the design can be seen in Figure 5.10.

As confirmed earlier, the accuracy rate was the same for both multipliers. StoHej with three search areas has 1.7x speedup when compared to one of BISC-MVM, see Table 5.4. Table 5.5 shows the average area overhead, cycle reduction, and energy reduction of StoHej when compared to BISC-MVM. The area for StoHej is slightly larger, mainly because of the estimation part and the fact that each adder in the multiplier has to subtract and add. However, the Area-Delay Product (ADP) is 30% smaller when compared to BISC-MVM's ADP. Thus, the latency reduction outweighs the area increase. In terms of energy consumption, StoHej provides a reduction of about 40.0%, which is mainly due to latency reduction as the power consumption remains about the same. The Area-Delay-Energy Product (ADEP) for StoHej

StoHej						
Data bitwidth	LUT Area	Cycles per Tile	Energy (mJ)	Speedup	Area-Cycles-Gain	Area-Cycles-Energy-Gain
9	1991	2738	3,358	1,71	1,30	2,23
8	1663	1411	1,738	1,69	1,24	2,10
7	1620	784	0,963	1,58	1,21	1,92
6	1459	117	0,756	1,62	1,22	1,36
5	1218	89	0,722	1,57	1,36	1,38
BISC-MVM						
9	1510	4686	5,766	1	1	1
8	1217	2386	2,943	1	1	1
7	1238	1237	1,532	1	1	1
6	1100	189	0,846	1	1	1
5	1054	140	0,733	1	1	1

Table 5.4: *FPGA results comparing StoHej and BISC-MVM*

StoHej's Area overhead	StoHej's Cycles Reduction	StoHejs's Energy Reduction
29.5%	38.7%	40.0%

Table 5.5: Aggregates of StoHej and BISC-MVM from the FPGA experiment.

is about 2.3x smaller than BISC-MVM. Thus, it can be concluded that StoHej performs better than BISC-MVM in this case. However, it is important to analyse the weights in a trained CNN. If the wrong search ranges are chosen, then no significant latency improvement can be obtained.

5.4 Conclusion

In conclusion, Chapter 5 documents three major experiments. The purpose of the first experiment is to evaluate StoHej as a stand-alone multiplier. StoHej was compared to BISC-MVM and CSC. In terms of latency, StoHej was 3.2 times faster than BISC-MVM and 5.5 times faster than CSC. The error of BISC-MVM and StoHej was significantly lower than that of CSC. The experiment also proved empirically that computing with complementary events has no negative impact on accuracy. In the second experiment, hardware usage and energy consumption data were obtained to determine the overhead. The experiment was also used to determine when StoHej outperforms BISC-MVM. The iteration reduction of StoHej and BISC-MVM is determined by the weight value. For 3-bit precision, there must be at least 0.5 values above 25%, for StoHej's performance to equal that of BISC-MVM. The intersection points shift back as the bit-size increases. The third and final experiment tests StoHej in a CNN context. The CNN context was the LeNet-5 network with the MNIST dataset. The baseline accuracy for the network was 98.92% with floating-point multiplication. Fixed point obtained similar accuracy when the bit width was 5, while StoHej and BISC-MVM obtained it at 4-bit width. CSC saturated at 92 %. In the next series of experiments, StoHej and BISC-MVM were tested with different scaling weights. The results showed that BISC-MVM was not optimal for scaling because complementary events were not used. The optimal scaling to 1 proved to be suboptimal. The final part of the experiment tested CNN in an FPGA. This was done to test the concept in hardware and collect energy and hardware data. The results of the experiment showed that StoHej had a speedup of 1.7x when compared to BISC-MVM. The Area-Delay Product for StoHej is 30% smaller than BISC-MVM. When energy consumption is included, the Area-Delay-Energy Product (ADEP) of StoHej is $2.3x$ smaller than BISC-MVM. StoHej's area overhead when compared to BISC-MVM is 29.5%, however StoHej also had a latency reduction of 38.7% and an energy reduction of

40.0%.

