Bewertung des Stochastischen Rechnens für Effiziente Inferenz Tiefer Neuronaler Netze

Javier Ferrer Ortega

Masterarbeit 2024

Rheinland-Pfälzische
Technische Universität
Kaiserslautern
Landau

Department of Electrical and Computer Engineering

Microelectronic Systems Design Research Group

Master Thesis

Evaluation of Stochastic Computing for

Efficient Deep Neural Network Inference

Bewertung des Stochastischen Rechnens für

Effiziente Inferenz Tiefer Neuronaler Netze

Presented:	October 31, 2024
Author:	Javier Ferrer Ortega (425198)
Research Group Chief:	Prof. Dr.-Ing. N. Wehn
Tutor:	M.Sc. Mohammad Hassani Sadi & Dr.-Ing. Christian Weis

Statement

I declare that this thesis was written solely by myself and exclusively with help of the cited resources.

Kaiserslautern, October 31, 2024

Javier Ferrer Ortega

Acknowledgments

To Prof. Dr.-Ing. Wofgang Kunz, for granting the Erasmus scholarship and to Mohammad Hassani Sadi for providing the idea and his kind assistance.

A mis padres: Gracias por todo.

A mi hermano, también. Más o menos. xd

Abstract

This thesis investigates the implementation of stochastic computing in deep convolutional neural networks and its impact on inference accuracy, through the extension of the PyTorch framework's 2D convolution and fully connected layer functions using stochastic computing.

Theoretical foundations cover key concepts such as Stochastic Computing as an important alternative to traditional binary computation, Deep Convolutional Neural Networks design and Parallel Programming with CUDA for Graphics Processing Units (GPUs).

A detailed description of both the CPU implementation and the GPU acceleration of the stochastic functions is provided as well as an explanation on how to use them in PyTorch with the Python programming language.

The evaluation methodology comprises the comparison in terms of accuracy and end-to-end execution times of different stochastic computing representations (Unipolar vs Bipolar) implemented in LeNet5 and VGG9 deep convolutional neural networks while varying the bitstream lengths of the stochastic numbers and the types of random number generators used to generate these stochastic bitstreams.

Finally, the study suggests that Unipolar representation of stochastic numbers is better in terms of overall accuracy and throughput compared to its Bipolar counterpart, as far as the CPU and GPU stochastic computing implementations are concerned.

Kurzfassung

Diese Arbeit untersucht die Implementierung der stochastischen Berechnung in tiefen Convolutional Neural Networks und ihre Auswirkungen auf die Inferenzgenauigkeit durch die Erweiterung der 2D-Faltung und der vollständig verbundenen Schichtfunktionen des PyTorch-Frameworks mithilfe der stochastischen Berechnung.

Die theoretischen Grundlagen umfassen Schlüsselkonzepte wie stochastische Berechnungen als wichtige Alternative zu traditionellen binären Berechnungen, das Design von Deep Convolutional Neural Networks und die parallele Programmierung mit CUDA für GPUs.

Es wird eine detaillierte Beschreibung sowohl der CPU-Implementierung als auch der GPU-Beschleunigung der stochastischen Funktionen gegeben und erklärt, wie man sie verwendet.

Die Evaluierungsmethodik umfasst den Vergleich hinsichtlich Genauigkeit und Ausführungszeiten verschiedener stochastischer Computerdarstellungen (unipolar vs. bipolar), die in den tiefen Convolutional Neural Networks LeNet5 und VGG9 implementiert sind, während die Bitstromlängen der stochastischen Zahlen und die Arten der Zufallszahlengeneratoren, die zum Generieren dieser stochastischen Bitströme verwendet werden, variiert werden.

Schließlich legt die Studie nahe, dass die unipolare Darstellung stochastischer Zahlen im Hinblick auf die Gesamtgenauigkeit und den Durchsatz besser ist als ihr bipolares Gegenstück.

Contents

1 Introduction

When Stochastic Computing first started being proposed as a different approach to traditional binary computation in the late 1960s, it was not known yet the relevance and great potential it could have years later in the fields of error correction, image processing and neural networks computation.[1]
Recent advancements have shown that "in Stochastic Computing (SC) neural networks (NN), hardware requirements and power consumption are significantly reduced by moderately sacrificing the inference accuracy and computation speed." In addition, research suggests that in terms of performance, SC NNs have also improved, making them comparable with conventional binary designs, then again, by utilizing less hardware.[2]
The technical perks that implementing SC NNs in embedded hardware might seem quite advantageous compared to their binary counterparts, however there are still some very important challenges to tackle first.
The concept of Stochastic Computing basically proposes the implementation of stochastic numbers to realize arithmetic functions with very few logic gates in hardware designs. These are shown as bitstreams of ones or zeros that are computed by rather simple circuits. Parallely, the numbers themselves are treated like probabilities. A bit-stream X, for instance, containing 25% of ones and 75% zeros is equal to probability: 0.25, where neither the length nor the structure of S needs to be fixed, therefore (1,0,0,0), (0,1,0,0), and (0,1,0,0,0,1,0,0) are all possible representations of 0.25.[3]
One of the most significant challenges within SC precisely resides in the way that stochastic numbers are represented, or to put it another way: encoded. For a stochastic number interpreted as a probability of 0.25, a bit stream length of only 4 bits (1,0,0,0) is enough to be encoded the most accurately, nonetheless, larger bit streams are needed to represent probabilities with more decimals; for example, the 16-bit stochastic numbers (1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0) and (0,1,0,1,0,1,0,1,0,1,0,1,0,1,1,1) both are equal to 9/16 and possess a 4-bit precision (0.5625).[3]
It is then reasonable to think that the overall precision of the output probabilities in a SC NN depends mostly on the length of the bit-stream sequence, or to express it more simply, the amount of 1s and 0s in the stochastic sequence is what determines whether a SC NN will be able to infer its target accurately or not.
This is precisely one of the main motivations of this work: to evaluate the behavior of a SC NN by obtaining the minimal bit-stream length required to achieve effective inference using different types of input data in a NN with convolutional and linear layers.
SC has also been implemented in the back-propagation algorithm used in NN training [4][5], however the main focus of this project resides in analyzing SC NN inference given that many of the current efforts in implementing NNs in FPGAs and embedded systems are focused towards NN inference mostly [2][6], as currently embedded NN training consumes a great amount of energy and hardware resources.[7][8]
Nevertheless, this work is not led towards the implementation of a hardware design but rather to the development of a software framework that can function as a tool to investigate the impact of using SC on the accuracy of a neural network, throughput, and further hardware implementations.
There exists already powerful and very popular high level software frameworks for Artificial Intelligence and Machine Learning like TensorFlow and PyTorch, nonetheless an extension that involves SC in fundamental operations of a DNN is still needed in order to use the large extent of functionalities already available in python and specifically for this work, a PyTorch extension of a SC Convolution and a SC Fully Connected Layer using multiplication and accumulation of stochastic bit-streams has been developed.

2 Objective

The primary goal of the project in which this thesis is based is to investigate the impact of using stochastic computing on the accuracy of deep neural networks. To achieve this goal, a software framework for basic computation of CNNs using SC was developed. In a first instance, the use of the C++ programming language in order to observe the functionality of the Convolution and Fully connected layer stochastic operations within the CPU, and later the use of CUDA/C++ to accelerate these stochastic operations by taking advantage of the multi-threaded data-parallelism that Graphical Processing Units (GPUs) provide. Subsequently, integration of the developed CUDA/C++ framework with PyTorch is done to carry on with the further evaluation of *Stochastic Computing Deep Convolutional Neural Networks* (SC DCNNs).

3 Theoretical Background

Before the main aspects of this thesis are discussed it is first necessary to make a concise explanation of the 3 core subjects needed to understand and realize this project. This section will summarize the concepts and most important aspects of Stochastic Computing (which has been briefly introduced already), Convolutional Neural Networks inference and training, and lastly the hardware features of GPUs that allow parallel computation and which is exerted through the CUDA programming language.

3.1 Stochastic Computing

As briefly commented before, Stochastic Computing takes advantage of bit sequences or bit streams that are encoded in a random or pseudo-random form to compute arithmetic operations using very simple circuits. These bit streams are called Stochastic Numbers and they are formally defined as "Given a probability px, $0 \leq px \leq 1$, the corresponding stochastic number X is a sequence of random binary numbers X_0, X_1, . . . for which any $Xj \in \{0, 1\}$ may equal 1 with probability px", [1] expressed in other words, to generate a stochastic bit-stream, the input value (which is usually a floating-point number between 0 and 1) is compared with a random number for each bit in the stochastic bit-stream, if the random number is less than the input value, the corresponding bit in the bit-stream is set to 1, otherwise, it is set to 0. For example, to represent the value 0.75 in a stochastic bit-stream, a random number sequence is created: 0.5, 0.8, 0.2, 0.7, 0.1, 0.9, 0.4, 0.6 and then each of these random numbers is compared to 0.75, giving as output a stochastic sequence: 1, 0, 1, 1, 1, 0, 1, 1. This process, represented in Figure 3.1(a), is repeated for as many bits as needed so as to create a precise enough bitstream. In hw designs, stochastic bitstreams are generated by means of random number generators (RNG). It is easier to convert a stochastic number to binary. The value of the stochastic number p is extracted from the amount of ones in the bitstream sequence, then it is just enough to count all the ones to effectively extract the value of p.[3] Figure 3.1(b) shows a counter that performs this conversion.

3.1.1 Stochastic Numerical Formats

In theory, Stochastic bit-streams can represent very large real numbers from -∞ to ∞, however this depends mostly on the format they are defined. For this work in particular, stochastic numbers are only needed in its representation from -1 to 1.

The format to represent the range [-1,1] is called Bipolar Stochastic number, while Unipolar format can directly represent positive real numbers between zero and one.

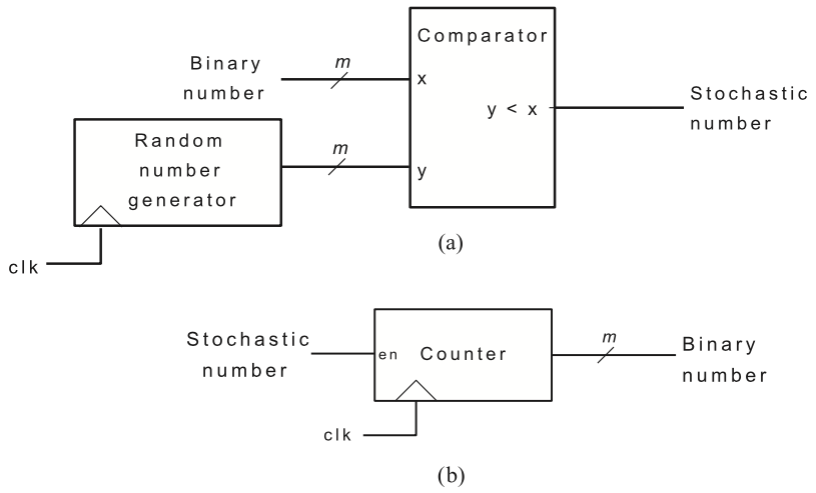


Figure 3.1: Number conversion circuits: (a) binary-to-stochastic; (b) stochastic-to-binary. (Taken from [3])

For real number x with positive real parameter M, such that $-M \leq x \leq M$, the stochastic number X is said to be bipolar if $x = M(2px - 1)$, equivalently, to normalize this stochastic number in bipolar format within the range [-1,1] to a probability value within [0, 1], the formula $px = 0.5\ (1 + x/M)$ is applied. An example of how different formats can represent a stochastic probability is shown in table 3.1; the Likelihood Ratio (LR) format can represent real numbers x from 0 to ∞ into probabilities px if $px = x/(1 + x)$ and converted back to real x with $x = px\ /\ (1 - px)$, in the same way, the Log Likelihood Ratio (LLR) format can represent real numbers x from $-\infty$ to ∞ into probabilities px if $px = e^x\ /\ (1 + e^x)$ and converted back to real x with $x = \log(px\ /\ (1 - px))$.[1]

p_X	Unipolar x	Bipolar x	LR x	LLR x
0	0	−1	0	$-\infty$
0.25	1/4	−1/2	1/3	−1.4
0.33	1/3	−1/3	1/2	−0.7
0.5	1/2	0	1	0
0.66	2/3	1/3	2	0.7
0.75	3/4	1/2	3	1.4
1	1	1	∞	∞

Table 3.1: A comparison of stochastic numerical formats (Taken from [1]).

3.1.2 Stochastic Arithmetic Operations

Like in traditional binary computation, Stochastic Computing can realize arithmetic operations using either single logic gates or a combination of these. This means, for example, that in order to negate a stochastic number, all the bits in a stochastic bit-stream are inverted with a NOT logic gate, but for arithmetic operations like addition and multiplication, SC provides a much more practical approach than those used with binary numbers.

While traditional binary computation uses Full, Half, Ripple-carry or Carry-look-ahead adders to perform addition, SC can use a Multiplexer to add two stochastic bit-streams with the probability of the select signal set to 0.5, even a simple OR gate could be used as an approximate adder for stochastic numbers in their Unipolar format. For Multiplication, binary computing usually uses circuits designed to realize the Booth's multiplication algorithm whereas SC can only use a simple AND gate for numbers in Unipolar format and an XNOR gate for numbers in Bipolar format.[1] Figure 3.2 shows an example of these SC multipliers and a MUX adder, however, in recent hardware designs [2] there have been implemented a couple more SC addition techniques, the first being an accumulative parallel counter (APC)-based that simply adds up or accumulates the 1's of the inputted stochastic bit-streams. The probability of the output signal here is determined only by the sum of the probabilities of the input signals and its output is already in binary representation, so there is no need to implement a stochastic-to-binary converter. The second SC adder to consider is a toggle flip-flop (TFF)-based, but for the purposes of this work, the functionality of the former (APC)-based adder was the one implemented in the Pytorch extension in order to conserve every positive bit of the input bit-streams so there was no loss in the computation accuracy.

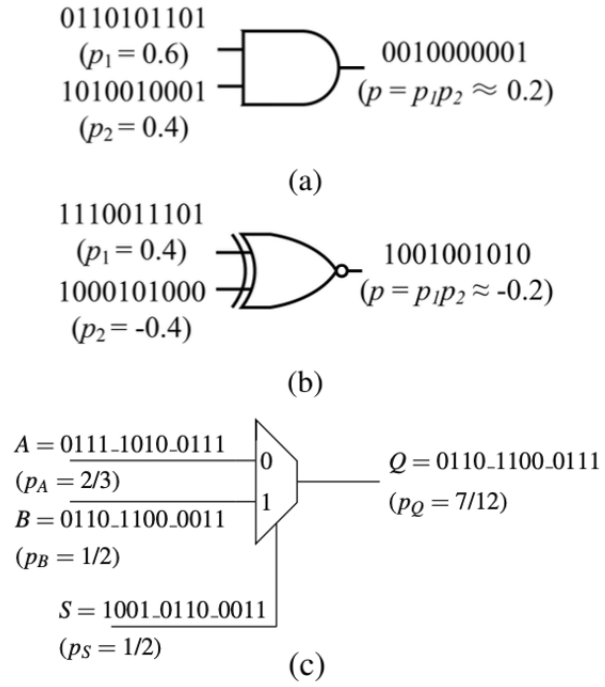


Figure 3.2: (a) SC multiplier for the unipolar representation. (b) SC multiplier for the bipolar representation. (c) The MUX as a weighted stochastic adder in the unipolar format. (Taken from [1] & [2])

Naturally, there are more digital features that can be implemented with SC, like Stochastic Comparators and Median filters used in Image Processing applications [1], or customized activation circuits for Neural Network's forward propagation like a Stochastic ReLU function [9] however, for the implementation of the Stochastic operations introduced in this work, only addition and multiplication of Stochastic numbers suffice given that, specifically for a NN forward propagation, only addition and multiplication are the main arithmetic operations used within a convolution and a linear layer of tensors. The algorithms behind these operations and other important layers of a CNN like Pooling and activation functions will be introduced next.

3.2 Convolutional Neural Network Design

Neural Networks have become perhaps one of the most fundamental components of modern Machine Learning and Computer Vision, they can be found in such a wide range of applications like natural language processing (NLP), autonomous driving and speech recognition. A NN as a model was inspired by the human brain and the way it works. Its nodes and interconnections are like the neurons in our brains. A standard NN has an input layer, an output layer and, between input and output, at least one hidden layer. These can be referred as Fully Connected neural networks, in which all of the units at one layer are connected to all of the units in the next layer. Figure 3.3 shows a basic model of a fully connected neural network.

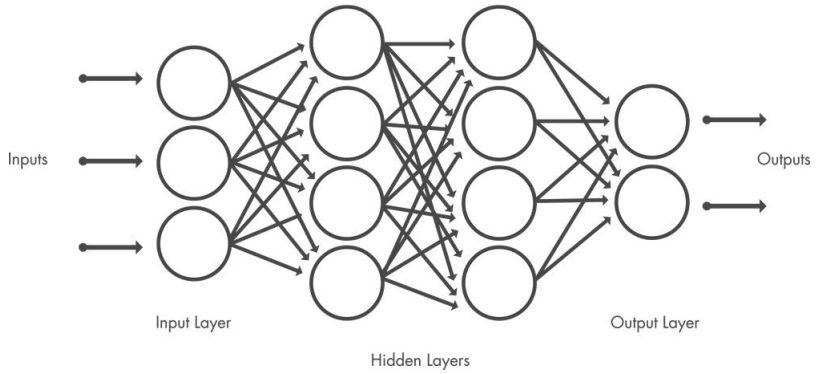


Figure 3.3: Fully connected neural network model. (Taken from [10])

This model, in combination with a set of filtering layers, can be used to classify images or even recognize data patterns like texts and sounds. When this input data is processed by one of these filter layers, the outputs are called convolutional layers. These layers are the 1rst layer of a convolutional network and convolutional layers follow more convolutional layers; then comes the pooling layers with their respective activation functions and the fully connected layer is usually the last layer. The name that acquires this extended NN is Convolutional Neural Network (CNN). With each layer, the CNN increases in its complexity but so do its identification capabilities.

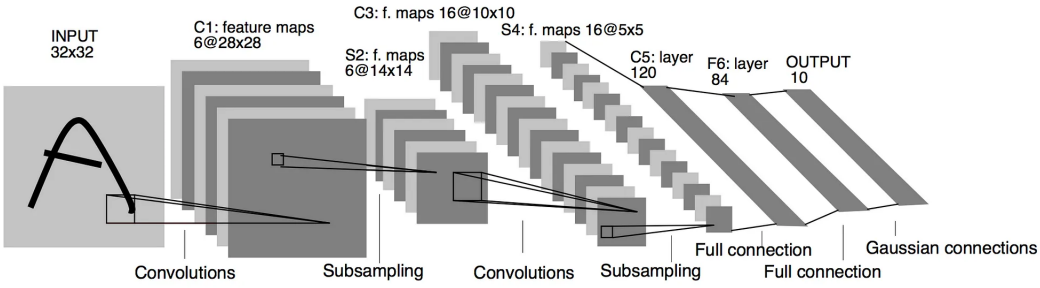


Figure 3.4: Classic convolutional neural network architecture.

Figure 3.4 shows the architecture of LeNet-5, which is known as the classic CNN architecture, developed by Yann LeCun in [11]. This architecture has got two convolutions and three fully-connected (FC) layers, each with their corresponding pooling and activation layers, that can classify a set of greyscale images of numbers from 0 to 9 from the MNIST dataset.

3.2.1 Convolutional layer

The convolution layer is the core of a CNN. If the input data is an image, for example, the convolution puts the input image through a set of convolutional filters, each of which extracts certain features from the image and the output of each convolved image is used as the input to the next layer. In computer vision there are usually convolutions in three dimensions, where 2D convolution is used for images and 3D convolutions are used in multimedia applications, for example. The numerical values of the convolutional filters, also referred as kernels, are called weights and these represent the most fundamental values within a CNN as these determine how the network processes input data and extracts features. The origin of these weights is usually random through Gaussian distribution or other types of initialization techniques [12], nonetheless these have to be modified (or 'trained') iteratively so that the CNN can detect or classify its objective correctly. More on this training process will be commented later but now is important to revise the mathematics behind a 2D convolution, since to understand the behavior of this operation and be able to build a SC version of it, its parameters have to be disclosed too.

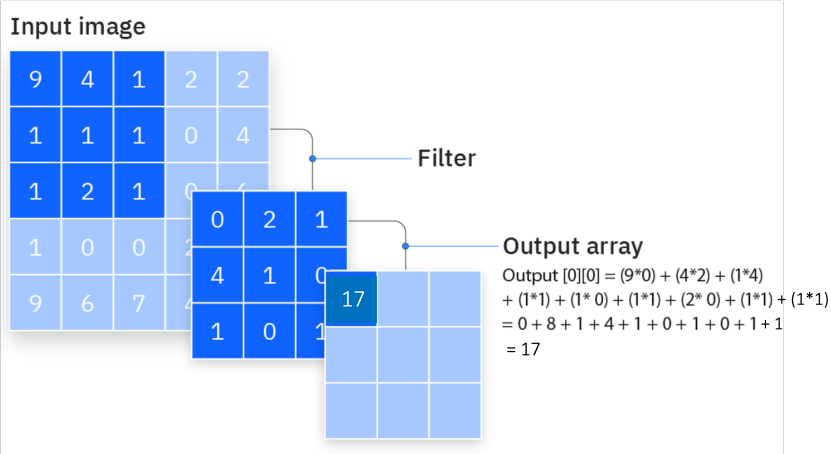


Figure 3.5: Example of 2D convolution arithmetic. (Taken from [13])

The formula that commands the construction of a standard 2D convolutional output layer based on [14] and [15] is as follows:

$$\text{Output}(i,j) = \sum_{m=0}^{k_h-1} \sum_{n=0}^{k_w-1} \text{Input}(i+m, j+n) \times \text{Weight}(m,n) \tag{1}$$

This formula represents the process of sliding the filter (or convolutional kernel) over the input feature map (e.g. an image), performing an element-wise multiplication, and then summing the results to produce each element of the output feature map (the output of the convolution), where:

- Output(i,j) is the value of the output feature map at position (i,j).
- Input(i+m,j+n) is the value of the input feature map at position (i+m,j+n).
- Weight(m,n) is the value of the kernel (or convolutional filter) at position (m,n). k_h and k_w are the height and width of the kernel, respectively.

Although there are more *hyperparameters* like Padding, Stride, Dilation and Bias that mean an important part of a convolution, these can be applied externally to the input or output feature map and can be safely ignored in the construction of the SC operations of this project's PyTorch extension, however their definitions might be of importance in further implementations so they can be consulted in [16] and [17].

3.2.2 Pooling & Activation layers

The pooling layer is usually found after a convolution and before a FC layer in the typical CNN architecture. Its function is to decrease the height and width of the inputs so that number of parameters and computation in the network is also decreased. There are several types of pooling like Max, Average, Global and Stochastic pooling but max pooling is the one that is commonly implemented. A clear example of the operation can be seen in Figure 3.6 and the equation for max pooling as expressed in [18] and [19] is as follows:

$$\text{Output}(i,j) = \max_{m=0}^{k_h-1} \max_{n=0}^{k_w-1} \text{Input}(s_h \cdot i + m, s_w \cdot j + n) \tag{2}$$

- Output(i,j): This is the value of the pooled output at position (i,j).
- *max*: The max operation selects the maximum value within the pooling window.
- k_h and k_w : These represent the height and width of the pooling window, respectively.
- $\text{Input}(s_h \cdot i + m, s_w \cdot j + n)$ is the value from the input feature map at position $(s_h \cdot i + m, s_w \cdot j + n)$.
- m and n iterate over the height and width of the pooling window.
- s_h and s_w are the strides in the height and width directions, determining how far the pooling window moves between regions. The stride determines the step size with which the pooling window slides over the input feature map.

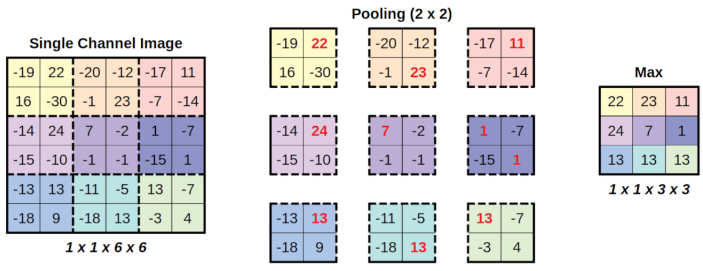


Figure 3.6: Example of Max Pooling arithmetic. (Taken from [19])

Given that real-world data is often complex and non-linear, there must exist non-linearity in the CNN's model for it to be able to learn diverse features and complex representations from data. To introduce non-linearity in a CNN, activation functions are used on the output values of a convolution or a FC layer. These functions ought to be differentiable in order to enable error backpropagation to train the model and can also be used to normalize the output of a neuron in a FC layer or a convolution to a specific range.

Basically activation functions map weighted inputs into a neuron output, in other words, the activation function decides whether a neuron will fire or not for a given input by producing an output value of some strength. The most commonly used activation functions in convolutional neural networks are:
Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$ whose outputs are mapped between 0 and 1, Hyperbolic tangent: $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ with outputs range between -1 and 1, Rectified Linear Unit: ReLU(x) = max(0,x) with a range $[0, \infty]$, Leaky ReLU with range $[-\infty, \infty]$ and Softmax$(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$ with range (0,1).

3.2.3 Fully Connected layer

The final layers of a classic CNN are Fully Connected (FC) layers, already introduced in the beginning of this section, and they are meant mainly for classification of the input features. Before a FC layer, the input data must be flattened as a single dimensional vector, where each element of this input vector could be referred as an input 'neuron', and the output values of a FC layer can be referred as output probabilities or output 'neurons'. Each output neuron is connected to every value of the input vector, as represented on Figure 3.7(b), and all the output neurons or probabilities of a single FC layer can be the inputs of a hidden layer, as represented on Figure 3.3, or they can become the final probabilities that will classify or detect the type of input data at the beginning of the CNN.
The formula that commands the construction of a standard FC layer is based on an affine linear transformation in [20]:

$$z_j = \sum_{i=1}^{n} w_{ji} \cdot x_i + b_j \tag{3}$$

Where z_j is the output of the j-th neuron, x_i is the i-th input to the layer (from the previous layer), w_{ji} is the weight connecting the i-th input to the j-th neuron, b_j is the bias term for the j-th neuron and finally n is the number of inputs to the layer. A non-linear transformation is then applied to z_j through a non-linear activation function f, as represented in the example of Figure 3.7(a), where an input vector of size nine outputs a vector size of four neurons.

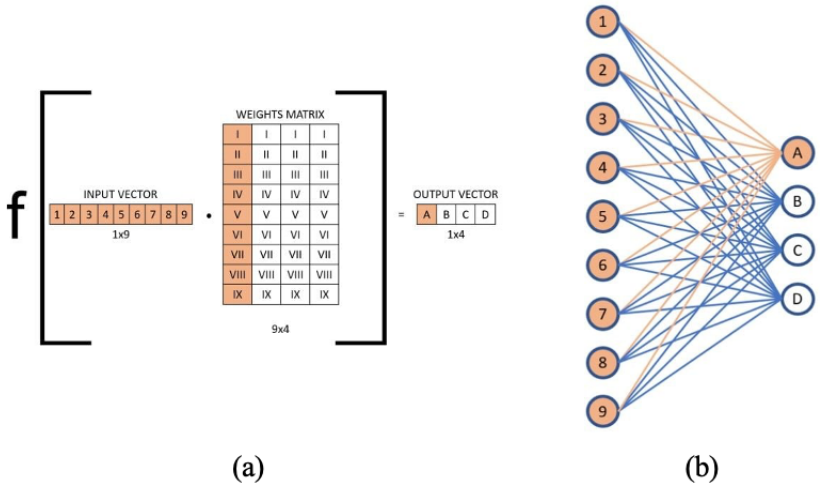


Figure 3.7: (a) FC layer's input multiplied by the weights matrix and its output vector. (b) FC layer's input and output neurons. (Taken from [21])

3.2.4 Backpropagation, Inference & SC CNNs

The whole process from the very first convolution layer of a CNN all the way to the final output neurons or class probabilities of a FC layer is defined as **Forward Propagation** however, to achieve an accurate prediction or classification of the input data, also defined as **Inference**, the weights of both the convolutional kernels and the FC layers must be modified, adjusted or rather *trained* several times with a training dataset so that the inference of a CNN is correct with the most diverse amount of test data possible.
This CNN training process is realized by means of the **Backpropagation algorithm** in a process referred as **Backwards propagation**.
Even though thorough understanding of the Backwards Propagation process is not required for the matters of this work as SC NNs are mainly being implemented in embedded systems for inference purposes, it is important to summarize its most important aspects as this algorithm constitutes the base in which all NNs obtain their inner reliability for ML applications.
After the output probabilities of a CNN are produced by the forward propagation, they are measured against a ground truth by using algorithms referred as loss functions like cross-entropy. This loss function measures the difference between the predicted output and the true label, quantifying how far the predictions are from the actual values. Then the backprop operates the gradients of the loss function considering each weight, bias and activation function in the network, layer by layer, starting from the output layer and moving backward to the input layer. The gradients that result from the computations are implemented later to update new network's weights and biases. It is an optimization algorithm like Stochastic Gradient Descent (or a variant like Adam) that updates the weights by subtracting the gradient multiplied by a learning rate, effectively moving the weights in the direction that reduces the loss. Finally, the process of forward pass, loss calculation, and backward pass is repeated over many epochs (iterations over the entire training dataset), so that over time, the network's weights are adjusted to minimize the loss function, improving the accuracy of the network's predictions on a training dataset. [22]
Unlike training, **inference** does not re-evaluate or adjust the weights of a CNN based on the results. Inference applies knowledge from a trained neural network model and a uses it to infer a result. So when a new unknown data set is input through a trained neural network, it outputs a prediction based on the accuracy of the neural network provided by backpropagation.
Much more can be stated and discussed about CNNs, their architecture, characteristics, applications and so forth, nevertheless this is such a deep and quite widespread subject which can be studied in more depth in [22] and [23] for example, so for now, to properly comprehend how Stochastic Computing can work synergistically within CNNs and hopefully Deep Convolutional Neural Networks too, that is the purpose of the following chapters.
In the context of Stochastic Computing Convolutional Neural Networks (SC CNN), the overall accuracy of the output probabilities of the last FC layer is mainly determined by the precision of every stochastic bitstream used in the stochastic operations (e.g., addition and multiplication) that is: the more precise the stochastic numbers are, the closer the accuracy of the classifications of the SC CNN inference will be in respect to the 'normal' accuracy of a trained CNN using binary computation. What determines the precision of the stochastic numbers is, then again, the length of every stochastic bitstream, therefore it is reasonable to deduce that in a SC CNN, the inference accuracy is directly dependent on the bitstream length selected to represent every stochastic number.

3.3 Parallel Programming with GPUs

The core of the project that this thesis is presenting is the PyTorch extension of the Convolution and FC layer operations using Stochastic Computing. In a first instance these have been implemented in the C++ programming language as proof of concept to demonstrate the correct functionality of the initial functions, however, when these operations are implemented in code for CPU it is demanded to use an amount of control flow statements like nested while or for loops that explode in time complexity.
For instance, a 2D convolution implemented for a single or multi-core CPU has a time complexity of: $O(C_{in} \times C_{out} \times k_h \times k_w \times H_{out} \times W_{out})$ where: C corresponds to input and output channels, k is width and height of kernel and H and W are height and width of the output feature map, while for a 3D convolution the time complexity takes as well the depth of the kernel and the output feature map. Simultaneously, a Fully connected layer be having a complexity of: $O(n_{in} \times n_{out})$ where n is number of input and output neurons. [24]
When this operations are combined with more layers in deep CNN models with large datasets and large amounts of processing data, the overall computation time can be way too much for any functional application. When taking into account that Stochastic Computing even adds one more level of time complexity to these operations, then it becomes mandatory to accelerate the overall execution time and what a better way to do this than by means of the multi-threaded data parallelism that Graphic Processing Units (GPUs) provide, thus understanding of the CUDA programming language fundamentals implemented in this work is important, as well as the overall high-level hardware that embodies GPUs, taking as main references the teachings and up-to-date existent documentation of NVIDIA [25], [26] and others [27], [28].

3.3.1 GPU Architecture

The core processing units of a GPU are the so called streaming multiprocessors (SM), which supervise a certain number of ALUs. The number of SMs used depends on the number of threads called in a CUDA kernel.

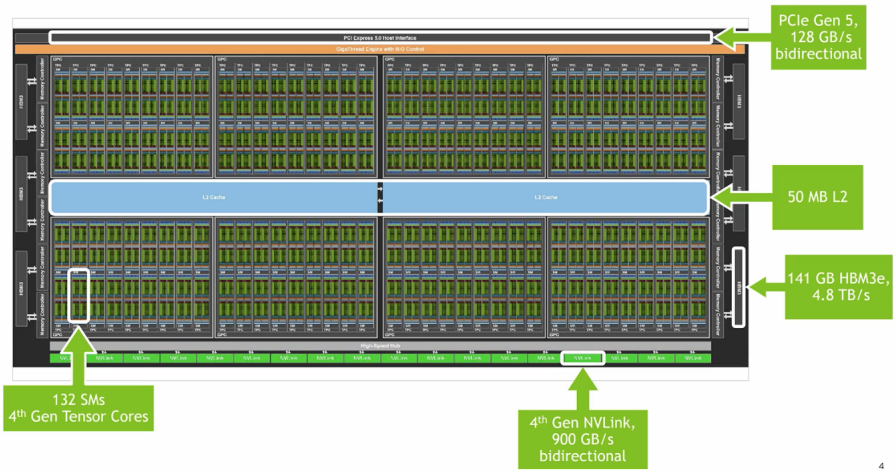


Figure 3.8: GPU overview of NVIDIA H200 SXM (Taken from [25])

Figure 3.8 shows an NVIDIA GPU based on the Hopper architecture with 128 SMs coupled with 141 GB of HBM (High Bandwidth Memory) at a high bandwith of 4.8 TB/s and 50 MB of L2 cache.

Each SM is organized into 4 sub-partitions, represented in Figure 3.9(a), that share an instruction and data cache, they each have a large register file as well as multiple arithmetic units for integer and single or double floating point operations along with 4 tensor cores which are specialized units for mixed precision matrix multiplication-and-accumulate operations, used mostly in deep learning applications.
While multi-core CPUs may work completely independent based on a Multiple Instructions Multiple Data (MIMD) system classification, each subpartition of a Streaming Multiprocessor is based on the SIMT architecture: Single-Instruction Multiple-Thread, which is a combination of SIMD (Single-Instruction Multiple-Data, in which all ALUs in an SM execute the identical line of code) with hardware multi-threading. The SIMT architecture does many things from creation to execution of large sets of hardware threads which are partitioned into groups called **warps**, each warp exposing 32 parallel threads, represented in Figure 3.9(b).

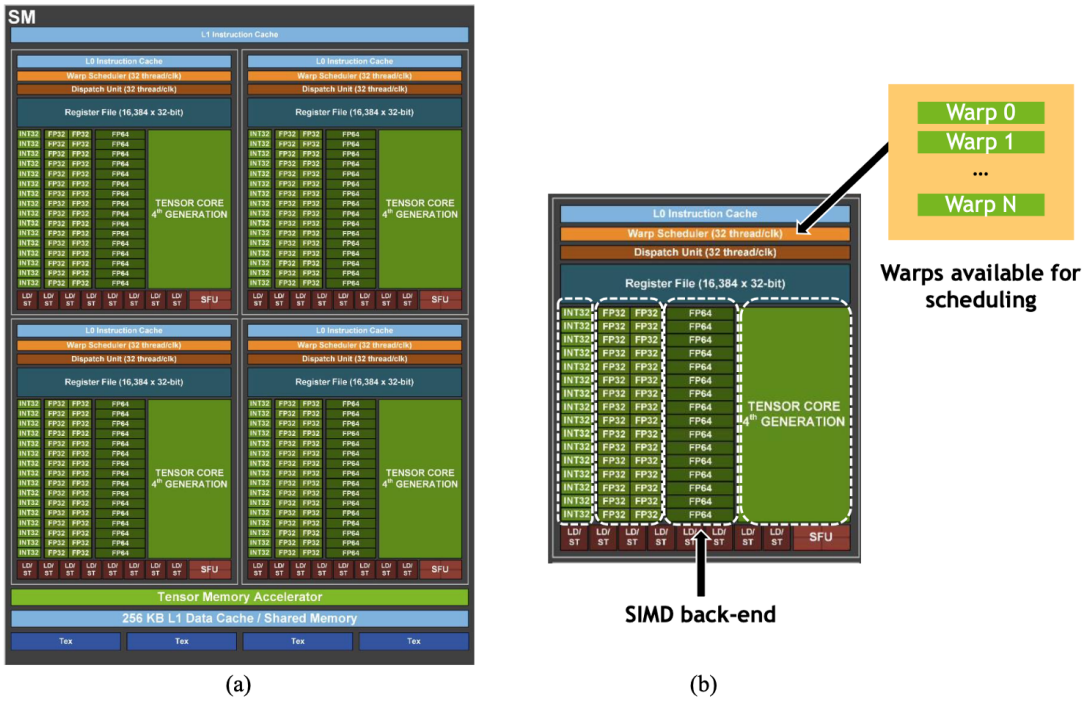


Figure 3.9: (a) SM with 128 FP32, 64 FP64, 64 INT32 cores resp., and 64k 32-bit registers. (b) SIMT threads are partitioned into groups called **warps**. (Edited from [25])

The importance of warps arises from the need to reach full computation efficiency which is achieved when all 32 threads of a warp agree (converge) on their execution path, as observed in Figure 3.10, where all 7 thread IDs should be active at the same time for full efficiency. Having more active warps per SM helps computation efficiency too.
The max. amount of warps that are concurrently active on an SM depends on the capabilities of the **GPU Device**: **Achievable** (depending on the CUDA kernel implementation) and **Achieved** (depending on the size of the *grid*) active warps, in this context, the most important way to increase active warps is by increasing *Occupancy*, given by:

$$\text{Occupancy} = \frac{\text{Achievable no. of active warps per SM}}{\text{Device no. of active warps per SM}} \tag{4}$$

Correspondingly, this occupancy is limited by: **Register usage** (partitioned among threads), **Shared Memory usage** (partitioned among thread blocks) and **Thread block size**.

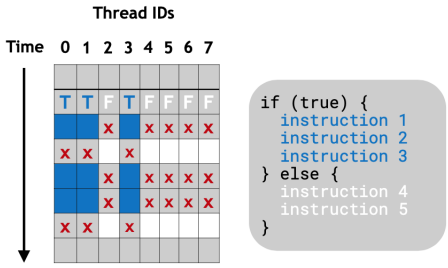


Figure 3.10: Example of warp divergence. (Taken from [25])

Then, to enable an efficient parallel implementation, there are three key software abstractions to consider: 1. a hierarchy of thread groups, 2. memory spaces and 3. synchronization.

3.3.2 Thread Hierarchy

There are 3 levels defined in the thread hierarchy: Grid, Thread Block and the individual Thread. A CUDA kernel is launched on a grid of thread blocks which are completely independent, then each of these thread blocks are executed on a SM, which can have multiple thread blocks running concurrently or in series and finally the individual threads execute on scalar CUDA cores. An additional level in the hierarchy was added with the Hopper architecture known as Thread Block Clusters. They assure proper scheduling of of thread blocks of a Grid.

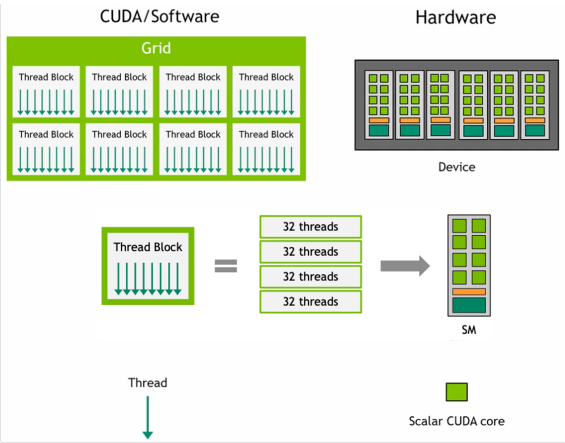


Figure 3.11: Thread Hierarchy representation. (Edited from [25])

At runtime, a block of threads is divided into warps containing threads with consecutive thread IDs starting from a thread with identifier equal to zero. The # of thread blocks in a Grid and the number of threads per block is passed programmatically through a CUDA kernel.
If the thread block size is not equal to the warp size (32), the hardware rounds up the number of threads and deactivates the rest of the unused threads during execution, which impacts computation efficiency, for this reason it is always better to pass a multiple of the warp size when declaring the thread block size in a CUDA kernel in terms of thread resource utilization.

3.3.3 Memory Hierarchy and Access Mechanisms

CUDA defines per-thread **registers**, which give the lowest possible latency when used in a CUDA kernel, but there also exists per-thread **local memory**, used for statically allocated data and its backed in the DRAM (global memory) therefore it is slower than registers. Then, at the thread block level, there is **shared memory** which is visible by all threads within a block so that data can be exchanged between threads within a thread block and, although slower than the registers, it is faster than the DRAM (global memory). Lastly, there is **global memory** which is visible by all threads within the Grid and contains read-only memory types: texture and constant, the former being cached in L1 and the latter in the constant caches.

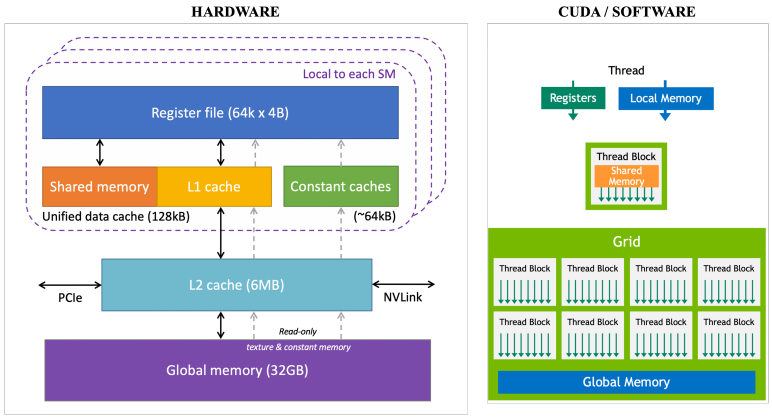


Figure 3.12: GPU Memory Hierarchy representation. (Edited from [25] & [29])

The L2 Cache shown in Figure 3.12 is used mostly for "smoothing" misaligned access patterns, caching common data accessed by multiple threads and allow faster access to local memory.

The perfect scenario to access global memory is when all the threads in a warp are aligned and sequential, also called *coalesced*, as observed on Figure 3.13(a), (b) and (c); the worst scenario happens when each thread of a warp touches a different sector of global memory, as observed on Figure 3.13(f) where the accesses are aligned but strided.

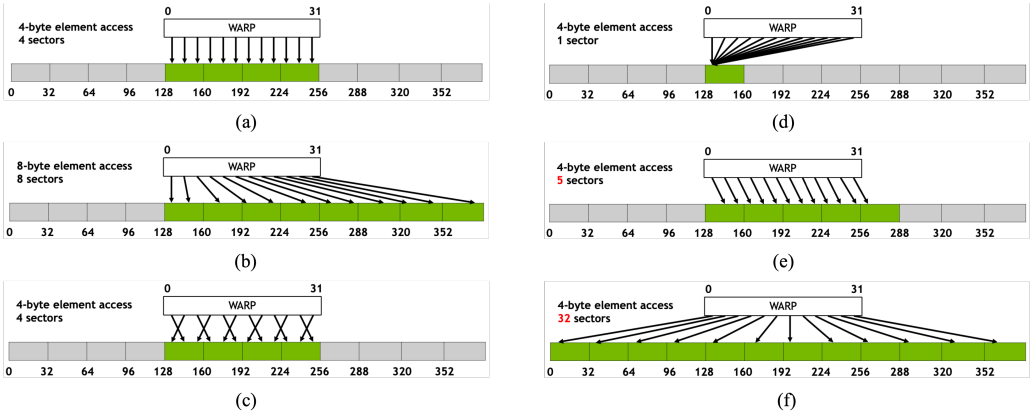


Figure 3.13: Global Access Patterns. (a) & (b) aligned and sequential and (c) aligned and non-sequential are examples of coalesced accesses. (d) access to same address, (e) mis-aligned and sequential, (f) aligned and strided are examples of non-coalesced accesses. (Edited from [25])

Finally, the Unified L1 and Shared Memory can be either a hardware managed cache or an user managed memory and it is divided into 32 banks, each 4-byte wide. The perfect scenario to access data in shared memory is when threads in a warp access different banks contiguously (referred as coalesced too), as shown on Figure 3.14(a). The worst scenario happens when every thread tries to write data to shared memory with a stride, so half of the threads of a warp compete against the other half, as shown on Figure 3.14(b).

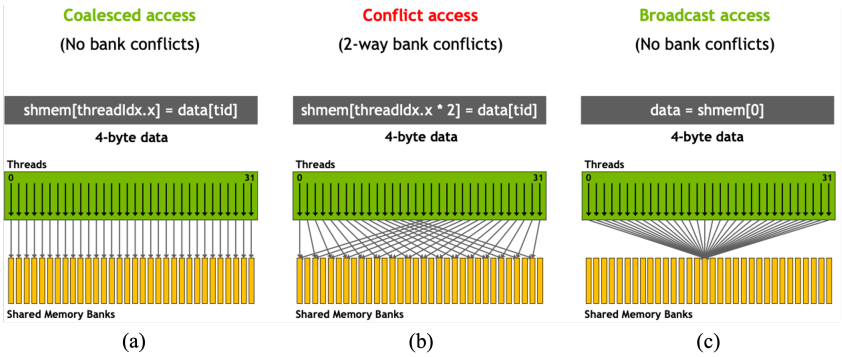


Figure 3.14: Shared Memory Access Patterns. (a) Fully coalesced access. (b) Bank conflicts. (c) Same bank access. (Taken from [25])

NVIDIA provides profiling tools like Nsight Compute or Nsight Systems to get detailed insights into memory access patterns, performance metrics like *occupancy*, bandwidth or latency and highlight any uncoalesced accesses etc., however for simple CUDA kernels sometimes basic code review accompanied by kernel execution time measurement is enough to estimate proper performance.

3.3.4 Synchronization

Since individual threads in a warp have their own program counter and call stack, they are free to execute independently so it cannot be assumed that threads in a warp are automatically re-converged after a conditional or at any point, that is why it is needed to synchronize the threads according to a hierarchy of barriers: 1. At the Grid level, threads can be synchronized when the CUDA kernel finishes execution however it is rather slow: $grid\_group :: sync()$ 2. At the thread block level, which is faster and very common: $\_\_syncthreads()$ and 3. At a warp or sub-warp level which is the fastest way for threads synchronization: $\_\_syncwarp()$.

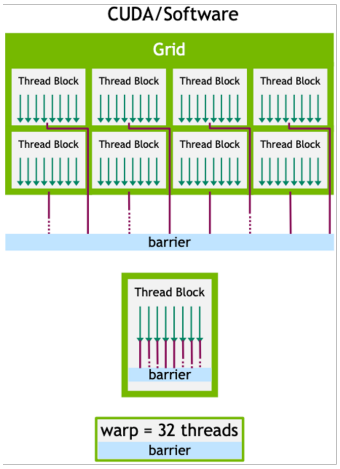


Figure 3.15: Sync barriers. (Taken from [25])

3.3.5 CUDA Programming Model

CUDA follows Single-Program Multiple-Data model of computation which means that scalar code is written (commonly processing only one data item at a time) but this is launched on thousands of threads that execute in parallel, each thread using its own thread index to figure out which data to work on or which execution path to follow.

The most important CUDA element where algorithms are to be parallelized with thread indexes is the CUDA kernel. The way to call threads indexes programmatically is through the variables in Table 3.2.

Table 3.2: Thread Indexing [28]

Variable	Limit	Meaning
$threadIdx.x$	maxThreadsPerBlock	ID of threads in a group
$blockIdx.x$	maxGridSize	ID of groups
$blockDim.x$	User defined	Actual number of threads in a group
$gridDim.x$	User defined	Actual number of groups

The following CUDA Listing 3.1 shows effectively a simple example of a CUDA kernel that adds two long vectors.

```
#define N (33 * 1024)

__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}
```

Listing 3.1: Addition of vectors CUDA kernel [28]

Line **int tid = threadIdx.x + blockIdx.x * blockDim.x;** is precisely how the global thread ID (tid) is computed, which is the unique identifier for each thread across all blocks in the grid. Each thread needs a unique tid to work on different elements of the input arrays. Line **while (tid < N)** is also fundamental because it checks whether a thread's offset is actually between 0 and N before it is used to access the input and output arrays. In this simple example, N is basically the number of elements in arrays 'a', 'b' and 'c' and consequently it is also the total amount of threads to be launched.

CUDA is written in *.cu files. The NVIDIA compiler **nvcc** separates CUDA-code from C++ code to compile the GPU code independently, the simplified compilation process to get an executable is shown on Figure 3.16.

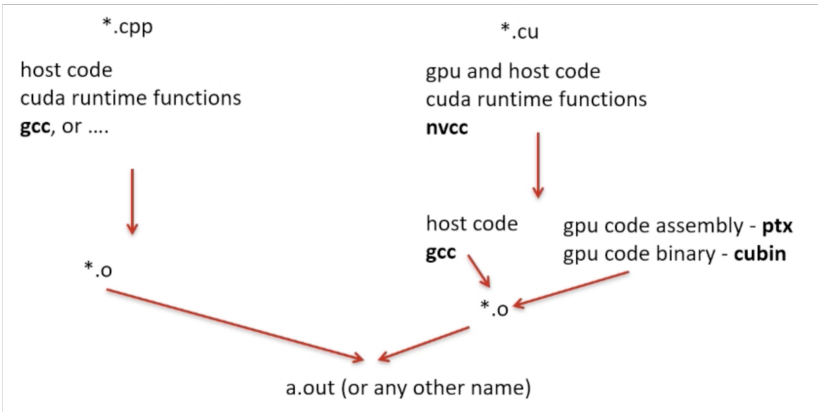


Figure 3.16: CUDA file compilation.