Chapter 6

Conclusions and Future Work

In conclusion, this thesis addressed the problem of providing low-cost and low energy consumption solutions for ANN/CNN inference on Internet of Things (IoT) devices by means of Stochastic Computing (SC). To this end we investigated and proposed ways to improve SC multiplier performance while not diminishing the computation accuracy and at minimal chip real estate expenditures. These efforts resulted in the proposed SC multiplier StoHej, which uses complement events to further reduce cycle count while not affecting accuracy. We also created variations of the multiplier algorithm, which can be used to further optimise in specific cases. These techniques are scaling the weight value, multiple Search Areas (SAs), and multiple reductions. These techniques significantly reduces latency, without diminishing accuracy, while keeping the hardware cost at minimal.

Chapter 2 provides the necessary background information about Stochastic Computing, Artificial Neural Network, and Convolutional Neural Network. It also describes the original reasoning behind Stochastic Computing.

Chapter 3 describes various methods for reducing latency in SC multipliers. The methods ranged from improving SNGs to parallel processing of stochastic numbers to early termination.

Chapter 4 describes the proposal for a new SC multiplier. The multiplier uses BISC-MVM as its framework and improves it by utilising complementary events to reduce the number of iterations. The new SC multiplier uses the following observations: (i) The product of a multiplication is between 0 and the minimum among A and B. Here we assume that A and B are between 0 and 1. A and B . (ii) The computation does need to operate more than the minimum of A and B cycles. (ii) Given that $0 \leq Product \leq A$ the multiplier can reach P by starting from 0 and counting upwards to P or starting from A and counting downwards to P. (iv) To start from A, the multiplier need to use the complement of B to achieve the correct result. (v) As B decides the number of cycles, the multiplier should always take the smallest version of B. When B is larger than 0.5, then the complement of B is smaller than B, as the complement of B is $1 - B$. This chapter also described how the multiplier could be used in a CNN context and what type of kernel organisation was best to increase multiplier utilisation.

Chapter 5 focuses on evaluating multipliers in two different contexts. The first is a general context where the multiplier is evaluated in terms of accuracy, latency, energy consumption, and hardware usage. The second context is in a CNN application. In the general context, the proposed multiplier was compared with the state-of-the-art and the conventional SC. The next step of the evaluation was data acquisition in CNN context, which was first performed on GPU simulation. The proposed solution was compared with the state-of-the-art and the conventional SC. The evaluation results showed that the proposed design

had the same accuracy but the proposed solution had lower latency. The disadvantage of the proposed solution was the increase in overhead area and overhead cycle. This was not considered in the software simulation. The next context took a pre-trained LeNet5 and replaced the multiplication algorithm with different variants. Float, fixed point, conventional SC, BISC and the proposed design were tested.

The CNN application was also tested in an FPGA. The multipliers were packed into a vector multiplier that takes multiple image inputs with a single weight. The organisation was also different from the conventional structure of the convolution operations. The micro-architecture of the MAC was built as a Matrix-Vector multiplier. The design also had adjustable tiles that allowed fine-tuning of area usage and speedup.

The results show that the proposed multiplier can be used as a viable alternative to the prior art for low-cost CNN acceleration. However, it is important to ensure that the application weights are above certain values or multiple searches ranges other than 0.5 are used. If this is not the case, the proposed design will suffer due to the overhead. However, in most cases, it will be possible to adjust the weights for a trained neural network by scaling. Experiments also show that reducing latency is important for energy consumption, which is sometimes overlooked. This is sometimes overlooked in SC designs where the latency is too high to produce a low energy consumption application.

Although the proposed improvements are significant, this technique is currently limited to multiplication and has not yet been tested in an SC circuit that has an arithmetic depth greater than one. For example, it would not be possible to combine this multiplier and follow it up with an SC adder. This may change if a more comprehensive theory is developed, and it would greatly improve latency for more complex SC functions. The BISC multiplier also has this problem. The next question that can be discussed is whether binary search pre-estimation can be used for other arithmetic functions. SC multiplication has well-defined bounds that do not increase after each multiplication. But for addition and division, it is possible to produce a number greater than 1.

6.1 Contributions

In this section, we briefly describe the contributions of this thesis. We divide them into SC multiplier design, stand-alone multiplier performance, and CNN performance. This thesis has the following contributions:

- A new SC multiplier design called StoHej that utilises complementary events to achieve the smallest number of cycles to reach the product. StoHej builds upon the following observations: (i) The multiplication of operands operands residing between [0, 1] results in a product between 0 and the minimum among them. (ii) One can compute the product by either starting from 0 and counting upwards to the product or starting from the minimum among the input operands and counting downwards. (iii) By using the complement of one operand, the multiplier can count down from the other input operand to the product. (iv) It is more advantageous to use the complement of the input operand when the operand is larger than 0.5 because the complement is $1-p$, and a smaller number means fewer cycles. These features make it possible for StoHej to reduce to worst-case latency from $O(N)$ to $O(\frac{N}{2})$. The detailed design is presented in Section 4.2.
- Three additional StoHej applicable techniques to further optimise its performance in terms of cycle count when utilised in certain applications. The three additional technique are the following: (i) Scaling the input operand, (ii) Utilising multiple Search Areas (SAs), and (iii) Multiple reduction cycles. (ii) Make use of the same complement mechanism for a smaller range, which means instead of using the complement for numbers between $[0.5, 1]$, the multiplier instead checks if the operand is

between $[0.25, 0.5]$, which changes the complement equation from $1 - p$ to $0.5 - p$. This report uses the term Search Area (SA) to refer to this range, which means that multiple SAs checks for multiple ranges. Multiple SAs are used when scaling is not practical for an inference CNN application and a majority of input operands are not larger than 0.5. (iii) Utilisation of multiple reductions, which uses the complement mechanism repeatedly. The number of repetitions is configurable and should be chosen carefully as additional repetitions require additional hardware. Note that the first and second techniques can be applied when a majority of the trained CNN weights are below 0.5. The scaling technique scales the values over 0:5 so that the complement mechanism is utilised more. The additional techniques are presented in Section 4.4.

- The third contribution is the evaluation of StoHej, BISC-MVM, and CSC multipliers in a standalone context. BISC-MVM stands for Binary Interfaced Stochastic Computing Matrix-Vector Multiplication and is the state-of-the-art for SC multipliers utilised in CNN inference applications. The three multiplier algorithms were implemented in C and all possible input combinations were tested for each bitstream size from 8 to 512 bits. In this experiment, StoHej proved to be 3.2x and 5.5x faster than BISC-MVM and CSC, respectively. In terms of accuracy, CSC has the lowest accuracy compared to StoHej and BISC-MVM, which provide the same accuracy, which indicates that StoHej's usage of complements did not cause any accuracy degradation. To evaluate area and energy consumption, we implemented StoHej and BISC-MVM on an FPGA. StoHej has a larger area than BISC-MVM, which was expected to the additional hardware required to compute with the complement. However, when considering the Area-Cycles-Energy Product (ACEP), StoHej had 2.9x smaller ACEP when compared to BISC-MVM when 75% of the weight values are over 0.5. We also performed an analysis to determine the number of weight values that need to be over 0.5 for the single SA StoHej to be cost-effective, which indicates that for smaller bit-sizes at least 25% needs to be over 0.5, however as the bit-sizes increases the percentage decreases. For 9 bits precision, less than 1% need to be over 0.5 to be cost-effective when compared to BISC-MVM.
- The fourth contribution is the evaluation of the proposed multiplier in a CNN context. StoHej and BISC-MVM were tested in a LeNet-5 network with the MNIST dataset. The different multiplier algorithm such as StoHej, BISC-MVM, CSC, Fixed-point, and Floating point was implemented on a GPU to obtain accuracy-rate for the different designs. In addition, StoHej with different SAs and scaling were also tested, to see their effect on accuracy rate and latency. To obtain energy and area data, StoHej and BISC-MVM were organised in a Multiply-Accumulate (MAC) array that was implemented on an FPGA. The experiment showed that StoHej with 3 SAs had a speedup of 1.7x and no loss in accuracy compared to BISC-MVM. StoHej has a larger area usage, however, when using the Area-Delay Product (ADP), StoHej had 30 % reduction when compared to BISC-MVM. The Area-Delay-Energy Product (ADEP) of StoHej is $2.3x$ smaller than BISC-MVM. StoHej's area overhead when compared to BISC-MVM is 29.5%, however StoHej also had a latency reduction of 38.7% and an energy reduction of 40.0%.