Table 3.3 shows some essential code keywords and functions commonly found in any CUDA script and most importantly, used in the files relating to this work.

Table 3.3: Essential CUDA Keywords found in this work [25], [27], [28]

Keyword	Use
__global__ void	A global function that runs on the GPU and is called from the CPU. Used to define CUDA kernels.
__device__ void	A device function that runs on the GPU and can only be called from within kernels or other device functions. It cannot be called directly from CPU.
cudaMalloc	Allocates memory on the GPU (device memory). E.g **cudaMalloc(&var, size);**
cudaMemcpy	Copies data between host (CPU) memory and device (GPU) memory. It is a blocking function, meaning it waits for the copy operation to complete before returning. E.g **cudaMemcpy(dest, src, size, cudaMemcpyHostToDevice);**
cudaMemcpyAsync	Performs the same operation as cudaMemcpy, but asynchronously. It does not block the CPU and returns immediately. The transfer is performed in a stream, allowing overlapping with kernel execution. E.g **cudaMemcpyAsync(dest, src, size, cudaMemcpyHostToDevice, stream);**
cudaDeviceSynchronize	Blocks the CPU until all previously issued commands on the GPU have completed. Often used to ensure that all kernels and memory transfers are done before proceeding. E.g **cudaDeviceSynchronize();**
cudaFree	Frees memory that was allocated on the GPU using cudaMalloc. E.g **cudaFree(var);**
cudaMemcpyHostToDevice	A flag used with cudaMemcpy or cudaMemcpyAsync to indicate that data is being copied from host memory (CPU) to device memory (GPU). E.g **cudaMemcpy(dest, src, size, cudaMemcpyHostToDevice);**
cudaMemcpyDeviceToHost	A flag used with cudaMemcpy or cudaMemcpyAsync to indicate that data is being copied from device memory (GPU) to host memory (CPU). E.g **cudaMemcpy(dest, src, size, cudaMemcpyDeviceToHost);**
__constant__	Declares a variable in constant memory, useful for storing values that don't change during kernel execution. E.g. **__constant__ float constVar;**
cudaMemset	Initializes or sets the memory on the device (GPU) to a specific value. E.g. **cudaMemset(var, 0, size);**
cudaMemcpyToSymbol	Copies data from host memory to a constant memory symbol on the device. E.g. **cudaMemcpyToSymbol(constVar, src, size);**
cudaDeviceProp	A structure that holds various properties of a CUDA-capable device (e.g., GPU name, memory size, etc.). E.g. **cudaDeviceProp prop;**
cudaGetDeviceProperties	Retrieves the properties of a GPU and stores them in a cudaDeviceProp structure. Useful for getting information about the available GPU. E.g. **cudaGetDeviceProperties(&var, device_id);**
*__shared__**	Declares statically allocated shared memory in a CUDA kernel, that is, the size of the memory must be known at compile time. E.g. **__shared__ int sharedArray[256];**
*extern __shared__**	Declares dynamically allocated shared memory, where the size is specified at runtime (during kernel launch), allows the size of the shared memory array to be flexible and typically used when the amount of shared memory needed is not known until the kernel is called. E.g. **extern __shared__ int sharedArray[];**

* Even though *shared memory* is not actually used in the SC operations implemented in this work, its code keywords are still mentioned here as shared mem is a fundamental component of performance optimizations related to *thread coalescing* or commonly used in parallelized operations like matrix transpositions and vector reductions.

3.3.6 Performance: ILP vs TLP

To try to maximize the performance of a CUDA kernel, one straightforward technique is by hiding either compute or memory access latencies, this means to have enough concurrency which is equal to $Bandwidth \times Latency$. For example, for a FP32 unit that can issue 8 $\frac{\text{operations}}{\text{cycle}}$ and that has a compute latency of 24 cycles, the concurrency is $8 \times 24 = 192$ operations. This means that to make the most of this FP32 unit, the device should be *saturated* with 192 *in-flight* operations. There are two ways to increase in-flight instructions and therefore to hide latencies: 1. By improving *Instruction-Level Parallelism* (ILP) meaning to have more independent instructions per thread or 2. by improving *Thread-Level Parallelism* (TLP) which means using more threads and in consequence having more independent instructions per kernel.

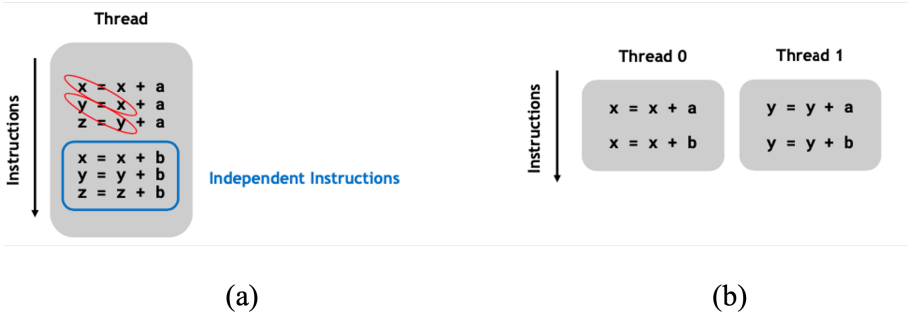


Figure 3.17: (a) Instruction-Level Parallelism (ILP) / (b) Thread-Level Parallelism (TLP). Taken from [25]

According to [30], peak performance in a GPU can be effectively reached if the ALUs (CUDA cores) are kept busy with a combination of ILP and TLP, in other words, a 100% GPU utilization can be reached with the use of as many independent instructions per thread as possible (ILP) and by having more active warps per SM (ideally 100% occupancy) as presented in Figure 3.18, nonetheless, there must be a balance when combining ILP and TLP; if a thread uses many registers to store temporal values, it won't be possible to launch many more threads per kernel given that a thread has a limit on the maximum number of registers available it can use. If a thread's register usage exceeds the available per-thread limit, the excess data is spilled into local memory, which resides in global memory and incurs significant performance overhead due to higher latency and lower bandwidth compared to registers.

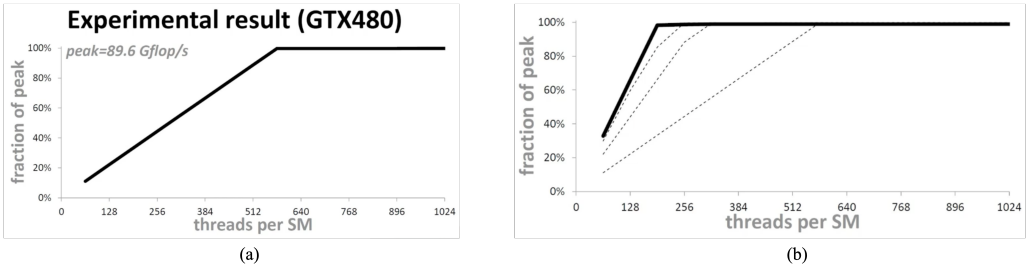


Figure 3.18: (a) No ILP: 100% utilization reached with 576 threads (b) ILP=4: 100% utilization reached with 192 threads. Taken from [30]

In summation, an important performance consideration to take is either having more registers per thread allowing more independent instructions, or having fewer instructions per thread but more active threads running per kernel.

3.3.7 CUDA Streams

CUDA Streams is an important feature that CUDA provides to improve overall parallelization, and is implemented in the project related to this work. Streams concerns the concurrent execution of CUDA kernels using streams. These are processed by overlapping kernels which is very useful if different requests need different resources, that is, a kernel which may use different data or execute different algorithms from other kernels can be overlapped with those other kernels concurrently. A representation can be observed in Figure 3.19.

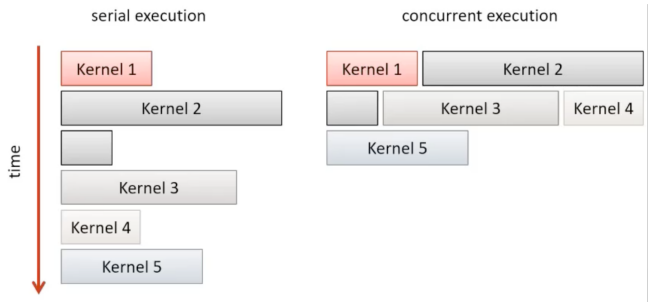


Figure 3.19: Serial vs concurrent execution of CUDA kernels. Taken from [27]

An example on how to set-up a single CUDA stream can be observed in Listing 3.2. If an algorithm is to be computed in two parts due to large volumes of data, first the streams must be created, then copy to device the 1rst half of the data in stream1 with *cudaMemcpyAsync*, start 1rst kernel execution while copying the 2nd half of the data to device and finally copying outputs of the 1rst kernel back to host while executing the 2nd kernel. A synchronization call is introduced after all streams finish execution with *cudaStreamSynchronize*() to assure all the data from every stream is properly copied back to host or using *cudaThreadsSynchronize*() to assure all threads on the card (including the copies) are finished.

```
int main() {
    int *h_data, *d_data;
    cudaStream_t stream1;
    // Allocate pinned (page-locked) host memory
    cudaMallocHost(&h_data, N * sizeof(int));
    // Initialize host data
    for (int i = 0; i < N; ++i) { h_data[i] = i; }
    // Allocate device memory
    cudaMalloc(&d_data, N * sizeof(int));
    // Create streams
    cudaStreamCreate(&stream1);
    // Asynchronously copy data from host to device in stream1
    cudaMemcpyAsync(d_data, h_data, N * sizeof(int), cudaMemcpyHostToDevice, stream1);
    // Launch kernel on stream1
    kernel1<<<(N + 255) / 256, 256, 0, stream1>>>(d_data);
    // Wait for the stream to complete their tasks
    cudaStreamSynchronize(stream1);
    // Copy the result back to host
    cudaMemcpyAsync(h_data, d_data, N * sizeof(int), cudaMemcpyDeviceToHost, stream1);
    // Free resources
    cudaFreeHost(h_data);
    cudaFree(d_data);
    // Destroy the streams
    cudaStreamDestroy(stream1);
    return 0;
}
```

Listing 3.2: Setting-up a CUDA Stream

3.3.8 GPU Capabilities & CUDA Features

Evidently, there are many more capabilities that a GPU provides when it comes to make the most of its overall architectures like Tensor Cores, shared Memory, thread block clusters and so on; however, in the context of this work only the capabilities already introduced in previous sections are needed for the proper understanding and implementation of the *Stochastic Computing* (SC) functions with CUDA & C++. Nevertheless, it may surge an eagerness to inquire why not using GPU Tensor cores to implement a SC CNN; as these are specialized hardware units designed to accelerate mixed-precision computations of matrix multiplications in deep learning operations, they may seem like a very good approach however, for the purposes of the SC functions developed in this work, they are really just an overkill. As it will be elaborated later, the multiply-accumulate operations of Stochastic Computing are bitstreams based and only require a logic gate (AND/XNOR) for multiplication and a counter for addition/accumulation.
Anyways, it is important to give special mention to those other GPU capabilities and CUDA features deemed to be of importance in future implementations, as can be observed in Table 3.4.

Feature	Description
Tensor cores	Tensor Cores are designed to perform matrix-matrix multiplications extremely fast. They typically use 16-bit floating point (FP16) or 8-bit integer (INT8) formats for input matrices and the results of the matrix multiplication are accumulated in 32-bit floating point (FP32) precision to ensure accuracy.
Shared memory	Shared memory in a GPU is a type of fast, on-chip memory that is accessible by all threads within the same block. It acts as a user-managed cache that can significantly improve performance by reducing the number of accesses to slower global memory, however, it requires careful synchronization and memory access pattern to avoid bank conflicts.
Texture memory	Specialized form of memory designed to optimize access patterns for specific types of data with 2D or 3D spatial locality (i.e., nearby threads access nearby data points), texture memory performs better than global memory due to the spatially aware caching mechanism. For example, when a thread accesses data from a texture, the cache can also load nearby elements, increasing the likelihood that subsequent accesses will hit the cache.
Unified Memory	This feature eases programming by automatically migrating memory from CPU to device, however, the downside is that there is no clear synchronization between the data copies to and from host and device. To use unified memory the function $cudaMallocManaged()$ is in charge of data transference but function $cudaDeviceSynchronize()$ must be used after kernel definition to make sure output data can be used in CPU.
Thread block clusters	Groups several thread blocks together into a cluster, allowing them to communicate and synchronize with each other, particularly useful for algorithms that require Inter-block collaboration without going to global memory, faster synchronization and shared memory usage across multiple blocks.
Atomic functions	Special operations that allow threads to safely read, modify, and write shared data in global or shared memory without causing race conditions. Ensure that the read-modify-write sequence is performed atomically, meaning no other thread can interfere during this operation. E.g. atomicAdd(), atomicSub().
Dynamic Parallelism	With dynamic parallelism, a running kernel (called the parent kernel) can launch child kernels. This eliminates the need to go back to the CPU (host) to launch more work, reducing CPU-GPU synchronization overhead. This can significantly improve performance in certain applications, especially those involving hierarchical or recursive algorithms.

Table 3.4: Some GPU capabilities & CUDA features (Summarized from [26])

4 Methodology and Implementation

The two most common types of networks one will often encounter when reading about ML or deep learning are *Fully Connected Neural Networks* (FCNNs), and *Convolutional Neural Networks* (CNNs). Of course there has been quite extensive research on other essential layers like the Long Short-Term Memory (LSTM) layer or the 'brand new' Attention Layer that is rocketing Transformer Neural Networks, but the former two FCNNs/CNNs are the basis of deep learning architectures, and almost all other deep learning neural networks stem from these [31]. Furthermore, Multiply-Add is the most frequent operation in modern neural networks, acting as a building block for fully-connected and convolutional layers, both of which can be viewed as a collection of vector dot-products. For these reasons, to evaluate the performance and precision of a *Stochastic Computing Convolutional Neural Network* (SC CNN) it is needed the implementation of a SC 2D convolution and a SC Fully Connected layer functions, much like the ones already available in ML frameworks like Tensorflow or PyTorch, nonetheless, multiply-add are substituted with SC multiplication (being logic AND for SC numbers in Unipolar format and logic XNOR for Bipolar format) and SC addition which is basically the mere accumulation of the positive bits or ones in the SC bitstream.
Specifically referring to PyTorch, this provides the developer with the opportunity to create C++ and CUDA extensions so that customized functions in these languages can work hand in hand with the already existing features and Tensor objects in Python.
Firstly, the implementation of the CPU version of these SC operations is presented in C++ and secondly, to accelerate the overall execution time of these serial operations with multi-threaded parallelism, a CUDA version of the same is proposed and tested. Lastly, to assert the proper functionality of the SC operations, they are inserted in a classic PyTorch CNN and tested with respect to inference performance and precision.

4.1 CPU Implementation

In a first instance, what is primarily needed is to convert PyTorch tensors into a SC version of them so that they can be processed in the same way a normal Torch 2D convolution ($torch.nn.Conv2d$) [16] and a normal Torch Fully connected layer ($torch.nn.Linear$) [20] processes normal PyTorch tensors. To achieve this, a Stochastic Tensor Generator function is required along with the *Random Number Generator* (RNG) necessary to generate the stochastic numbers of the stochastic tensors.
To store all the functions and properties relevant to the CPU version of this work, a file called "ScTorch.cpp" is created along with its header file "ScTorch.h", this name is very important as this file is used to create the PyTorch extension, explained in later sections. In ScTorch.cpp there is a C++ class called $StochasticTensor$ that is in charge of taking a normal PyTorch tensor data and convert it to a SC version with stochastic bitstreams, this means that a dimension expansion is needed so the representation of a 2D tensor with *Floating Point* (FP) numbers becomes consequently a 3D tensor whose innermost dimension is the SC bitstream equivalent of the original FP tensor. E.g. For a Torch tensor converted to Unipolar Stochastic using a bitstream length of 5: $\{\{0.4, 0.2, 0.8\}\} \rightarrow \{\{\{1,0,1,0,0\},\{0,1,0,0,0\},\{1,0,1,1,1\}\}\}$.

4.1.1 Stochastic Tensor Generator

In Listing 4.2 a constructor for the StochasticTensor class defines as first parameter a $vector < vector < double >>$ & inputVector that is basically the data coming from a normal PyTorch tensor to be converted to SC, then comes the bitstream length, then

the type of RNG to be used in the generation of SC bitstreams and finally the format of the SC numbers (either Unipolar or Bipolar), in lines 2 to 5.

```
1  // Constructor
2  StochasticTensor::StochasticTensor(const std::vector<std::vector<double>>& inputVector,
3      const int bitstreamLength,
4      RandomNumberGenType type,
5      BitstreamRepresentation mode)
6      : tensor() { generateTensor(inputVector, bitstreamLength, type, mode); }
7  void StochasticTensor::generateTensor(const std::vector<std::vector<double>>& inputVector,
8      const int bitstreamLength,
9      RandomNumberGenType type,
10     BitstreamRepresentation mode) {
11     std::vector<std::vector<std::vector<uint8_t>>> SCtensor(inputVector.size(),
12         std::vector<std::vector<uint8_t>>(inputVector[0].size(),
13         std::vector<uint8_t>(bitstreamLength)));
14         for (size_t i = 0; i < inputVector.size(); ++i) {
15             for (size_t j = 0; j < inputVector[i].size(); ++j) {
16             stochasticNumberGenerator(bitstreamLength, type,  inputVector[i][j],
17             mode, SCtensor[i][j]); }
18     } tensor = SCtensor;
19 }
```

Listing 4.1: Stochastic Tensor Class part 1

Then these values are passed to a *stochasticNumberGenerator*() function in lines [24,52] with the help of an intermediary function *generateTensor* in lines [8,21]. The *stochasticNumberGenerator*() function determines if the input number is to become a Unipolar probability or if Bipolar, it is then normalized from [-1, 1] to [0,1] in line 30 and after normalization, either a Mersenne Twister RNG in lines [35,40] or a 16-bit LFSR in [43,50] creates the SC bitstreams with a pseudorandom generated seed [32]. Finally the SC bitstreams are returned back to *generateTensor* function as $vector < uint8\_t >$. These vectors are allocated in a 3D SC tensor of type $vector < vector < vector < uint8\_t >>>$ (line 12) which is stored in a class variable *tensor* (line 20) to be used later as inputs of the SC conv2d and SC FC functions.

```
24 void stochasticNumberGenerator(const int bitstreamLength, RandomNumberGenType type,
25 double inputRealNumber,
26 BitstreamRepresentation mode,
27 std::vector<uint8_t>& output) {
28     double probability = 0.0;
29     if (mode == UNIPOLAR) { probability = inputRealNumber; }
30     else if (mode == BIPOLAR) { probability = (inputRealNumber + 1) / 2.0; };
31     output.resize(bitstreamLength);
32     if (type == MT19937) {
33         std::random_device rd;  // Non-deterministic random number generator
34         std::mt19937 gen(rd()); // Mersenne Twister engine seeded with random_device
35         std::uniform_real_distribution<double> dis(0.0, 1.0);
36         for (int i = 0; i < bitstreamLength; ++i) {
37             double randomValue = dis(gen);
38             output[i] = (randomValue < probability) ? 1 : 0;
39         } } else if (type == LFSR) {
40         std::random_device rd;
41         std::mt19937 gen(rd());
42         std::uniform_int_distribution<uint16_t> dis(0, 65535);
43         uint16_t lfsrSeed = dis(gen);
44         for (int i = 0; i < bitstreamLength; ++i) {
45             double randomNumber = static_cast<double>(std::bitset<16>(lfsrSeed).to_ulong())/65535.0;
46             output[i] = (randomNumber < probability) ? 1 : 0;
47             lfsrSeed = LFSR_StatesGenerator(lfsrSeed); } } }
```

Listing 4.2: Stochastic Tensor Class part 2

The type of RNG and the SC number format are defined as enumerators in the ScTorch.h file like $'enum\ BitstreamRepresentation\ \{UNIPOLAR,\ BIPOLAR\};'$ and $'enum\ RandomNumberGenType\ \{LFSR,\ MT19937\};'$. To create one single bit of the stochastic bitstream the operation is the same for both RNGs: $output[i] = (randomValue < probability)\ ?\ 1 : 0;$. Lastly, the structure of the 16-bit LFSR can be observed in Appendix Listing 8.1.
An example of the output of a converted FP tensor to stochastic tensor with bit-stream length of 10 is presented in Figure 4.1. It is necessary to remind that with a higher bit-stream length, the accuracy of the SC numbers improves considerably depending on the amount of decimals of the original FP tensor.

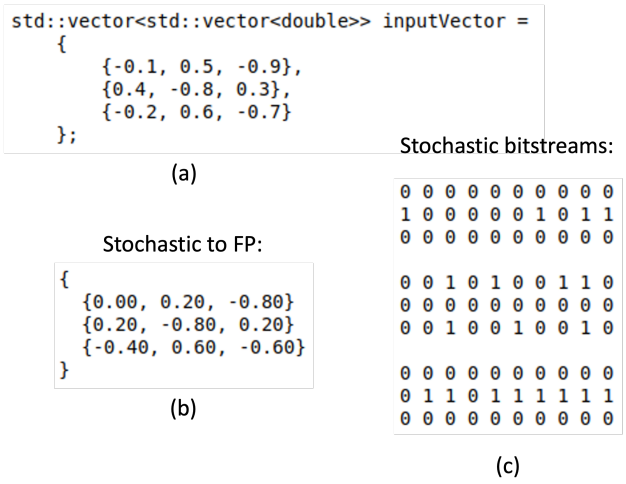


Figure 4.1: (a) Original FP tensor. (b) Stochastic tensor converted to FP. (c) Stochastic tensor.

4.1.2 SC 2D Convolution

The stochastic bitstreams tensor as illustrated in Figure 4.1(c) is precisely the input of the SC functions developed in this work. To perform a 1D, 2D or 3D convolution there are two fundamental components: the input data to be convolved and the hyperparameters that give shape to the convolution. The input data for a 2D convolution consists of an input tensor representing an image (with 1 channel in case of gray-scale images or 3 channels in case of RGB colored images, for example); and a tensor of weights also referred previously as convolutional kernel. This tensor of weights is the one trained by backpropagation in CNNs training and in the context of SC CNNs, this tensor's values as well as the values of the input image tensor must be in the [-1, 1] range so they can be represented as Bipolar SC numbers.
Listing 4.3 presents the CPU C++ version of a SC 2D convolution function that takes as input parameters two stochastic tensors (e.g. image data and weights) and the value of the 3 hyperparatemers Padding, Stride and Dilation of a normal convolution as mentioned initially in subsubsection 3.2.1. The first lines [5,9] generate a stochastic tensor from the image tensor (hereafter referred simply as *inputs*) and from the weights tensor (defined as *kernel* in code) and calculate their sizes. Then, with the sizes of the inputs and kernel and the given hyperparameters it is possible to calculate the size of the output convolution in lines [23,24]. The core of the convolution function lies within lines [33,60]. Since the algorithm is serial, there are 5 nested for loops so the time complexity basically is $O(n^5)$. First, it is needed to iterate over the height and width of the output tensor and then to iterate over the height and width of the kernel.