6.2 Future Work

There are multiple avenues to explore for future work. One avenue is to use a similar algorithm modified for other arithmetic operations such as division and trigonometric functions. Using this theory for other operations could create lower latency for low-cost circuits, such as Discrete Cosine Transform (DCT), that is used in JPEG compression. There are SC-based implementations for more complex operations. Still, these implementations suffer from the latency accuracy problem. This problem is more severe in SC circuits that have an arithmetic depth higher than 1. This means for example that two numbers

are multiplied and that product is then added together with a third number. The product of the first two numbers is correlated, which reduces the accuracy of the result. The circuit above would have an arithmetic depth of 2. The general solution is to increase the stream length, which increases the latency of the computation. It would be an avenue for further exploring, that would enable low-cost and low latency versions of more complex functions.

Another direction is the exploration of a floating denominator system. This means that the denominator of each number would be multiplied together, while the numerator would be using the proposed scheme. The multiplication of the denominator would be implemented as a shifter. This would increase the range of representation. For example, $\frac{1}{16}$ multiplied with $\frac{1}{16}$ would normally result in 0. However, with the new scheme, the denominator will be shifted and increase the range, which would result in an accurate product, which in this case is $\frac{1}{256}$. This would increase the accuracy for harder CNN problems, but without increasing the latency.

The third area to explore would be related to the utilization of alternative encoding. For example, if it is possible to use different fixed point representations. This could enable usage for applications that need a integer range larger than 1.

Bibliography

[1] Oludare Abiodun et al. "State-of-the-art in artificial neural network applications: A survey". In: *Heliyon* 4 (Nov. 2018), e00938. DOI: 10.1016/j.heliyon.2018.e00938.

[2] A. Alaghi and J. P. Hayes. "Fast and accurate computation using stochastic circuits". In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2014, pp. 1–4. DOI: 10.7873/DATE.2014.089.

[3] Armin Alaghi and John P. Hayes. "Survey of Stochastic Computing". In: *ACM Trans. Embed. Comput. Syst.* 12.2s (May 2013). ISSN: 1539-9087. DOI: 10.1145/2465787.2465794. URL: https://doi.org/10.1145/2465787.2465794.

[4] Armin Alaghi et al. "Accuracy and Correlation in Stochastic Computing". In: Feb. 2019, pp. 77–102. ISBN: 978-3-030-03729-1. DOI: 10.1007/978-3-030-03730-7_4.

[5] Hany Ammar and Zhouhui Miao. "Parallel Algorithms for the Training Process of a Neural Network-Based System". In: *International Journal of High Performance Computing Applications - IJHPCA* 14 (Mar. 2000). DOI: 10.1177/109434200001400101.

[6] Plamen Angelov et al. *Advances in Computational Intelligence Systems: Contributions Presented at the 16th UK Workshop on Computational Intelligence, September 7–9, 2016, Lancaster, UK*. Vol. 513. Jan. 2017. ISBN: 978-3-319-46561-6. DOI: 10.1007/978-3-319-46562-3.

[7] Safia Bibi, Muhammad Ullah, and Muhammad Shami. "Design and Analysis of Hybrid Tree Multipliers for Reduction of Partial Products". In: *Mehran University Research Journal of Engineering and Technology* 37 (July 2018), pp. 483–492. DOI: 10.22581/muet1982.1803.04.

[8] B. D. Brown and H. C. Card. "Stochastic neural computation. I. Computational elements". In: *IEEE Transactions on Computers* 50.9 (Sept. 2001), pp. 891–905. ISSN: 1557-9956. DOI: 10.1109/12.954505.

[9] Emine Cengil, Ahmet Çinar, and Zafer Güler. "A GPU-based convolutional neural network approach for image classification". In: *2017 International Artificial Intelligence and Data Processing Symposium (IDAP)*. 2017, pp. 1–6. DOI: 10.1109/IDAP.2017.8090194.

[10] Tianshi Chen et al. "DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning". In: vol. 49. Feb. 2014, pp. 269–284. DOI: 10.1145/2541940.2541967.

[11] Yung-Yao Chen et al. "Design and Implementation of Cloud Analytics-Assisted Smart Power Meters Considering Advanced Artificial Intelligence as Edge Analytics in Demand-Side Management for Smart Homes". In: *Sensors* 19 (May 2019), p. 2047. DOI: 10.3390/s19092047.

[12] Dan Ciresan et al. "Flexible, High Performance Convolutional Neural Networks for Image Classification." In: July 2011, pp. 1237–1242. DOI: 10.5591/978-1-57735-516-8/IJCAI11-210.

[13] Jason Cong and Bingjun Xiao. "Minimizing Computation in Convolutional Neural Networks". In: *Artificial Neural Networks and Machine Learning – ICANN 2014*. Ed. by Stefan Wermter et al. Cham: Springer International Publishing, 2014, pp. 281–290. ISBN: 978-3-319-11179-7.

[14] M. Daalen et al. "A Device for Generating Binary Sequences for Stochastic Computing". In: *Electronics Letters* 29 (Feb. 1993), pp. 80–. DOI: 10.1049/el:19930052.

[15] Gülin Dede and Murat Hüsnü Sazlı. "Speech recognition with artificial neural networks". In: *Digital Signal Processing* 20.3 (2010), pp. 763–768. ISSN: 1051-2004. DOI: https://doi.org/10.1016/j.dsp.2009.10.004. URL: https://www.sciencedirect.com/science/article/pii/S1051200409001821.

[16] R. Du, S. Magnusson, and C. Fischione. "The Internet of Things as a Deep Neural Network". In: *IEEE Communications Magazine* 58.9 (2020), pp. 20–25. DOI: 10.1109/MCOM.001.2000015.

[17] B. R. Gaines. "Stochastic Computing". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 149–156. ISBN: 9781450378956. DOI: 10.1145/1465482.1465505. URL: https://doi.org/10.1145/1465482.1465505.

[18] M. Ghiassi et al. "Automated text classification using a dynamic artificial neural network model". In: *Expert Systems with Applications* 39.12 (2012), pp. 10967–10976. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2012.03.027. URL: https://www.sciencedirect.com/science/article/pii/S0957417412004976.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[20] P. K. Gupta and R. Kumaresan. "Binary multiplication with PN sequences". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 36.4 (1988), pp. 603–606. DOI: 10.1109/29.1564.

[21] S. Han et al. "EIE: Efficient Inference Engine on Compressed Deep Neural Network". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. June 2016, pp. 243–254. DOI: 10.1109/ISCA.2016.30.

[22] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. arXiv: 1510.00149 [cs.CV].

[23] Hsuan Hsiao, Jason Anderson, and Yuko Hara-Azumi. "Generating Stochastic Bitstreams". In: Feb. 2019, pp. 137–152. ISBN: 978-3-030-03729-1. DOI: 10.1007/978-3-030-03730-7_7.

[24] D. Jenson and M. Riedel. "A deterministic approach to stochastic computation". In: *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2016, pp. 1–8. DOI: 10.1145/2966986.2966988.

[25] Norman Jouppi et al. "Motivation for and Evaluation of the First Tensor Processing Unit". In: *IEEE Micro* 38.3 (2018), pp. 10–19. DOI: 10.1109/MM.2018.032271057.

[26] Kyounghoon Kim et al. "Dynamic Energy-Accuracy Trade-off Using Stochastic Computing in Deep Neural Networks". In: *Proceedings of the 53rd Annual Design Automation Conference*. DAC '16. Austin, Texas: Association for Computing Machinery, 2016. ISBN: 9781450342360. DOI: 10.1145/2897937.2898011. URL: https://doi.org/10.1145/2897937.2898011.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Neural Information Processing Systems* 25 (Jan. 2012). DOI: 10.1145/3065386.

[28] E. Kussul and T. Baidyk. "Improved method of handwritten digit recognition tested on MNIST database". In: *Image Vis. Comput.* 22 (2004), pp. 971–981.