Inside the penultimate For loop in line 40 is where the SC multiplication and addition of bitstreams takes place. Lines [42,43] calculate the position or index of the value in the inputs tensor that is going to be multiplied by the value of the weights tensor and then accumulated with the other products.

```
1  std::vector<std::vector<double>> ScConv2d(
2      const StochasticTensor input, const StochasticTensor kernel,
3      int padding, int stride, int dilation) {
4
5      StochasticTensor SCtensorInput = input;
6      StochasticTensor SCtensorKernel = kernel;
7
8      StochasticTensor::SizeTuple inputSizes = SCtensorInput.getSize();
9      StochasticTensor::SizeTuple kernelSizes = SCtensorKernel.getSize();
10
11   // Validate input and kernel dimensions
12   int in_height = std::get<0>(inputSizes);
13   int in_width = std::get<1>(inputSizes);
14
15   int kernel_height = std::get<0>(kernelSizes);
16   int kernel_width = std::get<1>(kernelSizes);
17
18   if (kernel_height % dilation != 0 || kernel_width % dilation != 0) {
19     throw std::invalid_argument("Incompatible dilation and kernel dimensions");
20   }
21
22   // Calculate output dimensions with padding
23   int out_height = (in_height - kernel_height + 2 * padding) / stride + 1;
24   int out_width = (in_width - kernel_width + 2 * padding) / stride + 1;
25
26   const double inputBitstreamSize = std::get<2>(inputSizes);
27
28   // Initialize output with zeros
29   std::vector<std::vector<double>>
30     stochasticOutputWthAcc(out_height, std::vector<double>(out_width, 0));
31
32   // Perform convolution
33   for (int oh = 0; oh < out_height; ++oh) {
34     for (int ow = 0; ow < out_width; ++ow) {
35         std::vector<uint8_t> addedScOutput_acc(1, 0);
36         int count = 0;
37         int accumulatedOnes = 0;
38         //int numberOfAccumulations = 0;
39       for (int kh = 0; kh < kernel_height; kh += dilation) {
40         for (int kw = 0; kw < kernel_width; kw += dilation) {
41           // Handle padding with boundary checks
42           int in_h = oh * stride - padding + kh;
43           int in_w = ow * stride - padding + kw;
44
45           if (in_h >= 0 && in_h < in_height &&
46               in_w >= 0 && in_w < in_width) {
47             count = 0;
48             for (size_t bit_counter = 0; bit_counter < inputBitstreamSize; ++bit_counter) {
49                 if((!(input.getVectorAt(in_h, in_w)[bit_counter] ^
50                     kernel.getVectorAt(kh,kw)[bit_counter]))==1)
51                     {count++;}
52             }
53           accumulatedOnes += count;
54         }
55       }
56     }
57     stochasticOutputWthAcc[oh][ow] = static_cast<double>(accumulatedOnes);
58     }
59   }
60   return stochasticOutputWthAcc;
61 }
```

Listing 4.3: SC 2D Convolution

Stochastic bitstreams multiplication happens in lines [49,50] and let's remember this multiplication is done bit by bit with an AND logic gate in case of Unipolars or, in this case, an XNOR gate for Bipolars, just like introduced in subsubsection 3.1.2. If the product of the multiplied bits is equal to one, then the accumulator is increased by 1 (lines 51-53) and, after all the iterations over the kernel's width and height are finished, the final accumulation of positive bits or ones is assigned to a position in the output tensor. The output tensor is initially defined as full zeros (lines 29-30) and, in order to convert back or *denormalize* its final values to either Unipolar or Bipolar FP probabilities, the accumulations have to be divided by the number of iterations over the kernel dimensions (width + height), but this division can be realized outside the function. The output of SC 2D convolution is a tensor of FP values that are meant to be as approximated as possible to the result of a normal 2D convolution as introduced previously in subsubsection 3.2.1 and an example with random inputs, weights and a bitstream length of 1024 can be observed in Figure 4.2. The cosine similarity compared to the output tensor of the original Torch $conv2d()$ function is 98.7%.

```
Normal 2D conv:
tensor([[[[-0.0500,  0.0000,  0.0500, -0.3000],
          [-0.2000, -0.5500, -0.4000,  0.9000],
          [-0.3500,  0.1000,  0.8500, -1.5000],
          [ 0.0000,  0.7000, -0.8000,  0.9000]]]])
```

(a)

```
Stochastic 2D conv:
tensor([[-0.0733, -0.0067,  0.1133, -0.2467],
        [-0.2400, -0.6667, -0.4733,  0.9533],
        [-0.3200,  0.2533,  0.9200, -1.6000],
        [-0.0333,  0.7133, -0.7733,  0.8933]])
```

(b)

Figure 4.2: (a) Normal 2D conv output. (b) SC 2D conv output.

4.1.3 SC FC Layer

The function that performs a SC fully connected layer for CPU is called $forward()$ and it resides in a C++ class named $ScFcLayer$, and just as the $StochasticTensor$ class, they both are defined in the $ScTorch.h$ file and developed in $ScTorch.cpp$. There are 2 constructors that take the input data into the $ScFcLayer$ class and pass it to the $forward()$ function.

```
1 // Constructor with specified weights and bias
2 ScFcLayer::ScFcLayer(const std::vector<std::vector<double>>& weights,
3   const std::vector<double>& bias, const int bitstreamLength,
4   RandomNumberGenType type, BitstreamRepresentation mode)
5     : weights(weights), bias(bias), input_size(weights.size()), output_size(bias.size()),
6       scWeight(StochasticTensor(weights, bitstreamLength, type, mode)),
7       scBias(StochasticTensor({bias}, bitstreamLength, type, mode)),
8       bitstreamLength(bitstreamLength), type(type), mode(mode) {}
```

Listing 4.4: Sc Fc Constructors

The first constructor (in lines [2,8] of Listing 4.4) simply takes an existing 2D tensor (matrix) of FP weights and an existing vector of FP biases (both previously normalized to [-1,1]) and converts them to stochastic tensors with the information provided in the parameters of the constructor which are: bitstream length, RNG type (also either Mersenne Twister or 16-bit LFSR) and bitstream representation mode (either Unipolar or Bipolar). The second constructor (in lines [2,20] of Listing 8.2 in section 8) allows the creation of random stochastic tensors of weights and biases from the required number of input and output neurons. The type of RNG used for this is again the Mersenne Twister (mt19937) RNG as it is already embedded in the C++ standard library ($< stdlib.h >$).

```
1  // Forward pass without activation function
2  std::vector<double> ScFcLayer::forward(const std::vector<double>& inputs) {
3      if (inputs.size() != input_size) {
4          throw std::invalid_argument("Input size does not match the layer's input size.");
5      }
6      StochasticTensor scInput = StochasticTensor({inputs}, bitstreamLength, type, mode);
7      std::vector<double> outputs(output_size, 0.0);
8
9      for (int j = 0; j < output_size; ++j) {
10         std::vector<uint8_t> accumulatedMultiplication(1, 0);
11         std::vector<uint8_t> accumulatedBias(1, 0);
12         for (int i = 0; i < input_size; ++i) {
13           std::vector<uint8_t> scMultiplication = bitstreamOperation(scInput.getVectorAt(0,i),
14                 scWeight.getVectorAt(i,j), XNOR);
15           accumulatedMultiplication =
16                 concatenateSCVectors(scMultiplication,accumulatedMultiplication);
17         }
18         accumulatedBias = concatenateSCVectors(scBias.getVectorAt(0,j),accumulatedMultiplication);
19         double scale = double(accumulatedBias.size())/double(bitstreamLength);
20         outputs[j] = calculatePx(accumulatedBias, mode)*scale;
21     }
22     return outputs;
23 }
24
25 // Sigmoid activation function
26 std::vector<double> ScFcLayer::sigmoid(const std::vector<double>& inputs) {
27     std::vector<double> outputs = forward(inputs);
28     for (double& output : outputs) {
29         output = 1 / (1 + std::exp(-output));
30     }
31     return outputs;
32 }
```

Listing 4.5: Sc Fc Forward function

Listing 4.5 above shows exactly the CPU version of the SC fully connected layer $forward()$ function, although with a different approach as the one implemented for SC 2D convolution. $forward()$ takes as input parameter a single vector of FP values (the input neurons) which converts to stochastic bitstreams tensor (in line 6) and then multiplies by the stochastic tensor of weights (lines [13,14]) however, this time the whole resultant bitstream is maintained, that is, all the ones and zeros of the multiplication product remain stored in an uni-dimensional vector object $vector < uint8\_t >$ and are consequently accumulated or *concatenated* with the following products, resulting in a very long sequence of bits called $'accumulatedMultiplication'$ (in line 15). Afterwards, the vector representing stochastic bitstreams of the biases is concatenated as well to $'accumulatedMultiplication'$ and in order to convert back from bitstream to FP probability, the function $calculatePx()$ counts the total amount of positive bits and divides it by the length of the long vector, just like in the circuit presented in Figure 3.1(b). Important to note that the output values (neurons) must be denormalized and the way to do this is by multiplying the probabilities by a scalar (line 20) which, in this code, is calculated by dividing the long vector of concatenations by the initial bitstream length (line 19). Finally, the output neurons can easily be passed through an activation function like Sigmoid, presented as a function of the $ScFcLayer$ class in lines [26,32] and an output example next in Figure 4.3 with a selected bitstream length of 300 and the cosine similarity used to compare with the output tensor of the original torch.nn.Linear() function is of 98.7%. Bitstreams concatenation and bitstream-to-FP-converter functions can be observed in Listing 8.3 of section 8.

```
Outputs without activation:
tensor([[-2.5400, -1.6600, -0.8800, -0.2000,  0.3800,  0.8600]],
       grad_fn=<AddmmBackward0>)

Outputs with Sigmoid activation:
tensor([[0.0731, 0.1598, 0.2932, 0.4502, 0.5939, 0.7027]],
       grad_fn=<SigmoidBackward0>)
```

(a)

```
Stochastic fully connected layer:
tensor([-2.6467, -1.8867, -1.0600, -0.3867,  0.4800,  0.8533])

Stochastic fully connected layer (sigmoid act. func):
tensor([0.0471, 0.1347, 0.2303, 0.4045, 0.5858, 0.6900])
```

(b)

Figure 4.3: (a) Normal torch.nn.Linear() output (before & after Sigmoid).
(b) SC FC Layer output (before & after Sigmoid).

It is evident that $forward()$ computes a SC FC layer with many extra steps that were not needed in SC 2D Conv, but this approach deems potentially useful to consider if instead of FP probabilities, the output was still required in the form of stochastic bitstreams in case more stochastic arithmetic operations were to follow or even special stochastic operations like the Stochastic ReLU function proposed in [9].
In any case, performance optimization for this function seems necessary, and although this can be done with a simple accumulator just like the CPU version of SC 2D Conv has, there is a hardware approach yet to be presented. With multi-threaded data-parallelism, these stochastic operations can be accelerated hundreds of times with the same and more amounts of data throughput.

4.2 GPU Acceleration

As it has been mentioned earlier in subsection 3.3, the CPU version of SC 2D Conv and SC FC layer are serial with many nested loops and therefore rather slow in terms of time complexity. With the addition of SC arithmetic multiplication-accumulation there is even one more innermost iteration considered in the amount of loops within the CPU algorithms. The following exposes in $O(n)$ notation the time complexity expected from the serial SC operations, SC 2D Conv first and SC FC layer after:

$$O(C_{in} \times C_{out} \times k_h \times k_w \times H_{out} \times W_{out} \times N) \tag{5}$$

where: C corresponds to input and output channels, k is width and height of kernel, H and W are height and width of the output feature map and N is the bitstream length of the stochastic numbers.

$$O(n_{in} \times ((n_{out} \times N_{weights}) + N_{biases})) \tag{6}$$

where: n is number of input and output neurons and N is the bitstream length of the stochastic bitstreams for weights or biases.
Let's also remember the serial CPU version of a stochastic tensor generator takes a bidimensional matrix of FP decimals and **expands** it into a tensor of stochastic bitstreams. To realize this expansion, 3 nested $Forloops$ are required, so in addition to the overall execution time of the CPU serial SC functions exposed in equations 5 and 6, the time to convert a FP matrix to a bitstreams tensor given by 7 also represents a considerable part of processing overhead.

$$O(H_{in} \times W_{in} \times N_{in}) \tag{7}$$

where: H is height of input FP matrix, W is width of input FP matrix and N is the chosen bitstream length for the stochastic bitstreams.
Thus, the main objective of implementing parallelism here is to accelerate these functions by reducing their time complexity, in a first instance, and also by using some GPU hardware capabilities and some CUDA features oriented to performance optimization.

4.2.1 CUDA Stochastic Tensor Generator

Reminding the core of CUDA programming is the CUDA kernel, Listing 4.6 presents the kernel of a parallelized *Stochastic Tensor Generator* (STG) for stochastic bipolar numbers.
When working with CUDA kernels, it might be required to pre-process and post-process the data before and after sending it to a kernel. In this case, all the input data is flattened in uni-dimensional vectors of FP decimals before being transferred to a GPU device, that is, the inputs, weights and biases of a 2D convolution or a FC layer are all flattened from matrices with FP values ($vector < vector < double >> inputData$) to vectors ($vector < float > flattened\_input\_data = flatten2D(inputData)$), this eases considerably parallelization of data inside the kernel by means of Thread-level Parallelism, introduced in subsubsection 3.3.6.

```
1 __global__ void stochasticTensorGenerator(const float* inputData, const float* randomMatrix,
2         int8_t* output, int inputData_size, int RM_cols) {
3     int idx = blockIdx.x * blockDim.x + threadIdx.x;
4     if (idx < inputData_size * RM_cols) {
5         int n_idx = idx / RM_cols;
6         int rm_col_idx = idx % RM_cols;
7         int rm_idx = n_idx * RM_cols + rm_col_idx;
8         output[idx] = (randomMatrix[rm_idx] < ((inputData[n_idx] + 1) / 2.0)) ? 1 : 0; }
9 }
```

Listing 4.6: Stochastic Tensor Generator CUDA kernel

The input parameters of the STG kernel are $float\star\ inputData$: which is the flattened array of values to be converted to stochastic bitstreams, $float\star\ randomMatrix$: which is a flattened array of the random FP values in [0,1] range that are compared to inputData to determine the ones or zeros in the bitstream; $int8\_t\star\ output$: the large uni-dimensional array with every bitstream; $int\ inputData\_size$: the number of values in inputData array and $int\ RM\_cols$: basically the bitstream length. Then, the STG launches one thread per bit to allocate a 1 or 0 into the 1D flattened *output* array. This output array has a number of stochastic bits equal to the total number of elements in the $inputData$ array $\times\ RM\_cols$. Therefore if the input data has 2 FP numbers and the bitstream length is 3, the total amount of launched threads is 6. This assures that the generation of every stochastic tensor is efficiently parallelized and immediate, that is, without needing the implementation of *For* loops inside the STG CUDA kernel, taking full advantage of TLP for this specific functionality.
The input and output arrays as well as the constants: $inputData\_size$ and RM_cols are all loaded in global memory, then each thread identifier is calculated for each block (line 3) and the if conditional in line 4 limits the creation of more thread identifiers to those actually required. Variables n_idx, rm_col_idx and rm_idx are dynamically allocated in registers for these are the *fastest* (with less latencies) memory spaces within the GPU and Table 4.1 explains the index arithmetic behind each instruction implemented in lines 5, 6 and 7 that determine which data values of $inputData$ or $randomMatrix$ arrays are next for each thread to 'take' from global memory.

Table 4.1: STG Index arithmetic

int n_idx = idx / RM_cols:	**int rm_col_idx = idx % RM_cols:**	**int rm_idx = n_idx x RM_cols + rm_col_idx:**
Calculates the index of the element in the inputData array (previously flattened in cpu). Used to access the elements of inputData.	Calculates the column index in the randomMatrix row that corresponds to the current thread index idx, in other words, *rm_col_idx* cycles through the column indices for each row of randomMatrix array (previously flattened in cpu).	To access each element in the 1D flattened randomMatrix array, its 2D indices (row, col) must be converted to a 1D index, this is done by multiplying the row index (*n_idx*) by the number of columns (*RM_cols*) and adding the column index (*rm_col_idx*), in other words, *rm_idx* calculates the index of randomMatrix corresponding to the current thread index 'idx'.

Finally, every value of $inputData[n\_idx]$ is normalized to [0,1] and compared to a random number to form the large chain of stochastic bitstreams to be stored in $output[idx]$ array (line 8).

To illustrate better how TLP is implemented in the STG kernel, the following example presents a rather simple case where a FP matrix with only two values: **input_data[][] = [[-.2],[.6]]** is converted to a flattened stochastic tensor, considering a bitstream length of 3 bits per digit, therefore the total number of launched threads will be 6 as per $input\_data.size() \times bitstream\_length$:

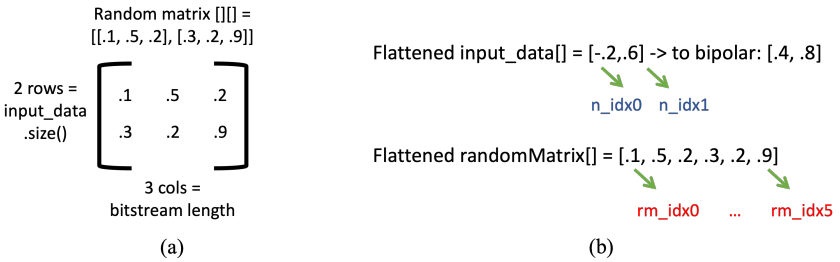


Figure 4.4: (a) Matrix with random FP numbers [0,1].
(b) Flattened inputs converted to bipolar format & flattened random matrix.

After normalizing, flattening and transferring the inputs and random numbers arrays to GPU device, the calculated indexes n_idx and rm_idx access their values to generate either a one or a zero in the stochastic bitstream in a parallel way, as depicted on Figure 4.5.

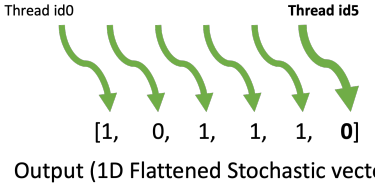


Figure 4.5: Parallel multi-threaded generation of a stochastic bitstream.

To understand better this index arithmetic behind the calculation of the access indexes, Table 4.2 shows the values they take per each of the threads; highlighted are the operations that thread no. 6 (with thread index 5) would need to realize to generate a zero in position 6 of the bitstream for this specific example.

int n_idx = idx / RM_cols:

idx	n_idx
0	0/3 = 0
1	1/3 = 0
2	2/3 = 0
3	3/3 = 1
4	4/3 = 1
5	5/3 = 1

(a)

int rm_col_idx = idx % RM_cols:

idx	rm_col_idx
0	0%3 = 0
1	1%3 = 1
2	2%3 = 2
3	3%3 = 0
4	4%3 = 1
5	5%3 = 2

(b)

int rm_idx = n_idx * RM_cols + rm_col_idx:

idx	n_idx	rm_col_idx	rm_idx
0	0	0	0*3+0 = 0
1	0	1	0*3+1 = 1
2	0	2	0*3+2 = 2
3	1	0	1*3+0 = 3
4	1	1	1*3+1 = 4
5	1	2	1*3+2 = 5

(c)

Table 4.2: (a) Calculation of n_idx index. (b) Calculation of rm_col_idx index. (c) Calculation of rm_idx index.

Finally, the output array is **not** copied back to CPU host, but it remains stored in device's global memory, for these stochastic arrays are the inputs of the SC Conv2d and SC FCLayer CUDA kernels. Since one STG kernel is used to convert only a single array of inputs, to generate stochastic tensors for the weights and biases too it is required to use the same kernel two or three more times; then to accelerate the execution of these many repeated kernels, CUDA streams allow their concurrent computation just as introduced in subsubsection 3.3.7 and shown in Figure 3.19.
The way CUDA streams are implemented to accelerate the use of more than one STG kernel is commented in later sections, particularly in the software techniques required to extend the CPU C++ and the GPU CUDA SC functions into Python with a "Modules Binder" for CPU and a "Kernels Wrapper" for CUDA functions. For now, the full focus will fall on the explanation of the CUDA kernels.

4.2.2 SC CUDA Conv2d

To try to 'fully' parallelize a single algorithm is not a simple task, in fact Amdahl's law states "the maximum potential improvement to the performance of a system is limited by the portion of the system that cannot be parallelized" so in the case of the SC functions here presented, there exists also a portion of the algorithms that are deemed to not be parallelizable. Nonetheless, the parallelization techniques here implemented in the CUDA SC functions still represent quite a considerable improvement with respect to the overall 'speed' of their CPU counterparts, therefore the effort behind the following improvements is considered to be enough to achieve a proper and efficient acceleration of a 2D convolution and fully connected layer using stochastic computing.
There are already many implementations on the web related to parallelized versions of a 2D conv and FC layer with global [33], shared [34] and even constant memories [35] of a GPU, but in order to properly achieve an efficient implementation of these using SC, a graphic abstraction of how many threads should be created and their execution paths is extremely helpful. The following is a schematic Figure 4.6 of the path that a single thread takes during a parallel execution of a 2D convolution, considering that the total number of threads invoked within a SC 2D Conv kernel is equal to the size of the output layer, that is, the convolution output's width x its height.

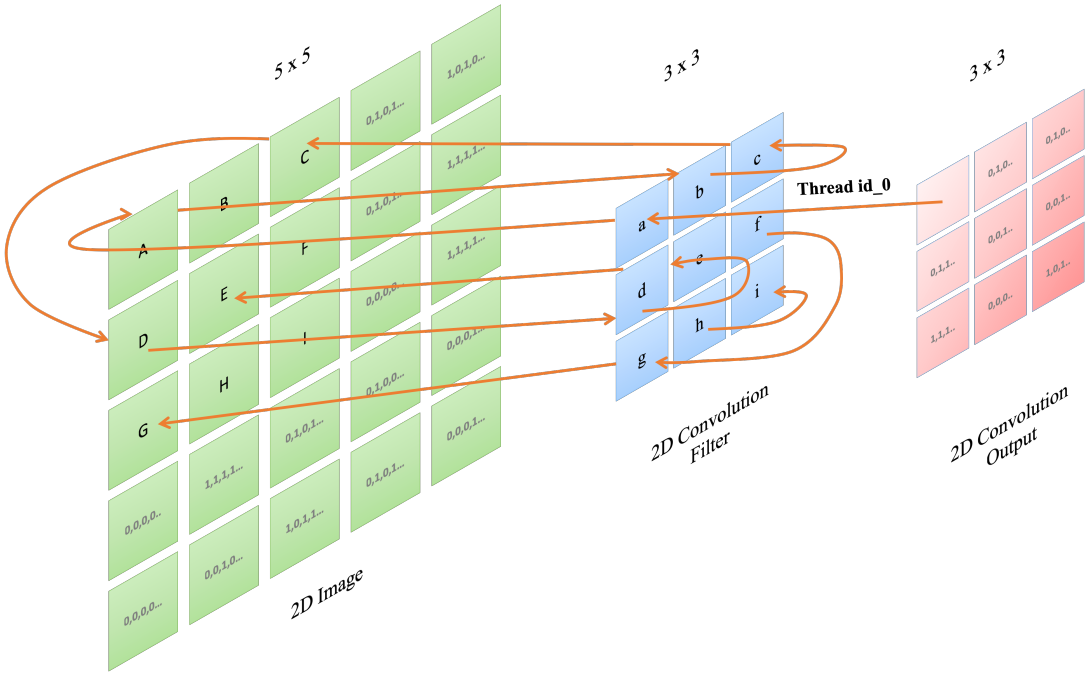


Figure 4.6: Path of a single thread ID_0 in the SC CUDA 2D Convolution.

In the previous schematic it is intended to show that there would be a total of 9 threads invoked for that specific example, as the output layer has 9 'pixels'. Then the very first thread takes from global memory the stochastic bitstreams of the flattened convolutional kernel starting at location 'a' and ANDs or XNORs them with the bitstream of the input at location 'A'; then the same happens with 'b'-'B', 'c'-'C' and so on until the final location at 'i'-'I' is reached and the thread can copy back the final accumulation of ones to the output array also in global memory.

It is reckoned then, that a good practical approach is this 'multi-threading the output size' technique given that every invoked thread within the kernel has an assigned workload for the entire time of the computation and there is not a single thread that remains idle, remembering that idle threads may impact directly on the number of active Warps per block, the *Occupancy* and therefore on the overall device computation performance. The kernel of the SC 2D convolution can be seen in Listing 4.7.

```
__global__ void conv2D(const int8_t* input, const int8_t* kernel, float* output, int inputHeight,
    int inputWidth, int kernelHeight, int kernelWidth,
    int outputHeight, int outputWidth, int N) {

    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < outputHeight && j < outputWidth) {
        float accumulatedOnes = 0;
        for (size_t m = 0; m < kernelHeight; ++m) {
            for (size_t n = 0; n < kernelWidth; ++n) {
                for (size_t bit_counter = 0; bit_counter < N; ++bit_counter) {
                    if((!(input[((i + m) * inputWidth * N + (j + n) * N)+bit_counter] ^
                    kernel[(m * kernelWidth * N + n * N)+bit_counter])) == 1)
                    {accumulatedOnes++;} } } }
        output[i * outputWidth + j] = accumulatedOnes;
    }
}
```

Listing 4.7: SC 2D Convolution CUDA kernel

The input parameters int8_t* input and int8_t* kernel correspond precisely to the (pointers pointing to the) stochastic bitreams locations of the input layers and the convolutional kernels (also called weights). They are type int8_t because the byte is the smallest data type available in CUDA (and C++), there is not a possibility to define data objects occupying only one single bit of memory. The rest of the integer variables are straight-forward but N, which is basically the bitstream length.
Blocks of 2 dimensions are defined to make thread indexing easier (in lines [5,6]), since the input stochastic tensors basically represent 2 dimensional matrices but converted to SC, reminding subsubsection 4.2.1. Then, due to parallelization it is only needed to iterate through the weights' height and width to multiply-accumulate, unlike the CPU version that has to iterate through the height and width of the output layer as well. Commonly a convolutional kernel is of width/height: 3x3, 5x5 or 7x7, so the usual amount of iterations in this parallel SC 2D convolution will be limited by the kernel's size (e.g. 49) $\times$ the bitstream length (due to the innermost *For* loop that moves through all the stochastic bits of every SC number.
This SC 2D CUDA Conv as well as the next SC CUDA FCLayer functions are based on computation of bipolar SC numbers, that is why multiplication is done by XNOR-ing every bit of the inputs and the weights bitstreams (in line 12); finally in lines 14-15 the number of positive bits resulting from the multiplication are accumulated and stored in an output array, to be unflattened after being copied back to CPU host. The output values have to be normalized to bipolar probabilities, but in order to reduce the amount of arithmetic operations inside the CUDA kernel (hence avoiding adding more computation latencies), this normalization is realized in CPU using the high level capabilities of the PyTorch framework. The output tensors from this function are quite similar if not the same in terms of precision compared to its CPU counterpart exposed in Figure 4.2, that is why it is considered irrelevant to show an example comparison between the output tensor of a normal torch.nn.conv2D() and the output of this SC Conv2D().

4.2.3 SC CUDA FCLayer

The CUDA kernel of the Stochastic Computing Fully Connected Layer function is called forward_pass() and is presented in Listing 4.8. It has a similar structure to the one implemented in SC Conv2D() but this time only 1 block dimension is defined to ease thread indexing (line 5).

```
1  __global__ void forward_pass(
2      int8_t* input, int8_t* weights, int8_t* biases, float* output,
3      int input_size, int output_size, int bit_length) {
4
5      int j = blockIdx.x * blockDim.x + threadIdx.x;
6
7      if (j < output_size) {
8          float accumulatedOnes = 0;
9          for (size_t i = 0; i < input_size; ++i) {
10             for (size_t k = 0; k < bit_length; ++k) {
11                 if((!(input[(i * bit_length)+k] ^
12                 weights[((i * output_size + j) * bit_length)+k])) == 1)
13                     {accumulatedOnes++;} } }
14
15         for (size_t biasIndex = 0; biasIndex < bit_length; ++biasIndex) {
16             accumulatedOnes += biases[(j * bit_length)+biasIndex]; }
17         output[j] = accumulatedOnes;
18     }
19 }
```

Listing 4.8: SC FCLayer CUDA kernel

Just like SC Conv2D(), the parallelization technique implemented for the SC FCLayer is 'multi-threading the output size', so the total number of invoked threads within the CUDA kernel is equal to the number of output neurons.
The input bitstreams are all arrays of int8_t ones and zeros and the output vector in line 19 returns the accumulated ones after bipolar XNOR multiplication of each of the inputs × its corresponding weight value (lines [12,13]) in addition to the bias (lines 15-16). The advantage that is achieved with this parallelization technique is that it is only needed to iterate over / loop through the input size × the bitstream length in order to multiply-accumulate.
Figure 4.7 shows precisely an example of the path of the first thread with ID_0. Thread id_0 takes the bitstream from the inputs and weights arrays, multiplies them, accumulates the positive bits in registers (line 13), does the same from A to E and finally adds the positive bits of the bias. If there are 3 output neurons, the number of invoked threads is 3.

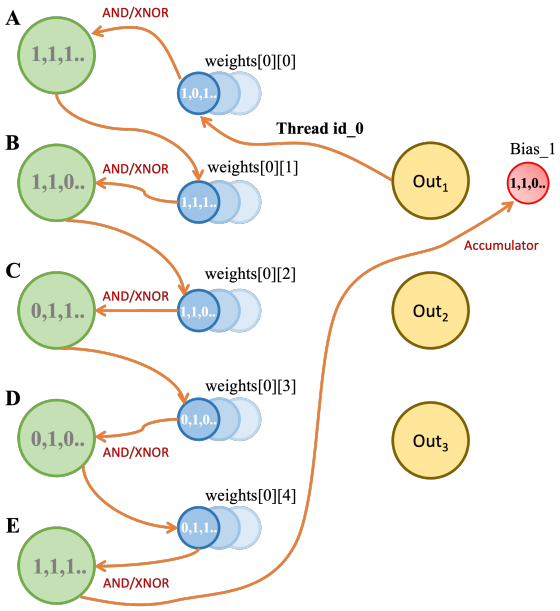


Figure 4.7: Path of a single thread ID_0 in the SC CUDA FCLayer kernel.

The previous techniques that parallelize the number of outputs clearly have the disadvantage that if a 2D Conv or FC layer have too many inputs and too few outputs, the GPU acceleration will not be substantial or representative enough compared to its CPU versions. It is possible to 'multi-thread the input size' by invoking the same number of threads as the number of input pixels in a 2D conv or input neurons in a fc layer in case they have many more inputs than outputs, nonetheless, this approach brings with it other challenges and possible caveats related to overall kernel 'speed' or device performance: firstly, the multiply-accumulate output of each thread will not be the final result, just an intermediate value that would need to be stored in a temporal space such as *shared memory* subsubsection 3.3.3, which is accessible by every thread within a block, and then once every intermediate value is allocated in an array in this temporal memory, a reduction of this intermediate array would need to be done to get the number of outputs. It is possible that a combination of these two approaches deems more efficient than implementing only one of them, however, for the purposes of this work, 'multi-threading the output size' is just enough to achieve proper results in a SC CNN with the same architecture as of the classic Le-Net 5 Figure 3.4.

4.3 PyTorch Extension

To be able to use all these CPU or CUDA SC functions in python with PyTorch tensors an extension is required. In case of the CPU ScTorch.h library, to create the extension it is necessary to call a **PYBIND11_MODULE(name, variable, ...)** macro inside the ScTorch.cpp script, just below the functions and classes. This macro basically exposes every function, class, enumerator or any required object within the C++ code into python. The documentation to use pybind11 can be found in [36]. Listing 4.9 shows precisely how this macro exposes the ScConv2d function along with the Stochastic Tensor generator class and the SC FcLayer class for the CPU SC library.

```
PYBIND11_MODULE(sc_torch_cpp, m) {  // Ensure the module name matches
    pybind11::enum_<RandomNumberGenType>(m, "RandomNumberGenType")
        .value("LFSR", RandomNumberGenType::LFSR)
        .value("MT19937", RandomNumberGenType::MT19937)
        .export_values();

    pybind11::enum_<BitwiseOperation>(m, "BitwiseOperation")
        .value("AND", BitwiseOperation::AND)
        .value("XNOR", BitwiseOperation::XNOR)
        .export_values();

    pybind11::enum_<BitstreamRepresentation>(m, "BitstreamRepresentation")
        .value("UNIPOLAR", BitstreamRepresentation::UNIPOLAR)
        .value("BIPOLAR", BitstreamRepresentation::BIPOLAR)
        .export_values();

    // Expose the StochasticTensor class
    pybind11::class_<StochasticTensor>(m, "StochasticTensor")
        .def(pybind11::init<const std::vector<std::vector<double>>&, int,
            RandomNumberGenType, BitstreamRepresentation>())

    // Expose the ScFcLayer class
    pybind11::class_<ScFcLayer>(m, "ScFcLayer")
        .def(pybind11::init<const std::vector<std::vector<double>>&, const std::vector<double>&,
            int,  RandomNumberGenType, BitstreamRepresentation>())
        .def(pybind11::init<int, int, int,  RandomNumberGenType, BitstreamRepresentation>())
        .def("forward", &ScFcLayer::forward)
        .def("sigmoid", &ScFcLayer::sigmoid)
        .def("relu", &ScFcLayer::relu)

    m.def("ScConv2d", &ScConv2d, "A stochastic 2D convolution function");
}
```

Listing 4.9: Python - C++ Modules Binder

The same macro can be used to call the CUDA SC functions into python, with an important difference: the CUDA kernels must be wrapped inside C++ functions that define important technicalities like # of threads per block and # of blocks per grid, static data allocation into memory spaces (global, shared, etc.), transference of data from host to device and vice-versa, synchronization of kernels or streams and so on. In turn, these CUDA/C++ functions are to be contained in yet another different C++ script with the declarations for the first functions and the pybind11 macro that works as the bridge between the python interpreter. To see how this 'bridge' script is defined it can be done in Appendix Listing 8.5.
The process to install a pure C++ or a mixed CUDA/C++ extension into PyTorch is explained in [37]. Basically there are to ways: by using setuptools which installs a setup.py file using a command-line shell like bash or by using the just in time (JIT) compiler.

The second way is easier to implement than the first one given that it is only needed to call a simple function in PyTorch's API called torch.utils.cpp_extension.load(); but it is also slower since it has to load everything again every time a .ipynb file is restarted or shutdown in an IDE like Jupyter Notebook. An example on how the setup.py of the first method looks like for the CPU functions is seen in Listing 4.10. Very important to notice the name of the extension: **sc_torch_cpp** (line 8) must match the one used in the macro (at the top of Listing 4.9).

```
1  from setuptools import setup
2  from torch.utils.cpp_extension import CppExtension, BuildExtension
3
4  setup(
5      name='sc_torch_cpp',
6      ext_modules=[
7          CppExtension(
8              name='sc_torch_cpp',
9              sources=['ScTorch.cpp'],
10             include_dirs=['.']  # This tells the compiler where to find ScTorch.h)],
11     cmdclass={'build_ext': BuildExtension}
12 )
```

Listing 4.10: How to install CPU library ScTorch.cpp with a setup.py file

It is also shown next in Listing 4.11 how to import the CUDA/C++ extension using the JIT compiler method.

```
1  import torch
2  import torch.nn as nn
3  import torch.nn.functional as F
4  from torch.utils.cpp_extension import load
5
6  ScCudaTorch = load(
7      name="sc_cuda_torch",
8      sources=["ScCudaTorch.cpp", "ScCudaTorch.cu"],
9      verbose=True
10 )
```

Listing 4.11: How to import ScCudaTorch.cpp / ScCudaTorch.cu scripts with JIT compiler.

A small example on how to call the CPU SC functions into python using the **sc_torch_cpp** keyword defined in Listing 4.10 after installing the extension is shown next in Listing 4.12, however, how to call the CUDA SC functions with the **ScCudaTorch** keyword in Listing 4.11 will be presented later with the full implementation of a SC CNN using 'Unipolar kernels'. Important to notice there are two file sources being called for the CUDA extension: 'ScCudaTorch.cpp' and 'ScCudaTorch.cu', the first script basically is the bridge that binds everything with python and the second one is where the CUDA kernels and their function wrappers are located. According to [37], when using the JIT compiler both names: 'ScCudaTorch.cpp' and 'ScCudaTorch.cu' can be the same but when using the first setup.py file they must be different.
In Listing 4.12 is the code that presents how to use the SC extension of PyTorch. First comes the call to the C++ properties, functions, enumerators and classes through the **sc_torch_cpp** keyword, those are: type of RNG to be used in the Stochastic Tensor Generator, stochastic number format (unipolar, bipolar) and bitstream length. Then the inputs, weights and in case of FC layer the biases too are all defined as either 1D or 2D arrays and inputted into ScFcLayer.forward() or ScConv2d(). The outputs of both functions are converted to Torch tensors in lines 19 and 33 and for the specific case of the ScConv2d(), the outputs must be normalized to bipolar format (line 35).

The variable *accumulationsNo* indicates the number of times / iterations that a multiply-accumulate operation took place inside the innermost loop, for a 2D conv with a 2x2 kernel this value is 4, for 3x3 is 9 and so on.., finally it's relevant to explain that CPU version of ScConv2d() supports padding, stride and dilation and they can be passed to the function in that order (line 32); naturally for a 2D convolution with no padding, stride nor dilation, the default values are 0,1,1. In the provided example the padding is equal to 1.

```
1  import torch
2  import torch.nn as nn
3  import torch.nn.functional as F
4  import sc_torch_cpp
5
6  RandomNumberGenType = sc_torch_cpp.RandomNumberGenType
7  BitstreamRepresentation = sc_torch_cpp.BitstreamRepresentation
8  gen_type = RandomNumberGenType.MT19937
9  mode = BitstreamRepresentation.BIPOLAR
10 bitstreamLength = 300
11
12 #SC FULLY CONNECTED LAYER EXAMPLE:
13 inputs = [random.uniform(-1.0, 1.0) for _ in range(128)] # Generate 128 random weights
14 bias = [random.uniform(-1.0, 1.0) for _ in range(10)] # Generate 10 random biases
15 weights = [[random.uniform(-1.0, 1.0) for _ in range(10)] for _ in range(128)]
16
17 sc_NN = sc_torch_cpp.ScFcLayer(weights, bias, bitstreamLength, gen_type, mode)
18 scFc_Layer = sc_NN.forward(inputs)
19 tensor_ScFc_layer = torch.tensor(scFc_Layer)
20
21 #SC 2D CONVOLUTION EXAMPLE:
22 # Define input tensor
23 input_tensor = [[.1, -.2, .3],
24                 [.4, .5, -.6],
25                 [.7, -.8, .9]]
26 # Define kernel
27 kernel_tensor = [[1, 0], [-1, -0.5]]
28
29 sc_input_tensor = sc_torch_cpp.StochasticTensor(input_tensor, bitstreamLength, gen_type, mode)
30 sc_kernel_tensor = sc_torch_cpp.StochasticTensor(kernel_tensor, bitstreamLength, gen_type, mode)
31
32 sc_2Dconv = sc_torch_cpp.ScConv2d(sc_input_tensor, sc_kernel_tensor, 1, 1, 1)
33 tensor_SCconv = torch.tensor(sc_2Dconv)
34 accumulationsNo = 4
35 tensor_SCconv = (2 * (tensor_SCconv / (accumulationsNo * bitstreamLength)) - 1) * accumulationsNo
```

Listing 4.12: CPU SC Functions implemented in python

4.3.1 PyTorch CNN Training & Inference

The first general objective of this work is to implement these SC libraries in a Stochastic Computing Convolutional Neural Network and evaluate their overall functionality but to do that, they have to be compared to a 'normal' CNN using traditional FP computation. The CNN that will be used for this purpose is the classic LeNet 5, briefly discussed previously already. A quick reminder of its overall architecture is: Two 2D convolutions + activation function and pooling, and three fully connected layers. PyTorch documentation is quite descriptive and helpful when diving into the framework and it already provides some recipes and tutorials detailing how to build and train some CNN models, in [38, 39] it can be found how to define a LeNet NN with either MNIST or CIFAR-10 datasets.
To test the correct functionality of this SC extension and the inference capabilities of a SC CNN using these SC functions, the weights and biases of a 'normal' model are obtained to be used later in the SC CNN.

In a first instance, a different and very demure alternative to MNIST is the $Fashion - MNIST$ dataset provided by TorchVision which is used here to obtain a sequence of weights and biases for every layer of the 'normal' model. In [39] is provided how to define and train a model using this dataset, nonetheless, to effectively implement SC into a CNN, modifications must be made when training the model; the most important adjustments are related to data normalization. Listing 4.13 effectively shows the CNN model from which weights and biases are taken for SC CNN inference but for Stochastic Computing to work out, the weights and biases must be clipped to [-1,1] and the outputs of every convolutional and fully connected layer be normalized to [0,1]; in this way every value in a SC CNN can be properly converted to Bipolar or Unipolar stochastic representation and successful inference can be achieved with respect to the 'normal' CNN with FP computation.

```
1  import torch
2  import torch.nn as nn
3  import torch.nn.functional as F
4
5  def normalize_to_01(tensor):
6      min_val = torch.min(tensor)
7      max_val = torch.max(tensor)
8      normalized_tensor = (tensor - min_val) / (max_val - min_val)
9      return normalized_tensor
10
11 # PyTorch models inherit from torch.nn.Module
12 class GarmentClassifier(nn.Module):
13     def __init__(self):
14         super(GarmentClassifier, self).__init__()
15         self.conv1 = nn.Conv2d(1, 6, 5)
16         self.pool = nn.MaxPool2d(2, 2)
17         self.conv2 = nn.Conv2d(6, 16, 5)
18         self.fc1 = nn.Linear(16 * 4 * 4, 120)
19         self.fc2 = nn.Linear(120, 84)
20         self.fc3 = nn.Linear(84, 10)
21
22     def forward(self, x):
23         conv1_out = self.conv1(x)
24         pool_out = self.pool(F.relu(conv1_out))
25         nomalised_pool_out = normalize_to_01(pool_out)
26         conv2_inter = self.conv2(nomalised_pool_out)
27         conv2_out = self.pool(F.relu(conv2_inter))
28         normalised_conv2_out = normalize_to_01(conv2_out)
29         flattened = normalised_conv2_out.view(-1, 16 * 4 * 4)
30         fc1_out = F.relu(self.fc1(flattened))
31         normalised_fc1_out = normalize_to_01(fc1_out)
32         fc2_out = F.relu(self.fc2(normalised_fc1_out))
33         normalised_fc2_out = normalize_to_01(fc2_out)
34         fc3_out = self.fc3(normalised_fc2_out)
35         return conv1_out, pool_out, nomalised_pool_out, conv2_inter, conv2_out,
36             normalised_conv2_out, flattened, fc1_out, normalised_fc1_out, fc2_out,
37             normalised_fc2_out, fc3_out
38
39 net = GarmentClassifier()
```

Listing 4.13: LeNet model with normalized outputs

The normalization function in line 5 follows simple Min-Max normalization and it is called after relu-ing and pooling every output layer except from the last FC layer. Then, in Listing 4.14, standard cross-entropy loss and the Stochastic Gradient Descent optimizer are used to train the model for 10 epochs while normalizing the weights to [-1,1] with the *clip_weights*() function in line 7.

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

# Function to clip weights
def clip_weights(model, min_value=-1, max_value=1):
    for param in model.parameters():
        param.data.clamp_(min_value, max_value)

for epoch in range(10):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(training_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        conv1_out, pool_out, nomalised_pool_out, conv2_inter, conv2_out, normalised_conv2_out,
            flattened, fc1_out, normalised_fc1_out, fc2_out,
            normalised_fc2_out, fc3_out = net(inputs)
        loss = criterion(fc3_out, labels)
        loss.backward()
        optimizer.step()

        # Apply weight clipping
        clip_weights(net)

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0

print('Finished Training')

PATH = './mnist_net.pth'
torch.save(net.state_dict(), PATH)

net = GarmentClassifier()
net.load_state_dict(torch.load(PATH))
```

Listing 4.14: LeNet model training with clipped weights

The model is saved in './mnist_net.pth' so the normalized weights can be inserted later in the SC CNN; but in order to correctly build an entire SC CNN with the developed SC functions, it is necessary to complete first the 2D SC convolution: what the ScCudaConv2d() function does is basically the same as the torch.nn.Conv2d(1, 1, 5) function, taking only 1 input channel, 1 output channel, a 5x5 gaussian convolutional kernel and default values for each hyperparameter (padding=0, stride=1 and dilation=1), therefore, to get a SC 2D convolution considering more input and output channels, the ScCudaConv2d() is wrapped with a python function that iterates over the number of required input and output channels and passes to it each of the convolutional kernels (weights) trained previously; Listing 4.15 shows exactly the code in charge of this. There are a few important details to explain about the following wrapper function, the first is that it already takes for granted the use of a 5x5 kernel, that is why the size of the convolutional output layer in lines 5 and 9 subtract 4 to both input height and width and also why the number of accumulations needed to denormalize the output values to bipolar probabilities multiplies 5*5*input_channels (line 6).

```
1  def Sc_Conv2_py(custom_weights, custom_biases, input_tensor, h_INPUT, h_KERNEL, input_channels,
2      output_channels, bitstream_Length):
3      batch_size, _, height, width = input_tensor.shape
4      # Output dimensions after 5x5 convolution
5      output_tensor = torch.zeros(output_channels, height - 4, width - 4)
6      accumulationsNo = 5*5*input_channels
7
8      for out_channel in range(output_channels):
9          output = torch.zeros(height - 4, width - 4)  # Temporary output for each output channel
10         for in_channel in range(input_channels):
11             Sc_custom_weights = custom_weights[out_channel, in_channel, :, :].tolist()
12             Sc_custom_Input = input_tensor[0, in_channel, :, :].tolist()
13
14             random_matrix_INPUT = ScCudaTorch.generate_random_matrix(h_INPUT,
15                 bitstream_Length, rng_type)
16             random_matrix_KERNEL = ScCudaTorch.generate_random_matrix(h_KERNEL,
17                 bitstream_Length, rng_type)
18
19             output += torch.tensor(ScCudaTorch.ScCudaConv2d(Sc_custom_Input, Sc_custom_weights,
20                 random_matrix_INPUT, random_matrix_KERNEL))
21
22         output = (2 * (output / (accumulationsNo * bitstream_Length)) - 1) * accumulationsNo
23         output += custom_biases[out_channel]  # Add the bias for the current output channel
24         output_tensor[out_channel, :, :] = output  # Assign to the output tensor
25     return output_tensor
```

Listing 4.15: Wrapper function for ScCudaConv2d() including # of input & output channels.

Before calling ScCudaConv2d() function with the **ScCudaTorch** keyword (lines [19,20]), it is required to create two matrices with random FP values from 0 to 1 and pass them as parameters of ScCudaConv2d() (lines [14,17]), for these will be used in the Stochastic Tensor Generators. Denormalization to bipolar is done in line 22 after every output layer within the innermost loop around input_channels has been properly added and allocated (line 19) in the temporary output for each output channel (defined in line 9).

The following Listing 4.16 presents an example on how to use the **Sc_Conv2_py()** function with python in order to obtain the outputs of the first 2D convolutional layer (before relu+pooling) of the normalized LeNet model shown in Listing 4.13 using one image of the class labels in the *FashionMNIST* dataset for inference. Lines 1 and 2 correspond to the trained weights / biases of the 'normal' CNN; these are to be converted to lists of FP values from Torch tensors inside the Sc_Conv2_py() wrapper function, which in turn passes them to ScCudaConv2d() that converts them to stochastic bitstreams (lines [11,12]).

```
1  custom_weights = net.conv1.weight.data
2  custom_biases = net.conv1.bias.data
3  bitstream_Length = 2048
4  input_channels = 1
5  output_channels = 6
6
7  h_INPUT = 28*28 # height of the matrix //NUMBER OF ELEMENTS IN INPUT
8  h_KERNEL = 5*5  # height of the matrix //NUMBER OF ELEMENTS IN KERNEL
9
10 start_time = time.time()
11 custom_output = Sc_Conv2_py(custom_weights, custom_biases, original_image, h_INPUT, h_KERNEL,
12     input_channels, output_channels, bitstream_Length)
13 print(f"Execution time: {(time.time() - start_time):.2f} seconds")
14
15 similarity_score = tensor_similarity(conv1_out.squeeze(0), custom_output)
16
17 error_percentage = calculate_mean_absolute_error(conv1_out.squeeze(0), custom_output)
```

Listing 4.16: Sc_Conv2_py() example in python

An example on how to use ScCudaFcLayer() is now presented in Listing 4.17. The variables in the first lines 1,2,3 take the weights and biases tensors from the trained model and convert them to lists. The weights tensor must be transposed first. Then the matrices of random [0,1] values are created for the inputs, the weights and biases and passed as parameters to ScCudaFcLayer(). The first parameter *output_sc_FcLayer2_tensor_norm* of the function is the normalized to [0,1] output layer of the previous FC layer + ReLU.