[29] Endre László, P. Szolgay, and Zoltán Nagy. "Analysis of a GPU based CNN implementation". In: Aug. 2012, pp. 1–5. ISBN: 978-1-4673-0287-6. DOI: 10.1109/CNNA.2012.6331451.

[30] Yann LeCun, Y. Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521 (May 2015), pp. 436–44. DOI: 10.1038/nature14539.

[31] Yang Yang Lee and Zaini Abdul Halim. "Stochastic computing in convolutional neural network implementation: a review". In: *PeerJ Computer Science* 6 (2020), e309.

[32] Siting Liu and Jie Han. "Energy efficient stochastic computing with Sobol sequences". In: Mar. 2017. DOI: 10.23919/DATE.2017.7927069.

[33] Zhenzhu Meng, Yating Hu, and Christophe Ancey. "Using a Data Driven Approach to Predict Waves Generated by Gravity Driven Mass Flows". In: *Water* 12 (Feb. 2020). DOI: 10.3390/w12020600.

[34] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012. ISBN: 026201825X.

[35] B. Moons et al. "Energy-efficient ConvNets through approximate computing". In: *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*. Mar. 2016, pp. 1–8. DOI: 10.1109/WACV.2016.7477614.

[36] M. Hassan Najafi and David Lilja. "High Quality Down-Sampling for Deterministic Approaches to Stochastic Computing". In: *IEEE Transactions on Emerging Topics in Computing* PP (Jan. 2018), pp. 1–1. DOI: 10.1109/TETC.2017.2789243.

[37] S. Park and Kyungho Chung. "CENNA: Cost-Effective Neural Network Accelerator". In: *Electronics* 9 (2020), p. 134.

[38] Atul Rahman and Kiyoung Choi. "Efficient FPGA Acceleration of Convolutional Neural Networks Using Logical-3D Compute Array". In: Jan. 2016, pp. 1393–1398. DOI: 10.3850/9783981537079_0833.

[39] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[40] H. Sim and J. Lee. "A new stochastic computing multiplier with application to deep convolutional neural networks". In: (2017), pp. 1–6.

[41] M. Sun and W. P. Tay. "Inference and data privacy in IoT networks". In: *2017 IEEE 18th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*. 2017, pp. 1–5. DOI: 10.1109/SPAWC.2017.8227701.

[42] M. Verhelst and B. Moons. "Embedded Deep Neural Network Processing: Algorithmic and Processor Techniques Bring Deep Learning to IoT and Edge Devices". In: *IEEE Solid-State Circuits Magazine* 9.4 (Fall 2017), pp. 55–65. ISSN: 1943-0590. DOI: 10.1109/MSSC.2017.2745818.

[43] Joseph Walsh et al. "Deep Learning vs. Traditional Computer Vision". In: Apr. 2019. ISBN: 978-981-13-6209-5. DOI: 10.1007/978-3-030-17795-9_10.

[44] Chris Winstead. "Tutorial on Stochastic Computing". In: Feb. 2019, pp. 39–76. ISBN: 978-3-030-03729-1. DOI: 10.1007/978-3-030-03730-7_3.

[45] Y. Xie et al. "Fully-Parallel Area-Efficient Deep Neural Network Design Using Stochastic Computing". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 64.12 (Dec. 2017), pp. 1382–1386. ISSN: 1558-3791. DOI: 10.1109/TCSII.2017.2746749.

[46] A. Zhakatayev et al. "Sign-Magnitude SC: Getting 10X Accuracy for Free in Stochastic Computing for Deep Neural Networks*". In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 2018, pp. 1–6.

[47] Jiajun Zhang and Chengqing Zong. "Deep Neural Networks in Machine Translation: An Overview". In: *IEEE Intelligent Systems* 30 (Sept. 2015), pp. 16–25. DOI: 10.1109/MIS.2015.69.

[48] Y. Zhang et al. "Parallel Hybrid Stochastic-Binary-Based Neural Network Accelerators". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 67.12 (2020), pp. 3387–3391. DOI: 10.1109/TCSII.2020.2994464.

[49] J. Zurada. *Introduction to Artificial Neural Systems*. USA: West Publishing Co., 1992. ISBN: 0314933913.