```
1  new_weightsfc = net.fc3.weight.data.t().tolist()
2  new_biasfc = net.fc3.bias.data
3  new_biasfc = new_biasfc.tolist()
4
5  noInputs = 84
6  noOutputs = 10
7
8  new_INPUTfc_rndomMatrix = ScCudaTorch.generate_random_matrix(noInputs,
9      bitstream_Length, rng_type)
10 new_weightsfc_rndomMatrix = ScCudaTorch.generate_random_matrix(noInputs*noOutputs,
11     bitstream_Length, rng_type)
12 new_biasfc_rndomMatrix = ScCudaTorch.generate_random_matrix(noOutputs,
13     bitstream_Length, rng_type)
14
15 accumulationsNo = noInputs + 1
16
17 start_time = time.time()
18 output_sc_FcLayer3 = torch.tensor(ScCudaTorch.ScCudaFcLayer(output_sc_FcLayer2_tensor_norm,
19     new_weightsfc, new_biasfc, new_INPUTfc_rndomMatrix, new_weightsfc_rndomMatrix,
20     new_biasfc_rndomMatrix, noOutputs))
21 print(f"Execution time: {(time.time() - start_time):.2f} seconds")
22 output_sc_FcLayer3 = (2*(output_sc_FcLayer3/(accumulationsNo*bitstream_Length))-1)*accumulationsNo
23
24 similarity_score = tensor_similarity(fc3_out.squeeze(0), output_sc_FcLayer3)
25
26 error_percentage = calculate_mean_absolute_error(fc3_out.squeeze(0), output_sc_FcLayer3)
```

Listing 4.17: ScCudaFcLayer() example in python

In both Listing 4.16 and Listing 4.17 the last two variables: *similarity_score* and *error_percentage* calculate the cosine similarity and mean absolute error metrics of the output tensors calculated with SC to be compared with the output tensors of the 'normal' CNN model for the same 1rst conv2d layer and for the same 3rd FC layer. This is exactly how overall Stochastic Computing accuracy for CNN inference is evaluated. If the cosine similarity is closer to 1 and the mean error closer to 0 that basically means the accuracy of these SC functions is sufficient, or excellent, or bad/useless. The functions that calculate these metrics are shown in Appendix Listing 8.6.
The *FashionMNIST* dataset defines 10 class labels = ('T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle Boot') for grayscale images with a 28x28 pixels size, so only one sample of any of these classes is needed to test the forward propagation inference of a SC CNN using the SC functions.

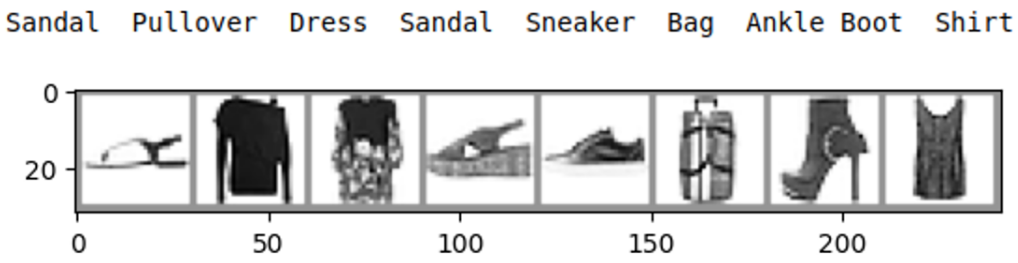


Figure 4.8: Some *FashionMNIST* class samples.

In Table 4.3 there are presented some metrics obtained after passing the initial input tensor of an image of a random 'sandal' (from the validation set, not from training set evidently) into the SC CNN model compared to the output layers of the original CNN model. The complete architecture of the normalized SC CNN is presented using the so called "Unipolar SC CUDA functions" in subsection 4.5; also some important parameters to mention: the bitstream length for this particular example was 1024 and the type of RNG used to create the random numbers' matrices for the STGs was Mersenne Twister [32].

Metric	1rst Sc_Conv2	2nd Sc_Conv2	1rst ScCudaFcLayer	2nd ScCudaFcLayer	3rd ScCudaFcLayer
Execution time[s]	4.32	12.78	15.02	4.93	0.45
Similarity score	0.9977	0.9195	0.5399	0.5818	0.7823
Mean absolute error	0.06	0.22	0.09	0.19	2.34

Table 4.3: Some initial metrics for SC CNN forward propagation inference.

It can be observed that the similarity score decrements and the mean absolute error increments with every additional layer, which is not ideal for accurate inference; also the execution time increases as there are more inputs in both Sc_Conv2 and ScCudaFcLayer. Next, in Table 4.4, it is possible to see the actual values of the last fully connected layer of the SC CNN for different bitstream lengths and their mean absolute errors when compared to the last FC layer of the original CNN.

Bitstream Length	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle Boot	m. abs. error
2048 (SC CNN)	-1.3437	-0.1777	-0.0820	-1.0840	-1.1211	3.5625	-0.9453	1.8398	-0.7520	-0.0488	1.84
1024 (SC CNN)	1.2188	-1.6406	-0.2715	-0.3828	-1.7686	2.3252	0.1562	0.3662	-0.5820	0.1553	2.34
512 (SC CNN)	0.2109	-1.3281	2.3320	1.7813	0.7617	-0.0508	0.9883	-0.8477	-1.0820	-2.4219	3.32
3° Fc Layer (CNN)	-0.2262	-0.6120	-0.8006	-2.8583	-4.0558	10.7591	-1.6378	2.8466	-2.1727	-1.1517	/

Table 4.4: 3rd Fully connected layer probabilities from SC CNN & the reference CNN.

Naturally, as the bitstream length decreases, the error increases (in red) and the values are further away from the original 3°FcLayer values of the CNN acting as ground truth. For a bitstream length of 512, there was not even possible to correctly infer the sample of a 'sandal' class (in blue), as the highest probability was 2.332 (in purple) corresponding to the 'Pullover' class.

Given the previous results and metrics it is then required to ask: is it possible to improve the accuracy and overall performance in terms of throughput ($\frac{\#\ of\ tensor\ values}{second}$)?
The answer lies in [9], which introduces the idea of *Unipolar SC Hardware for CNN*. This development basically comprises the usage of Unipolar stochastic numbers instead of Bipolars so to improve the overall accuracy of the forward propagation inference in a SC CNN. The hardware architecture of its computing neurons consists of AND gates for multiplications, a parallel counter for accumulation, and a stochastic ReLU module for activation, as shown in Figure 4.9.

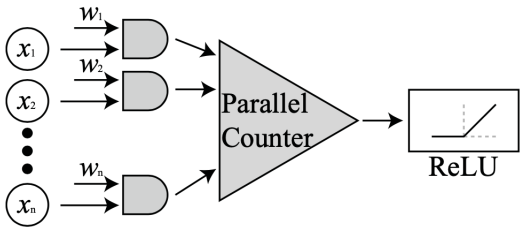


Figure 4.9: Structure of a Unipolar SC neuron. (Taken from [9])

In this process, stochastic bitstreams from prior layers are combined with weight bitstreams using AND gates. Since the input bitstreams are non-negative due to ReLU activation, while weights may be negative, the multiplication's sign matches the weight's sign. The AND gate computes only the magnitude in unipolar encoding, and the weight's sign is stored. For accumulating these multiplication results, each neuron employs two small adder trees—one for negative products and one for positive ones. Each adder tree functions as a parallel counter, counting the 1s in each cycle to produce separate negative and positive sums. These two sums are then combined to yield the final accumulated result for that cycle.

4.4 Towards Precision & Execution Time Optimization

It is precisely the *Unipolar SC Hardware for CNN* idea introduced previously that allows for overall accuracy optimization of the forward propagation inference of SC CNNs with respect to its implementations using Stochastic Computing in Bipolar format. Then, this technique is replicated in the SC CUDA functions and tested with the model of a LeNet-5 SC CNN using the previous greyscale Fashion-MNIST dataset and also, in order to elevate a notch the amount of computed data, the RGB-coloured CIFAR-10 dataset is tested as well, so to observe the behaviour of Sc_Conv2 and ScCudaFcLayer with inputs comprising more than just one input channel and a wider range of pixel colors and intensities.

4.4.1 Unipolar CUDA Stochastic Tensor Generator

The code for the Unipolar version of the STG CUDA kernel is in Listing 4.18. It has some similarities to its former Bipolar version, with the addition of conditional statements that tell each thread id what to do if the inputted tensor floating point (FP) and normalized to [-1,1] value is equal, below or above 0.

```
__global__ void stochasticTensorGenerator(const float* inputData, const float* randomMatrix,
int8_t* output, int inputData_size, int RM_cols) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < inputData_size * RM_cols) {
        int n_idx = idx / RM_cols;
        int rm_col_idx = idx % RM_cols;
        int rm_idx = n_idx * RM_cols + rm_col_idx;

        // Determine if inputData[n_idx] is positive or negative
        if (inputData[n_idx] > 0) {
            // Add 1 to all positions separated by RM_cols (bitstream length)
            if (rm_col_idx == 0) {
                output[idx] = 1; } else {
                output[idx] = (randomMatrix[rm_idx] < (inputData[n_idx])) ? 1 : 0; }
        } else if (inputData[n_idx] < 0) {
            // Add -1 to all positions separated by RM_cols (bitstream length)
            if (rm_col_idx == 0) {
                output[idx] = -1; } else {
                output[idx] = (randomMatrix[rm_idx] < ((inputData[n_idx] * -1))) ? 1 : 0; }
        } else if (inputData[n_idx] == 0) {
            output[idx] = 0; }
    }
}
```

Listing 4.18: Unipolar STG CUDA kernel

The core of the index arithmetic remains the same as of the bipolar STG (lines 5, 6, 7 explained in Table 4.1) however, if the input probability is above 0, a one is added to all positions separated by RM_cols (equal to the bitstream length); if the input probability is below 0, a minus one (-1) is added to all positions separated by RM_cols (equal to the bitstream length) and if the input probability is equal to 0, the whole bitstream is filled with zeros as well. This way simulates effectively how to store the sign of the inputs and weights being passed to the STG by 'sacrificing' just one bit location (the first location) of every stochatic bitstream. Naturally, these 'binary' (non-stochastically generated) positive or negative ones are not to be multiplied nor accumulated in neither SC 2D convolution nor the SC FC layer, for they are just skipped.

4.4.2 Unipolar SC CUDA Conv2d

The code for the Unipolar version of the SC CUDA 2D convolution kernel is in Listing 4.19 and it has some modifications respect to its former Bipolar version. In a first instance, it transfers the constant integer values of the width dimension of the input feature map, the convolutional kernel dimensions (width and height), the output dimensions (width and height) and the bitstream length all of them to **constant memory**.

```
1  __constant__ int inpWidthC2;
2  __constant__ int KernelHeightConv2;
3  __constant__ int KernWidthC2;
4  __constant__ int OutputHeightConv2;
5  __constant__ int OutputWidthConv2;
6  __constant__ int NConv2;
7
8  __global__ void conv2D(const int8_t* input, const int8_t* kernel, float* output) {
9      int i = blockIdx.y * blockDim.y + threadIdx.y;
10     int j = blockIdx.x * blockDim.x + threadIdx.x;
11     if (i < OutputHeightConv2 && j < OutputWidthConv2) {
12         float accumulatedOnes_pos = 0;
13         float accumulatedOnes_neg = 0;
14         for (size_t m = 0; m < KernelHeightConv2; ++m) {
15             for (size_t n = 0; n < KernWidthC2; ++n) {
16                 if((input[((i + m) * inpWidthC2 * NConv2 + (j + n) * NConv2)])
17                         * (kernel[(m * KernWidthC2 * NConv2 + n * NConv2)]) == 1){
18                     for (size_t bit_counter = 1; bit_counter < NConv2; ++bit_counter) {
19                         if(((input[((i + m) * inpWidthC2 * NConv2 + (j + n) * NConv2)+bit_counter]
20                             & kernel[(m * KernWidthC2 * NConv2 + n * NConv2)+bit_counter])) == 1)
21                         {accumulatedOnes_pos++;} }
22                 } else if ((input[((i + m) * inpWidthC2 * NConv2 + (j + n) * NConv2)])
23                         * (kernel[(m * KernWidthC2 * NConv2 + n * NConv2)])== -1){
24                     for (size_t bit_counter = 1; bit_counter < NConv2; ++bit_counter) {
25                         if(((input[((i + m) * inpWidthC2 * NConv2 + (j + n) * NConv2)+bit_counter]
26                             & kernel[(m * KernWidthC2 * NConv2 + n * NConv2)+bit_counter])) == 1)
27                         {accumulatedOnes_neg++;} }
28                 }
29             }
30         }
31         output[i * OutputWidthConv2 + j] = accumulatedOnes_pos - accumulatedOnes_neg;
32     }
33 }
```

Listing 4.19: Unipolar SC CUDA Conv2 kernel

A representation of constant memory can be seen colored in light blue (next to L1/SharedMemory) in Figure 4.10. As it may be possible to infer, constant memory has less latencies than global memory because its closer to the streaming multiprocessors (GPU cores) therefore memory accesses are faster and could save some micro-seconds of computation time, but to take in mind it is Read-only and is limited to 64KB.

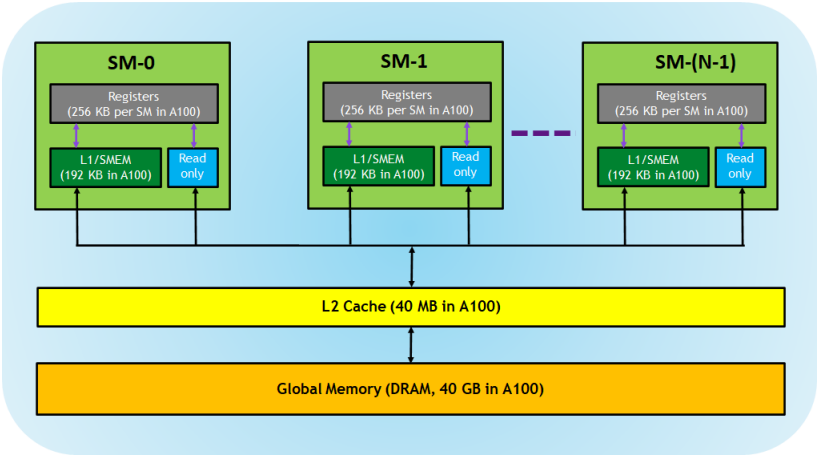


Figure 4.10: Memory hierarchy in GPUs. (Taken from [40])

The next relevant modification is directly related to the SC multiply-accumulate operation. The two variables allocated in memory registers: *accumulatedOnes_pos* and *accumulatedOnes_neg* represent the results of the Unipolar multiplication.
accumulatedOnes_pos represents the positive result of the multiplication between a positive input and a positive weight or a negative input and negative weight while *accumulatedOnes_neg* represents the negative result of the multiplication between a positive input and a negative weight or a positive input and negative weight. The multiplication of bitstreams is an AND operation and if the result/product is equal to one, either *accumulatedOnes_pos* or *accumulatedOnes_neg* are incremented (in lines 21 and 27). Finally, the subtraction of both variables (in line 31) is the value of the accumulated ones that represents the real value of the final counter (before conversion to Unipolar probability) so this is sent to the (flattened) output tensor of the SC 2D convolution. In Figure 4.11 can be observed an example of the output tensor after a Unipolar SC 2D convolution with random parameters: 6 input channels & 16 output channels, 12x12 input features map & 5x5 convolutional kernel, and a bitstream length of 1024 compared to the outputs of a normal FP 2D convolution with the same parameters. The cosine similarity score between the two tensors is 0.9955 and the mean absolute error: 0.05.

```
Unipolar SC conv2d_out:
tensor([[[-1.1381, -1.0141, -0.7065,  ..., -0.6596, -0.7709, -0.8217],
         [-0.9487, -0.5395, -0.1801,  ..., -0.3393, -0.1713, -0.4428],
         [-0.9008, -0.1704, -0.3647,  ..., -0.2915, -0.0854, -0.3032],

         [-1.4409, -1.8081, -1.6137,  ..., -1.7905, -1.9907, -1.8706],
         [-1.5717, -1.6577, -1.4077,  ..., -0.8774, -1.0473, -1.2270],
         [-1.1733, -1.1762, -1.1030,  ..., -1.1704, -1.1567, -1.1352]],

        [[-0.5905, -0.6520, -0.6979,  ..., -0.6881, -0.4762, -0.3131],
         [-0.1920, -0.6910, -1.3756,  ..., -1.1774, -0.9801, -0.7780],
         [ 0.1810,  0.4213,  0.6351,  ..., -1.1842, -1.1949, -0.7213]]])
```

```
NORMAL conv2d_out:
tensor([[[-1.1418, -1.0208, -0.6709,  ..., -0.5911, -0.7173, -0.8007],
         [-0.9335, -0.4811, -0.0882,  ..., -0.2394, -0.0819, -0.3875],
         [-0.8953, -0.0791, -0.3252,  ..., -0.1987,  0.0155, -0.2785],

         [-1.4705, -1.8527, -1.7044,  ..., -1.6350, -1.9804, -1.8648],
         [-1.5613, -1.6311, -1.4139,  ..., -0.9175, -1.0691, -1.2144],
         [-1.1577, -1.1784, -1.0846,  ..., -1.1067, -1.1191, -1.0982]],

        [[-0.5719, -0.6559, -0.6923,  ..., -0.6508, -0.4476, -0.3843],
         [-0.2605, -0.7276, -1.2350,  ..., -1.1132, -0.9195, -0.7100],
         [ 0.2842,  0.3680,  0.6468,  ..., -1.2362, -1.1332, -0.7053]]])
```

Figure 4.11: Unipolar SC CUDA Conv2 output tensor vs normal Conv2 outputs

4.4.3 Unipolar SC CUDA FCLayer

The code for the Unipolar version of the SC CUDA FCLayer kernel is in Listing 4.20 and it also has some modifications respect to its former Bipolar version. The first important modification is also on the implementation of constant memory to store constant variables which are size of input *input_sizeFcL* and output *outSizeFcL* neurons and the bitstream length N_fcl. Then, just like in Unipolar SC CUDA Conv2 kernel, there are two variables allocated in memory registers: *accumulatedOnes_pos* and *accumulatedOnes_neg* that represent the results of the Unipolar multiplication with AND. Let's remember if the sign of the multiplication result between the input neuron and the weights is positive then *accumulatedOnes_pos* is incremented otherwise, if the sign is negative, *accumulatedOnes_neg* is incremented and this multiply-accumulate operations happen serially for every input neuron (see in lines [11,21]). Once every input neuron bitstream has been multiplied-accumulated with its corresponding bitstreams of weights, then it comes the addition of the biases, which is basically an extra accumulation to *accumulatedOnes_pos* or *accumulatedOnes_neg* (see lines [22,28]). Finally the subtraction of positive - negative accumulators is saved and returned (line 29), for it is converted to Unipolar probability in host code, which in this case is python/PyTorch.

```
1  __constant__ int input_sizeFcL;
2  __constant__ int outSizeFcL;
3  __constant__ int N_fcl;
4
5  __global__ void forward_pass(
6      int8_t* input, int8_t* weights, int8_t* biases, float* output) {
7      int j = blockIdx.x * blockDim.x + threadIdx.x;
8      if (j < outSizeFcL) {
9          float accumulatedOnes_pos= 0;
10         float accumulatedOnes_neg = 0;
11         for (size_t i = 0; i < input_sizeFcL; ++i) {
12             if((input[(i * N_fcl)]) * (weights[((i * outSizeFcL + j) * N_fcl)]) == 1){
13                 for (size_t k = 1; k < N_fcl; ++k) {
14                     if(((input[(i * N_fcl)+k] & weights[((i * outSizeFcL + j) * N_fcl)+k])) == 1)
15                         {accumulatedOnes_pos++;} }
16             } else if ((input[(i * N_fcl)]) * (weights[((i * outSizeFcL + j) * N_fcl)]) == -1){
17                 for (size_t k = 1; k < N_fcl; ++k) {
18                     if(((input[(i * N_fcl)+k] & weights[((i * outSizeFcL + j) * N_fcl)+k])) == 1)
19                         {accumulatedOnes_neg++;} }
20             }
21         }
22         if (biases[(j * N_fcl)] == 1) {
23             for (size_t biasIndex = 1; biasIndex < N_fcl; ++biasIndex) {
24                 accumulatedOnes_pos += biases[(j * N_fcl)+biasIndex]; }
25         } else if (biases[(j * N_fcl)] == -1) {
26             for (size_t biasIndex = 1; biasIndex < N_fcl; ++biasIndex) {
27                 accumulatedOnes_neg += biases[(j * N_fcl)+biasIndex]; }
28         }
29         output[j] = accumulatedOnes_pos - accumulatedOnes_neg;
30     }
31 }
```

Listing 4.20: Unipolar SC CUDA FCLayer kernel

In Figure 4.12 can be observed an example of the output tensor after a Unipolar SC FC layer with random parameters: 84 input neurons, 10 output neurons and a bitstream length of 1024 compared to the outputs of a normal FP 2D convolution with the same parameters. The cosine similarity score between the two tensors is 0.9973 and the mean absolute error: 0.24.

```
Unipolar SC FcLayer:
tensor([-0.0928, -0.4863, -0.4980, -3.1826, -3.9658, 10.8271, -1.2451,  2.2402,
        -2.3652, -0.9941])

ORIGINAL FcLayer:
tensor([[-0.2262, -0.6120, -0.8006, -2.8583, -4.0558, 10.7591, -1.6378,  2.8466,
         -2.1727, -1.1517]], requires_grad=True)
```

Figure 4.12: Unipolar SC CUDA FCLayer output tensor vs normal torch.nn.Linear() outputs

Important to mention, every single For loop in both Sc_Conv2 and ScCudaFcLayer starts the initial index from 1 instead of 0 to properly skip the first location that has stored the sign (1 or -1) of the bitstream:

```
for (size_t bit_counter = 1; bit_counter < NConv2; ++bit_counter)
```

4.4.4 C++/CUDA Kernels-Wrapper Functions

For the PyTorch extension to work, every CUDA kernel developed here must be wrapped in C++ functions that define important properties around the kernels execution like number of threads per block, no. of blocks per grid, data transference to GPU global or constant memories, implementation of CUDA streams, synchronization of threads or streams, pre- or post- processing of data like tensor flattening or unflattening and memory deallocation (release), amongst many other features.
Both Bipolar/Unipolar SC CUDA Conv2 kernels are wrapped in C++ wrapper function *ScCudaConv2d*. This asks for 4 input parameters which are the FP tensor of normalized 'pixel' values (or feature map), the tensor of convolutional weights and to generate the stochastic tensors of both these, the matrices of random numbers with # of columns equal to the bitstream length and # of rows equal to the total amount of input pixel values or weights. Then comes the creation of CUDA streams (lines [6,8]) to implement concurrent execution of kernels as can be portrayed in Figure 4.13, and then comes the flattening of the inputs (lines [9,10]).

```
1  std::vector<std::vector<double>> ScCudaConv2d(
2      const std::vector<std::vector<double>>& polarInputData,
3      const std::vector<std::vector<double>>& polarKernelData,
4      const std::vector<std::vector<double>>& randomMatrix_Input,
5      const std::vector<std::vector<double>>& randomMatrix_Kernel)
6      cudaStream_t stream1, stream2;
7      cudaStreamCreate(&stream1);
8      cudaStreamCreate(&stream2);
9      std::vector<float> inputData = flatten2D(polarInputData);
10     std::vector<float> RM_flat = flatten2D(randomMatrix_Input);
11     int inputData_size = inputData.size();
12     int RM_cols = randomMatrix_Input[0].size(); //bitstream length
13     int output_size = inputData_size * RM_cols;
14     float* d_inputData, d_RM;
15     int8_t* d_output;
16     cudaCheckError(cudaMalloc(&d_inputData, inputData_size * sizeof(float)));
17     cudaCheckError(cudaMalloc(&d_RM, inputData_size * RM_cols * sizeof(float)));
18     cudaCheckError(cudaMalloc(&d_output, output_size * sizeof(int8_t)));
19     int blockSize = 256;
20     int numBlocks = (output_size + blockSize - 1) / blockSize;
```

Listing 4.21: C++ kernel-wrapper function ScCudaConv2d part 1

Afterwards comes determination of the input and output data sizes (lines [11,13]) of the STG kernel that converts the image features map (input layer) to bitstreams, then memory allocation into GPU device of these previous sizes (lines 18,20) and then definition of no. of threads per block and no. of blocks per grid (lines [22,23]) of this first STG.

This same allocation of sizes into memory is repeated again for a second STG kernel that converts the convolutional kernel (weights) to bitstreams.
Particularly elaborating more on the topic of chosen # of threads and blocks, the number of threads is 256 in this case as a trial value, it could very well be modified to other value below 1024 (as this is the limit of threads per block) based on testing with different values and analyzing the kernel computation performance of each option, remembering it should also be a multiple of 32 according to the size of an active Warp.
To determine the minimum amount of blocks required to parallelize the input data: $(output\_size + blockSize - 1)$ / $blockSize$; this simple formula is used (line 23), although there can be more efficient ways to determine number of blocks like the CUDA **cudaOccupancyMaxActiveBlocksPerMultiprocessor** function which calculates the maximum number of active thread blocks per multiprocessor for a given kernel configuration; this function may maximize thread *Occupancy* and consequently performance too, but as it is used before kernel execution, for small or not so large data sizes its use may only add pre-processing overhead time; in general, selecting an optimal configuration of threads & blocks for a given CUDA kernel is not a trivial aspect [41] as it should be decided after further analysis and experimentation with profiling tools like NVIDIA Nsight Compute [42], but for the purposes of this work, the # of threads and # of blocks implemented are enough for the SC functions.
Next, in lines [25,28] comes definition of widths and heights of inputs, weights and output layer of the 2D convolution, allocation of output bitstream into GPU device memory and then comes definition of no. of threads in x and y dimensions and no. of blocks also in x and y dimensions for the Bipolar/Unipolar SC CUDA Conv2 kernel (lines [33,35]).

```
24      const int N = RM_cols; // bitstream length
25      int inputHeight = polarInputData.size(), inputWidth = polarInputData[0].size();
26      int kernelHeight = polarKernelData.size(), kernelWidth = polarKernelData[0].size();
27      int outputHeight = inputHeight - kernelHeight + 1;
28      int outputWidth = inputWidth - kernelWidth + 1;
29      float* h_output = new float[outputHeight * outputWidth];
30      float* d_outputConv2;
31      cudaCheckError(cudaMalloc(&d_outputConv2, outputHeight * outputWidth * sizeof(float)));
32      // Define grid and block dimensions for SC CONV2D KERNEL
33      dim3 blockDim(32, 32);
34      dim3 gridDim((outputWidth + blockDim.x - 1) / blockDim.x,
35          (outputHeight + blockDim.y - 1) / blockDim.y);
```

Listing 4.22: C++ kernel-wrapper function ScCudaConv2d part 2

In the next code fragment of the C++ wrapper function $ScCudaConv2d$, is where the actual execution of the STGs and SC Conv2d kernels take place, starting with transference of constant values to constant memory with $cudaMemcpyToSymbol()$ function (in lines [28,33]), then transference of inputs and weights' matrices with their corresponding FP random number matrices to STG 1 and 2 (lines [35,42]), followed by the launch of both STG kernels in stream 1 and 2. (lines [45,50]), immediately after comes synchronization of both streams and then that's when SC Conv2d kernel can be launched with the outputs of STG 1 and STG 2, still allocated in GPU device's global memory. Lastly comes the convolution output data transference to CPU host to variable h_output. To measure the end-to-end execution time of the whole process -from data allocation into device, through STG kernels execution for both inputs & weights data + SC Conv2D kernel execution, ending with data transference back to host-, CUDA events are created to record the time in milliseconds (lines [37,40] and [74,77]); with this time is how the throughput of both SC functions can be calculated.

```
            cudaEvent_t start, stop;
            cudaEventCreate(&start);
            cudaEventCreate(&stop);
            cudaEventRecord(start);
    cudaMemcpyToSymbol(inpWidthC2, &inputWidth, sizeof(int));
    cudaMemcpyToSymbol(KernelHeightConv2, &kernelHeight, sizeof(int));
    cudaMemcpyToSymbol(KernWidthC2, &kernelWidth, sizeof(int));
    cudaMemcpyToSymbol(OutputHeightConv2, &outputHeight, sizeof(int));
    cudaMemcpyToSymbol(OutputWidthConv2, &outputWidth, sizeof(int));
    cudaMemcpyToSymbol(NConv2, &N, sizeof(int));
    // Copy INPUT data to device
    cudaCheckError(cudaMemcpyAsync(d_inputData, inputData.data(), inputData_size * sizeof(float),
        cudaMemcpyHostToDevice, stream1));
    cudaCheckError(cudaMemcpyAsync(d_RM, RM_flat.data(), inputData_size * RM_cols * sizeof(float),
        cudaMemcpyHostToDevice, stream1));
    // Copy KERNEL data to device
    cudaCheckError(cudaMemcpyAsync(d_inputDataK, inputDataK.data(), inputData_sizeK*sizeof(float),
        cudaMemcpyHostToDevice, stream2));
    cudaCheckError(cudaMemcpyAsync(d_RMK, RM_flatK.data(), inputData_sizeK*RM_colsK*sizeof(float),
        cudaMemcpyHostToDevice, stream2));
    // Launch input image feature map STG kernel
    stochasticTensorGenerator<<<numBlocks, blockSize, 0, stream1>>>(d_inputData, d_RM,
        d_output, inputData_size, RM_cols);
    cudaCheckError(cudaGetLastError());
    // Launch weights STG kernel
    stochasticTensorGenerator<<<numBlocksK, blockSizeK, 0, stream2>>>(d_inputDataK, d_RMK,
        d_outputK, inputData_sizeK, RM_colsK);
    cudaCheckError(cudaGetLastError());
    cudaStreamSynchronize(stream1);
    cudaStreamSynchronize(stream2);
    conv2D<<<gridDim, blockDim>>>(d_output, d_outputK, d_outputConv2);
    cudaCheckError(cudaGetLastError());
    cudaCheckError(cudaMemcpy(h_output, d_outputConv2, outputHeight * outputWidth * sizeof(float),
                cudaMemcpyDeviceToHost));
            cudaEventRecord(stop);
            cudaEventSynchronize(stop);
            float milliseconds = 0;
            cudaEventElapsedTime(&milliseconds, start, stop);
```

Listing 4.23: C++ kernel-wrapper function ScCudaConv2d part 3

Finally, it comes the unflattening of the Unipolar/Bipolar tensor of output probabilities to be exposed in python followed by GPU device memory deallocation and that's the entirety of the whole C++ kernel-wrapper function: *ScCudaConv2d*, remembering that to be extended into python, a binder script is also needed, as presented in Appendix Listing 8.5.

```
    std::vector<std::vector<double>> outputConv2(outputHeight, std::vector<double>(outputWidth));
    for (int i = 0; i < outputHeight; ++i) {
        for (int j = 0; j < outputWidth; ++j) {
            outputConv2[i][j] = static_cast<double>(h_output[i * outputWidth + j]); } }
    // Free device memory
    cudaFree(d_inputData);
    cudaFree(d_RM);
    cudaFree(d_output);
    cudaFree(d_inputDataK);
    cudaFree(d_RMK);
    cudaFree(d_outputK);
    cudaFree(d_outputConv2);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
    delete[] h_output;
    return outputConv2;
}
```

Listing 4.24: C++ kernel-wrapper function ScCudaConv2d part 4

Both Bipolar/Unipolar SC Cuda FCLayer kernels are wrapped in C++ wrapper function *ScCudaFcLayer* and its structure is quite similar to *ScCudaConv2d*. This asks for 7 input parameters which are a vector of FP normalized input neurons, a matrix of FP weights, a vector of FP biases, the matrices of random numbers to convert each of these inputs to stochastic bitstreams and lastly the number of outputs. Just like in the previous *ScCudaConv2d* wrapper function, CUDA streams are created for concurrent kernel execution but instead of only 2 now it's 3 streams (one for input neurons, weights or biases) Figure 4.13, then comes flattening of all these inputted data structures, determination of sizes and their allocation into GPU device memory, definition of # of threads and blocks for the SC FC Layer kernel and each of the 3 STGs used to convert input neurons, weights and biases, then transference of flattened data to global memory and constant values to constant memory, then comes execution of the 3 STG kernels, synchronization of the 3 streams, execution of the FC Layer kernel, transference of outputs back to CPU host, measurement of end-to-end execution time and finally memory GPU device deallocation.

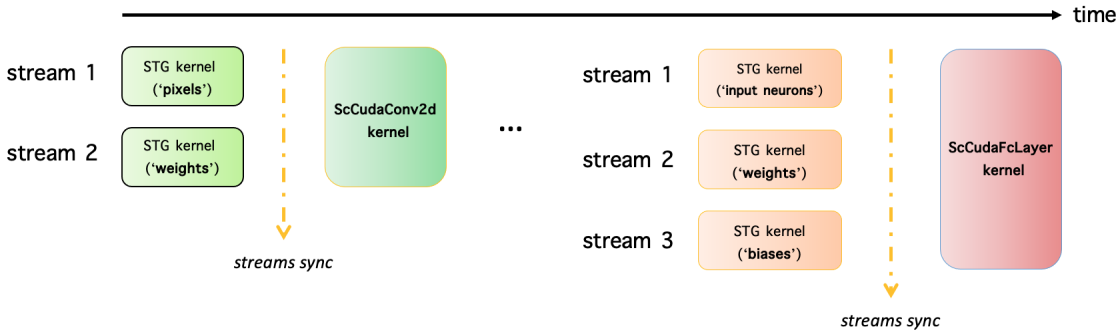


Figure 4.13: CUDA streams in ScCudaConv2d and ScCudaFcLayer

The complete *ScCudaFcLayer* wrapper function can be observed in Appendix Listing 8.7 and Listing 8.8. A short note on # of threads and blocks here: the same formula to determine minimum amount of blocks required to parallelize the input data is used ($int\ numBlocks = (output\_size + blockSize - 1)/blockSize$;) in each of the 4 CUDA kernels here implemented (since launching more blocks than actually needed adds overhead) and the # of threads is 256; after experimentation with different # of threads & blocks it could be determined a more efficient threads + blocks configuration depending on a higher *Occupancy* (active warps) and better thread coalescing (organizing memory accesses of threads into favorable patterns) to hide computation latencies [43], but then again, for the purposes of this work, the # of threads and # of blocks implemented here are enough for the SC functions.

4.5 SC CNN - PyTorch Implementation

Finally, to put the developed SC functions into action, it is necessary to build the forward propagation of a Stochastic Computing Convolutional Neural Network (SC CNN) model in a similar way as it is done commonly with PyTorch (one example was already introduced in subsubsection 4.3.1 Listing 4.13). The compact version of a LeNet-5 forward pass using the SC CUDA functions into python with PyTorch and its parameters are presented in Listing 4.25 and Listing 4.26. The following code fragment shows the required parameters to infer the features map of any image from the CIFAR10 dataset.

The weights and biases are taken from a trained model very similar to Listing 4.13 and Listing 4.14, with the only distinction that CIFAR10 uses RGB 32x32 images with 3 input channels instead of 1 channel and 28x28 pixels.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.cpp_extension import load

ScCudaTorch = load(
    name="sc_cuda_torch",
    sources=["ScCudaTorch.cpp", "ScCudaTorch.cu"],
    verbose=True )

############################## PARAMETERS ######################################
############################## FIRST SC CONV2D LAYER ############################
custom_weights1 = net.conv1.weight.data
custom_biases1 = net.conv1.bias.data
input_channels1 = 3
output_channels1 = 6
h_INPUT1 = 32*32
h_KERNEL1 = 5*5
############################## SECOND SC CONV2D LAYER ###########################
custom_weights2 = net.conv2.weight.data
custom_biases2 = net.conv2.bias.data
input_channels2 = 6
output_channels2 = 16
h_INPUT2 = 14*14
h_KERNEL2 = 5*5
############################## 1ST SC LINEAR LAYER ##############################
weightsFc3 = net.fc1.weight.data.t().tolist()
biasFc3 = net.fc1.bias.data
biasFc3 = biasFc3.tolist()
noInputs3 = 16*5*5
noOutputs3 = 120
no_accumulations3 = noInputs3 + 1
inputFcRndomMatrix3 = ScCudaTorch.generate_random_matrix(noInputs3,bStreamLength,_RNG)
weightsFcRndomMatrix3 = ScCudaTorch.generate_random_matrix(noInputs3*noOutputs3,bStreamLength,_RNG)
biasFcRndomMatrix3 = ScCudaTorch.generate_random_matrix(noOutputs3,bStreamLength,_RNG)
############################## 2ND SC LINEAR LAYER ##############################
weightsFc4 = net.fc2.weight.data.t().tolist()
biasFc4 = net.fc2.bias.data
biasFc4 = biasFc4.tolist()
noInputs4 = 120
noOutputs4 = 84
no_accumulations4 = noInputs4 + 1
inputFcRndomMatrix4 = ScCudaTorch.generate_random_matrix(noInputs4,bStreamLength,_RNG)
weightsFcRndomMatrix4 = ScCudaTorch.generate_random_matrix(noInputs4*noOutputs4,bStreamLength,_RNG)
biasFcRndomMatrix4 = ScCudaTorch.generate_random_matrix(noOutputs4,bStreamLength,_RNG)
############################## 3RD SC LINEAR LAYER  #############################
weightsFc5 = net.fc3.weight.data.t().tolist()
biasFc5 = net.fc3.bias.data
biasFc5 = biasFc5.tolist()
noInputs5 = 84
noOutputs5 = 10
no_accumulations5 = noInputs5 + 1
inputFcRndomMatrix5 = ScCudaTorch.generate_random_matrix(noInputs5,bStreamLength,_RNG)
weightsFcRndomMatrix5 = ScCudaTorch.generate_random_matrix(noInputs5*noOutputs5,bStreamLength,_RNG)
biasFcRndomMatrix5 = ScCudaTorch.generate_random_matrix(noOutputs5,bStreamLength,_RNG)
```

Listing 4.25: SC CNN parameters

The parameters of the first and second SC 2D convolution are tensors of weights and biases, input & output channels and the sizes of the input features map & convolutional kernel; then comes the parameters of the first, second and third SC fully connected layers, they are tensors of weights and biases too but converted to FP lists like: *e.g.* $weightsFc3 = net.fc1.weight.data.t().tolist()$ or $biasFc3 = biasFc3.tolist()$

so they can be passed to the C++/CUDA extension *ScCudaTorch.ScCudaFcLayer* function, then there are no. of input and output neurons and **no. of accumulations** (*e.g. no_accumulations4 = noInputs4 + 1*). This parameter is important to convert the output of the SC functions to Bipolar or Unipolar probabilities and it corresponds to the number of times a *multiply-accumulate* operation took place within the CUDA kernels; it can be pre-calculated simply as *kernel_width* × *kernel_height* × *input_channels* in case of a SC 2D convolution, and in case of a SC FC layer, as *number_of_input_neurons* + 1 (+1 due to the addition of biases). Lastly there comes generation of matrices of random FP normalized [0,1] numbers for the STGs of the inputs, weights and biases.

```
############################SC FORWARD PROPAGATION:: ########################
############################ FIRST SC CONV2D LAYER ############################
custom_output = Sc_Conv2_py(custom_weights1, custom_biases1, original_image, h_INPUT1,
    h_KERNEL1, input_channels1, output_channels1, bStreamLength)
pool = nn.MaxPool2d(2, 2)
pool2d_tensor = pool(F.relu(custom_output.unsqueeze(0)))
pool2d_tensor_norm = normalize_to_01(pool2d_tensor)
############################ SECOND SC CONV2D LAYER ############################
secnd_conv2d = Sc_Conv2_py(custom_weights2, custom_biases2, pool2d_tensor_norm, h_INPUT2,
    h_KERNEL2, input_channels2, output_channels2, bStreamLength)
pool = nn.MaxPool2d(2, 2)
sc_conv2_out = pool(F.relu(secnd_conv2d))
norm_sc_conv2_out = normalize_to_01(sc_conv2_out)
sc_flattened = torch.flatten(norm_sc_conv2_out, 0)
sc_flattened_list = sc_flattened.squeeze(0).squeeze(0).tolist()
############################ 1ST SC LINEAR LAYER ############################
output_sc_FcLayer1 = torch.tensor(ScCudaTorch.ScCudaFcLayer(sc_flattened_list, weightsFc3,
    biasFc3, inputFcRndomMatrix3, weightsFcRndomMatrix3, biasFcRndomMatrix3, noOutputs3))
output_sc_FcLayer1 = (output_sc_FcLayer1/(no_accumulations3*bStreamLength))*no_accumulations3
first_scfc_layer = F.relu(output_sc_FcLayer1)
norm_sc_FcLayer1_tensor = normalize_to_01(first_scfc_layer)
output_sc_FcLayer1_tensor_norm = norm_sc_FcLayer1_tensor.squeeze(0).squeeze(0).tolist()
############################ 2ND SC LINEAR LAYER ############################
output_sc_FcLayer2 = torch.tensor(ScCudaTorch.ScCudaFcLayer(output_sc_FcLayer1_tensor_norm,
    weightsFc4, biasFc4, inputFcRndomMatrix4, weightsFcRndomMatrix4, biasFcRndomMatrix4, noOutputs4))
output_sc_FcLayer2 = (output_sc_FcLayer2/(no_accumulations4*bStreamLength))*no_accumulations4
second_scfc_layer = F.relu(output_sc_FcLayer2)
norm_sc_FcLayer2_tensor = normalize_to_01(second_scfc_layer)
output_sc_FcLayer2_tensor_norm = norm_sc_FcLayer2_tensor.squeeze(0).squeeze(0).tolist()
############################ 3RD SC LINEAR LAYER  ############################
output_sc_FcLayer3 = torch.tensor(ScCudaTorch.ScCudaFcLayer(output_sc_FcLayer2_tensor_norm,
    weightsFc5, biasFc5, inputFcRndomMatrix5, weightsFcRndomMatrix5, biasFcRndomMatrix5, noOutputs5))
output_sc_FcLayer3 = (output_sc_FcLayer3/(no_accumulations5*bStreamLength))*no_accumulations5
############################ ::END SC FORWARD PROPAGATION:: ############################
# Define labels for each of the 10 values
labels = { 0: 'plane', 1: 'car ', 2: 'bird ', 3: 'cat ', 4: 'deer ', 5: 'dog ', 6: 'frog ',
    7: 'horse ', 8: 'ship ', 9: 'truck ' }
# Find the index of the maximum value in the tensor
max_index = torch.argmax(output_sc_FcLayer3).item()
# Retrieve the corresponding label and the maximum value
max_label = labels[max_index]
max_value = output_sc_FcLayer3[max_index].item()
# Print the label with the highest value
print(f"Final Output: {max_label}: {max_value}")
```

Listing 4.26: SC CNN forward propagation model

At last, the forward propagation model of a SC CNN is presented in the previous code fragment Listing 4.26. It's also segmented in 5 parts for better understanding and it uses elements of the PyTorch framework to do tensor operations like max pooling, ReLU-ing, flattening, min-max normalization, tensor squeezing, labeling and conversion to FP lists.

Particularly lines 19, 26 and 33 of the 1rst, 2nd and 3rd FC layer operations are essential because this is where conversion of the outputted tensor of accumulated ones into Unipolar or Bipolar probabilities takes place, considering that for Bipolar numbers denormalization is also needed so it should be included like:

```
(2*(output_sc_FcLayer2/(no_accumulations*bitstream_Length))-1)*no_accumulations
```

whereas for Unipolar numbers, conversion is straightforward:

```
(output_sc_FcLayer2/(no_accumulations4*bStreamLength))*no_accumulations4
```

Remembering that in the case of the SC 2D convolution, this conversion is found inside the Sc_Conv2_py wrapper function Listing 4.15.

In the end, the labels for each of the 10 classes within CIFAR10 or fashion_MNIST are defined, then the index of the maximum value in the last $output_s c_F c Layer3$ tensor is found and matched with its corresponding label to assert whether inference for a specific image is successful.

5 Results and Discussion

Thanks to the acceleration achieved by the GPUs implementation, the time complexity of the former SC functions in CPU was reduced from seconds or minutes to miliseconds, depending on the size of the 2D convolution input image and the input neurons in case of the SC fully connected layer. The time complexity of the new GPU accelerated functions is presented here:

$$O(C_{in} \times C_{out} \times k_h \times k_w \times N) \tag{8}$$

where: C corresponds to input and output channels, k is width and height of kernel and N is the bitstream length of the stochastic numbers. The previous equation in O(n) notation shows for the GPU implementation of the SC CUDA 2D convolution the elimination of two critical for loops using the width and height of the input image tensor, compared to Equation 5.

$$O(n_{in} \times N_{weights} + N_{biases}) \tag{9}$$

where: n is number of input neurons and N is the bitstream length of the stochastic bitstreams for weights or biases. In the same context, the previous equation in O(n) notation shows for the GPU implementation of the SC FC layer the elimination of one critical for loop using the number of outputs, compared to Equation 6. The for loops used to generate the bitstreams tensor were all eliminated by parallelization with GPU, so there is not a single iteration happening inside the stochastic tensor generator CUDA kernels, therefore the time notation of Equation 7 is reduced merely to GPU thread launch overhead, transference of data from CPU to device and viceversa and finally computation latency (how much time the thread takes to collect data from global memory and do an operation with it in the ALU).

An overview of all the work implemented in this thesis project can be seen here in Figure 5.1. First two functions to implement Stochastic Computing in a 2D convolution and a Fully connected layer were developed for both CPU with C++ and GPU with C++ and CUDA. Then, for the GPU CUDA kernels two implementations supporting Bipolar and Unipolar stochastic numbers are done so to compare the accuracy of using one format over the other. The results show evidently that Unipolar numbers are more accurate when doing inference with a SC CNN, as it will be presented hereafter. Then, using the Unipolar CUDA kernels, two SC CNN were implemented, in a first instance the LeNet-5 architecture introduced earlier was used to infer the classes from the fashionMNIST dataset which has got grayscale images of 28x28 sizes, and afterwards, a VGG9-like *Stochastic Computing Deep Convolutional Neural Network* (SC DCNN) was implemented to precisely **test and evaluate** the accuracy and throughput of the implemented CUDA SC functions with the CIFAR-10 dataset with colored images of 32x32 pixels. The VGG9-like also referred here as SCDCNN9 is based on the one implemented in [44], whose structure can be seen in Figure 5.2. It has 9 layers in total of which 6 are 2D convolutions and 3 fully connected layers. Evidently there is ReLU and max pooling after each convolution and convolution blocks. The convolutional weights are 3x3 sized and the padding is 1. Other hyperparameters are set to default values. The padding in the case of the SC DCNN9 was implemented externally for the SC CUDA functions do not support hyperparameters like the CPU functions do.

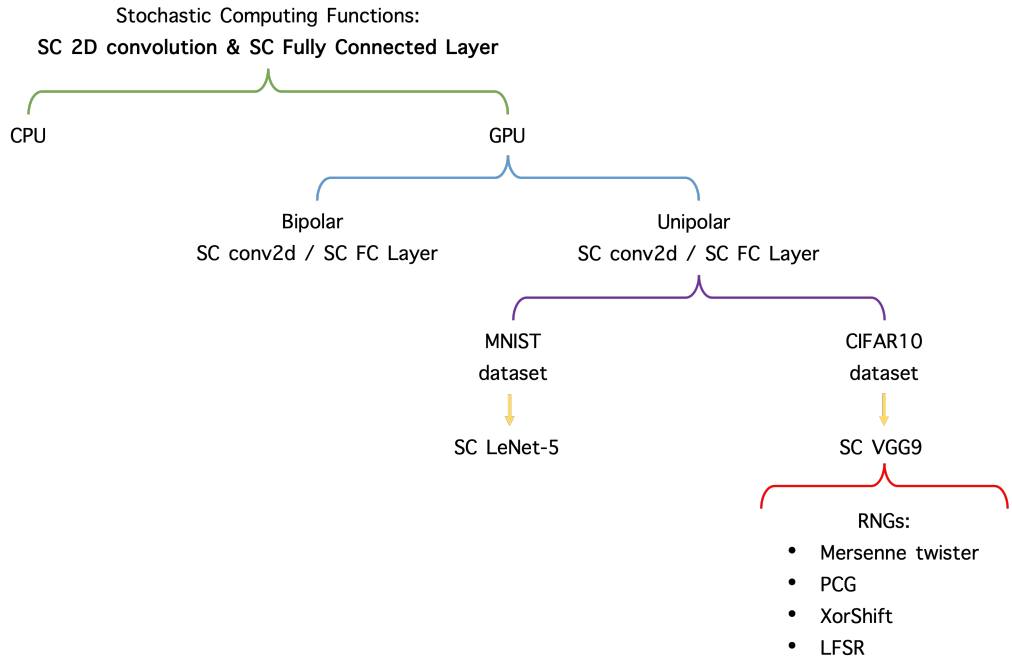


Figure 5.1: SC implementations overview.

Finally it is possible to see that 4 different types of software pseudo-random number generators were tested so as to effectively try to reduce the overall output error of the forward propagation of the sccnn9 with 1 image of each class.
After having trained the next DCNN model in PyTorch with 20 epochs at a learning rate of 0.001, clipping weights to [0,1] and output layers to [0,1] too, it is found the overall accuracy of the VGG9-like on the 10000 test images is of 67 %, which is better than LeNet-5 implemented with CIFAR-10 giving an accuracy of 54% [45]. Then, the trained weights and biases from this model are saved and inputted into the sccnn9 so they can be converted to stochastic bitstreams and SC inference can be tested.

```
SCCNN9(
  (pad1): ZeroPad2d(padding=(1, 1, 1, 1), value=0.0)
  (conv1_1): Conv2d(3, 8, kernel_size=(3, 3), stride=(1, 1))
  (conv1_2): Conv2d(8, 8, kernel_size=(3, 3), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2_1): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv2_2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3_1): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (conv3_2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=512, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=120, bias=True)
  (fc3): Linear(in_features=120, out_features=10, bias=True)
)
```

Figure 5.2: VGG9-like model implemented in the SC DCNN. Based on [44].

In the next subsections there is presented the results of implementing SC LeNet-5 with fashion MNIST and the VGG9-based SC DCNN9 with CIFAR-10. Special focus is set on throughput (operations per second and frames per second), accuracy percentage and mean percentage error of the stochastic output probabilities compared to the original normal probabilities.

5.1 CPU SC Functions' Execution Times & Precision

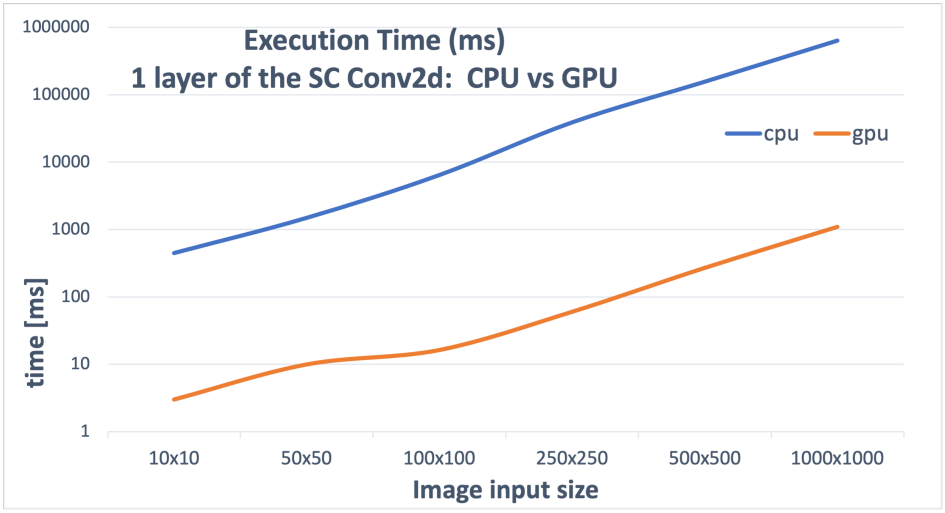


Figure 5.3: End-to-end execution time in ms of 1 layer of a 2D convolution for both CPU (in blue) and GPU (in orange).

The previous graph Figure 5.3 presents effectively the exponential growth that the CPU SC functions have when calculating a 2D convolution compared to the SC CUDA functions. The x axis shows the size of the input images and y axis shows time in milliseconds. Its clear that execution time for a large input size of 1000x1000 the SC CUDA is still in the range of 1 second while the CPU functions execution time is already in the range of mostly 1 hour. The same phenomenon happens below in Figure 5.4: there is exponential growth in the end to end execution time of the CPU SC functions of the fully connected layer while the execution time of the SC CUDA functions is still in the range of milliseconds for large numbers of inputs and output neurons. The overall similarity, accuracy or mean error of this CPU functions is basically the same as of the SC CUDA functions therefore the results and correlations of these will be observed next in the result analysis subsection of the SC CUDA functions.

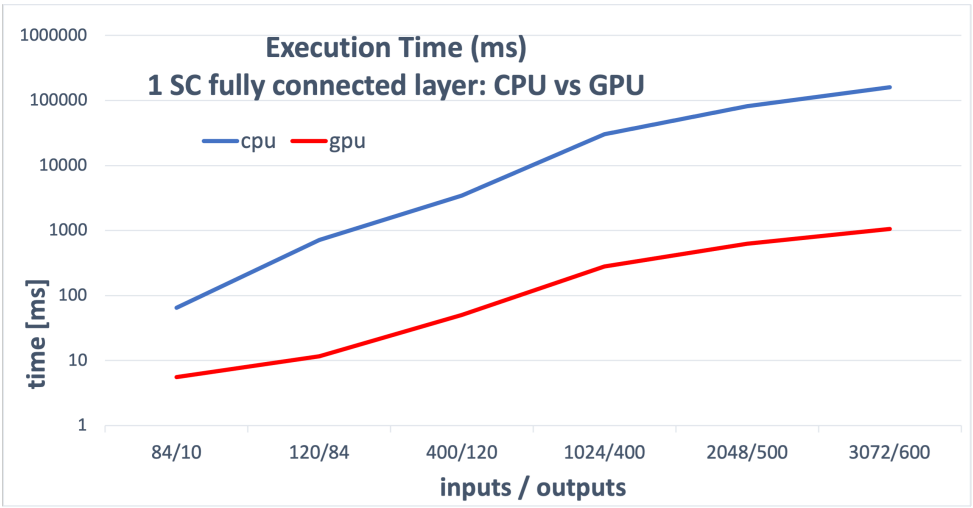


Figure 5.4: End-to-end execution time in ms of 1 fully connected layer for both CPU (in blue) and GPU (in orange).

5.2 SC CNNs comparison metrics and datasets

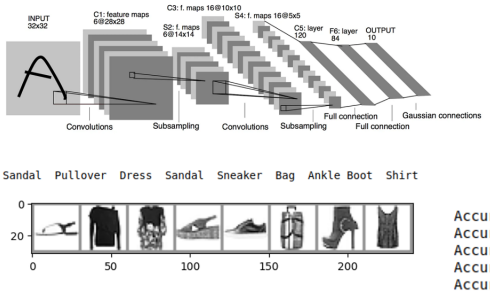
In order to get the comparisons of Bipolar and Unipolar functions, two SC DCNNs were developed which are a SC LeNet-5 with 2 convolutions and three fully connected layers, and a SC VGG9 with 6 convolutions and 3 fully connected layers. These networks were compared to their normal decimal numbers counterparts by means of metrics like Mean Absolute Error and the cosine similarity metric as seen in Figure 5.5.

$$\text{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} = \frac{\sum_{i=1}^{n} |e_i|}{n}. \qquad \text{similarity} = \frac{x_1 \cdot x_2}{\max(\|x_1\|_2, \epsilon) \cdot \max(\|x_2\|_2, \epsilon)}$$

(a) (b)

Figure 5.5: (a) Mean Absolute Error where y_i is prediction, x_i is true values and n is total number of data points. (b) Cosine similarity where x_1 and x_2 are first and second tensors and eps is a small value to avoid division by zero. Default: 1e-8

In the case of the SC LeNet-5 network, 100 images per class (1000 in total) which were previously inferred correctly by the normal LeNet-5, were taken so to make the comparisons with the stochastic computing LeNet5, as seen in Figure 5.6, and in the case of the SC VGG9 30 images per class (300 in total) which were previously inferred correctly by the normal VGG9 were taken so to make the comparisons with the stochastic computing VGG9 Figure 5.7.



Accuracy on 1000 instances per class (10K in total):

```
Accuracy for class: T-shirt/top is 86.0 %
Accuracy for class: Trouser is 97.2 %
Accuracy for class: Pullover is 83.6 %
Accuracy for class: Dress is 91.5 %
Accuracy for class: Coat  is 84.9 %
Accuracy for class: Sandal is 97.2 %
Accuracy for class: Shirt is 65.5 %
Accuracy for class: Sneaker is 97.3 %
Accuracy for class: Bag   is 97.0 %
Accuracy for class: Ankle Boot is 92.3 %
```

100 correctly inferred samples (1000 in total):

```
Accuracy for class: T-shirt/top with scnet is 100.0 %
Accuracy for class: Trouser with scnet is 100.0 %
Accuracy for class: Pullover with scnet is 100.0 %
Accuracy for class: Dress with scnet is 100.0 %
Accuracy for class: Coat  with scnet is 100.0 %
Accuracy for class: Sandal with scnet is 100.0 %
Accuracy for class: Shirt with scnet is 100.0 %
Accuracy for class: Sneaker with scnet is 100.0 %
Accuracy for class: Bag   with scnet is 100.0 %
Accuracy for class: Ankle Boot with scnet is 100.0 %
```

Figure 5.6: 100 correctly infered images per class by normal LeNet-5 from fashionMNIST.

Mean Absolute Error is a simple, interpretable metric that calculates the average absolute difference between each pair of elements. It's commonly used for regression tasks because it gives a sense of the average magnitude of errors, without favoring large or small errors disproportionately. It's good for comparing tensors where exact value differences matter. Cosine similarity measures the angle between two vectors rather than their magnitude, making it suitable for comparing tensors where only direction matters. It ranges from -1 to 1, where 1 means completely similar, 0 means orthogonal, and -1 means opposite.

Other Metrics for Tensor Comparison: **Mean Squared Error (MSE):** Commonly used in regression, it penalizes larger errors more than smaller ones, providing a squared average of differences. **Euclidean Distance (L2 Norm):** Measures straight-line distance between tensors, capturing absolute differences, and is useful in clustering and nearest-neighbor tasks. **Structural Similarity Index (SSIM):** Especially useful for images, as it takes into account luminance, contrast, and structural information, providing a more perceptual measure of similarity.

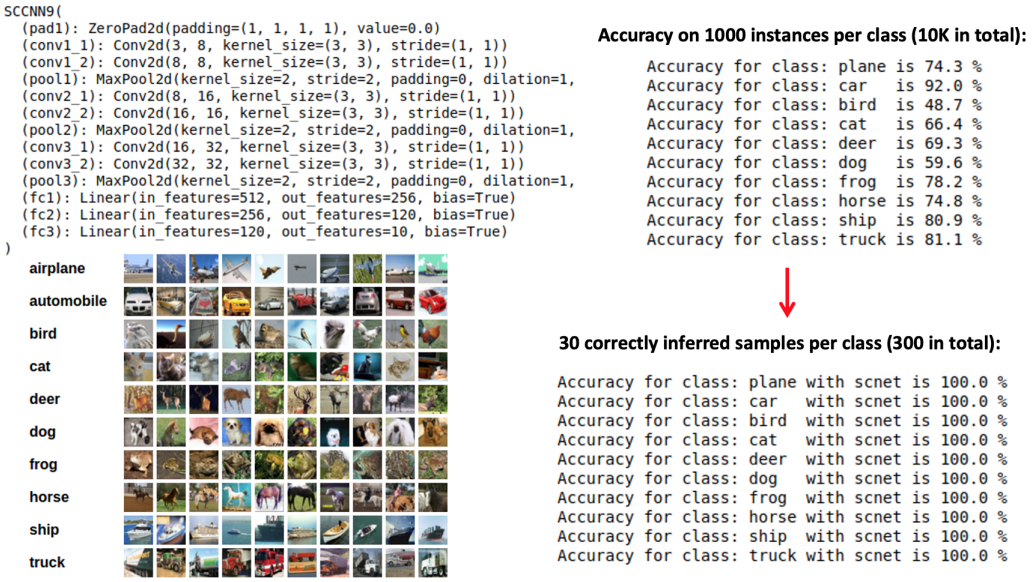


Figure 5.7: 30 correctly inferred images per class with normal VGG9 from CIFAR-10.

5.3 SC LeNet-5 Throughput & Inference Accuracy

The first graph to be discussed about the results of implementing a SC CNN LeNet-5 is Figure 5.8 which presents the MAE of the output tensors of each layer of the SC LeNet5 when compared to the original output tensors of each layer of the normal LeNet5.

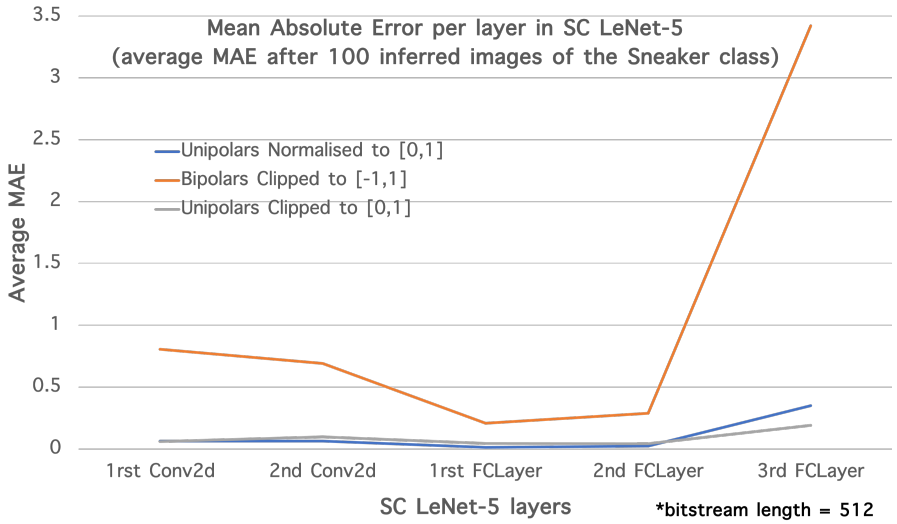


Figure 5.8: Mean Absolute Error per layer in SC LeNet-5 (average MAE after 100 inferred images of the Sneaker class)

There are three major variations that were implemented to the SC LeNet and that were measured and compared to the outputs of the original LeNet which are normalization of weights, biases and output tensors after ReLU to [0,1] with Unipolar SC functions, then instead of normalization it comes clipping of weights, biases and output tensors after ReLU to [0,1] and finally clipping of weights, biases and output tensors after ReLU to [-1,1] with bipolar functions. After inferring 100 images from the Sneaker class, the average MAEs of these 100 inferences were calculated and the results in the graph show the Unipolar functions are more precise than bipolar functions as they have less error in the outputs of each layer. It is more clear when seeing that the MAE of the final fully connected layer for the bipolar SC LeNet-5 spikes compared to the unipolar counterparts which remains relatively stable.

Then, the same is done so to analyse the results in terms of outputs similarity, where values closer to 1 indicate very similar output tensors. Figure 5.9 shows again the best results are for the unipolar SC LeNet and the bipolar functions present quite dissimilar outputs.

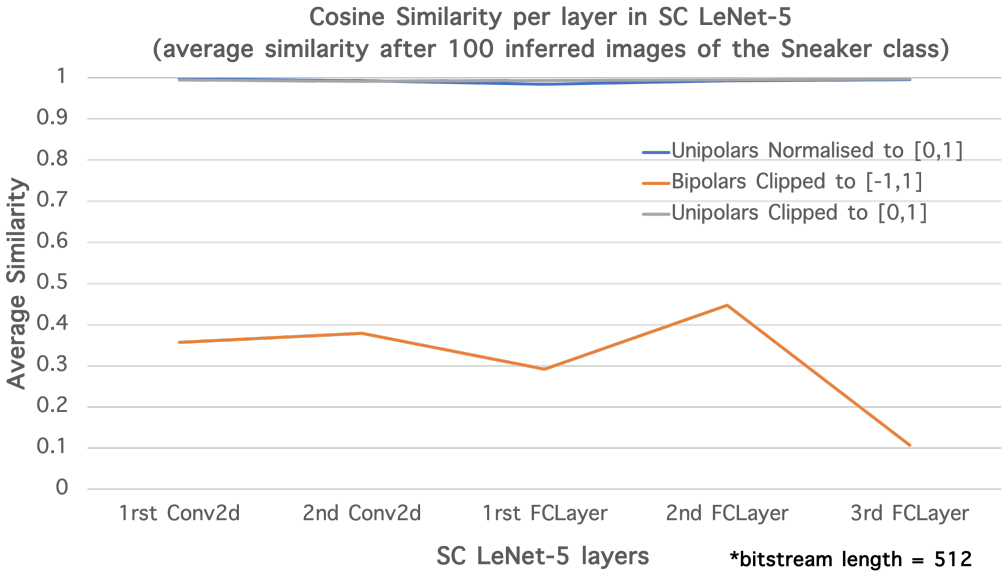


Figure 5.9: Average similarity after 100 inferred images of the Sneaker class.

In terms of execution time per layer, the unipolar SC LeNet5 also performs better than bipolar functions Figure 5.10 because of one particular reason which is that multiplication by zeros can be skipped in unipolar calculations because zeros are actually zeros, unlike bipolar bitstreams where zeros are represented in a bitstream by half amount of ones and other half of zeros; this form to represent bipolar zeros adds stochastic error inasmuch as zero bitstreams are not always perfectly divided by the same amount of ones and zeros -there can be more ones than zeros in a bitstream- so a zero in bipolar representation can be for example -0.056 or 0.233, which do not happen when using unipolar representation: zeros are always zeros as the range is [0,1] not [-1,1].
Since the 1rst fully connected layer possesses more parameters (considering it has flattened input values, weights and biases), it is the layer that takes more time to compute -almost 3.5 seconds in case of the unipolars and 4 seconds for the bipolars-, the final FClayer is the 'fastest' one to finish as there are only 84 input values and 10 output probabilities.

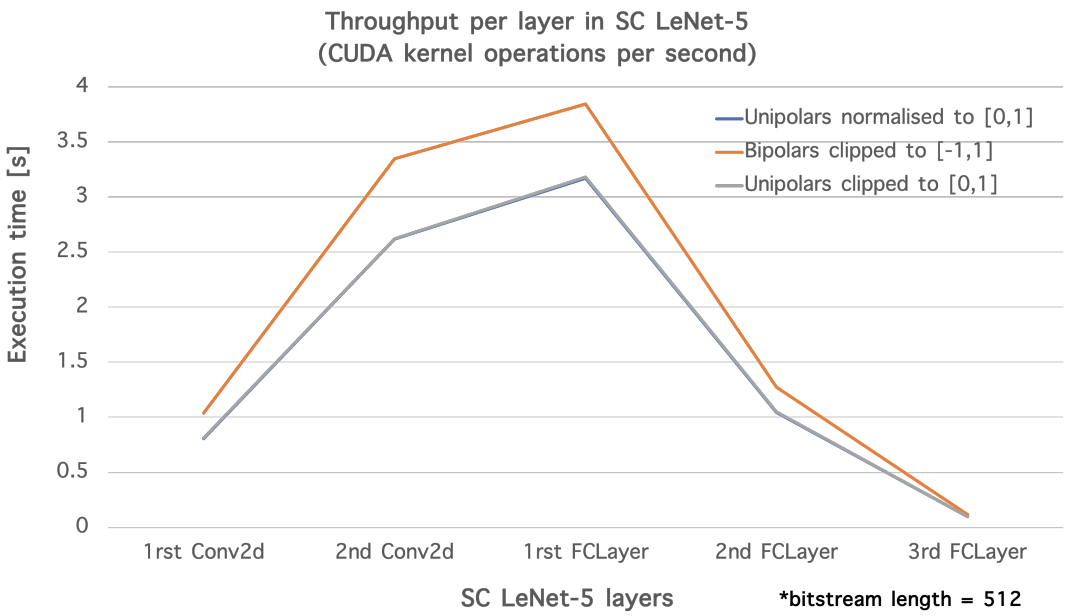


Figure 5.10: Execution time per layer in SC LeNet-5

The way the multiplications by zero are skipped in the unipolar functions can be revised in subsection 4.4 however there is a diagram in Figure 5.11 that can be reviewed so to observe how the CUDA Unipolar functions work in terms of skipping multiplications by zero, increasing the overall execution time and consequently SC CNN image inference throughput. Basically the first location of every stochastic bitstream generated by the Stochastic Tensor Generators is used to 'save' the sign either positive + or negative - of the incoming input values, whether they are weights, biases or pixels but if the input value is effectively zero, then a 0 is allocated into the first location of the bitstreams arrays. Then, in the SC 2D convolution or SC Fully Connected Layers the signs are checked and if they are equal, an accumulator of positive values is increased if the product of the multiplication is 1; but if the signs are different, then the accumulator that is increased is of negative values and finally the difference of the positive accumulations and negative accumulations is the output of the CUDA SC operations, however, what happens if the first location of the bitstream array is 0? The answer is nothing, neither positive nor negative accumulators are increased therefore skipping the multiplications (either XNOR or AND) and saving important thread computation time.

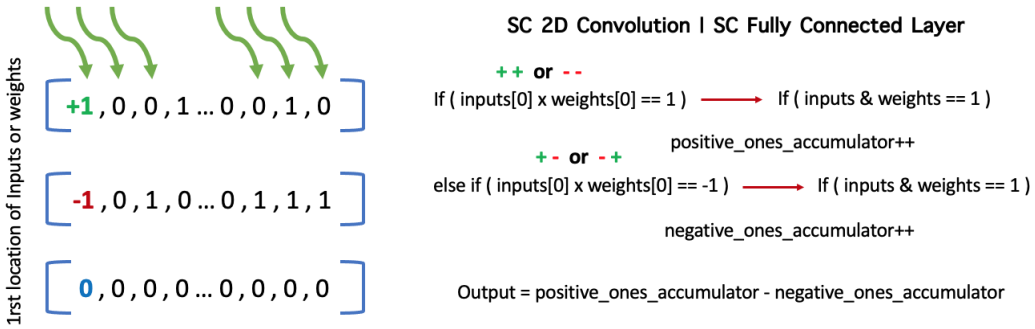


Figure 5.11: Unipolar functions overview.

Effectively in Figure 5.12 this time savings can be observed in a greater scale in terms of frames per minute -number of inferred images per minute-. With a bitstream length of 256, the Unipolar functions in SC LeNet showed better throughput than bipolars such that 16 frames are outputted by unipo SC network against 12 by bipolar SC network, evidently this difference would be more pronounced with shorter bitstream lengths.

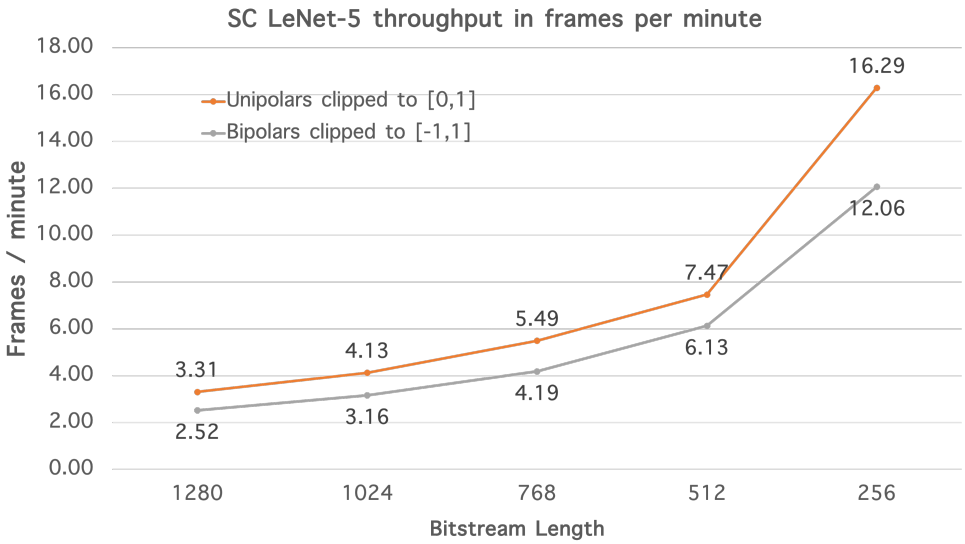


Figure 5.12: SC LeNet-5 throughput in frames per minute

Finally, in Figure 5.13 and Figure 5.14 is possible to see in fact the real gain obtained from the implementation of unipolar SC functions into the SC LeNet5 in terms of amount of inferred images per each of the fashionMNIST classes and in total after inferring a 1000 images, compared to the percentage of correctly inferred images by the forward propagation of the bipolar SC network. While Unipolar functions with a bitstream length of 512 do very good at inferring pretty much all the images passed to the forward prop, the bipolar forward prop can only manage to infer slightly over only half of the images, proving categorically that Unipolar numbers are far better in terms of accuracy when used in the forward propagation of SC CNNs.

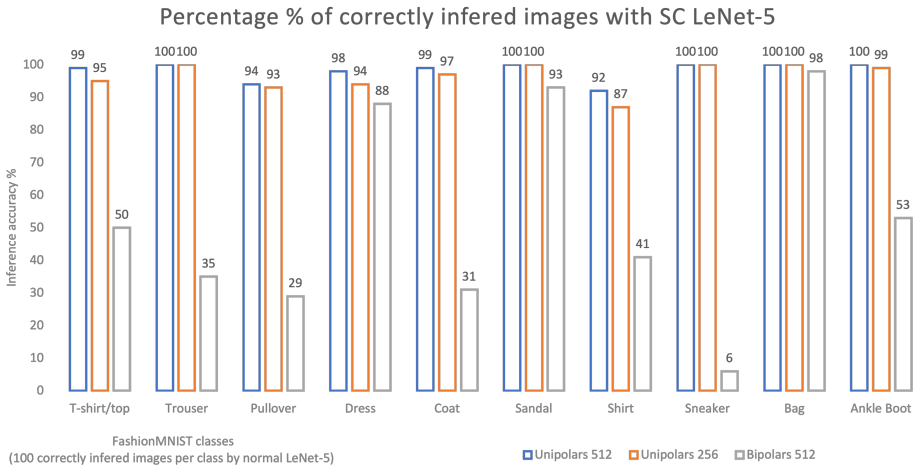


Figure 5.13: Percentage of 100 images inferred per class by the LeNet-5 SC CNN.

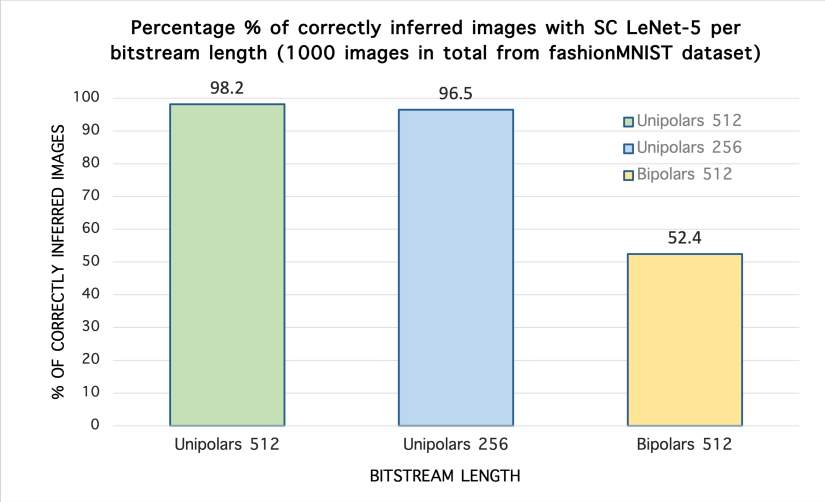


Figure 5.14: Percentage % of correctly inferred images with SC LeNet-5 per bitstream length (1000 images in total from fashionMNIST dataset)

5.4 SC VGG9 Throughput & Inference Accuracy

Nevertheless, to properly put in practice and measure the usefulness of the developed SC framework, it is necessary to implement the SC functions effectively in a Deep Convolutional Neural Network so to create a SC DCNN and observe the precision and accuracy of stochastic computing implemented in more layers. For this purpose, a deep convolutional neural network with 6 2D convolutions and 3 fully connected layers based on a VGG9[44] has been created in PyTorch and trained with clipped weights and biases to [0,1] in case of unipolar numbers and clipped to [-1,1] in case of bipolar numbers. Then these trained weights and biases are used in the SC version of the VGG9 -referred as SC VGG9-. The results of using both unipo and bipo representations can be seen in the following graphs. Figure 5.15 presents the number of correctly inferred images per class. 30 images previously inferred correctly by the normal VGG9 were taken and stored in a dictionary to be used then with the SC VGG9.

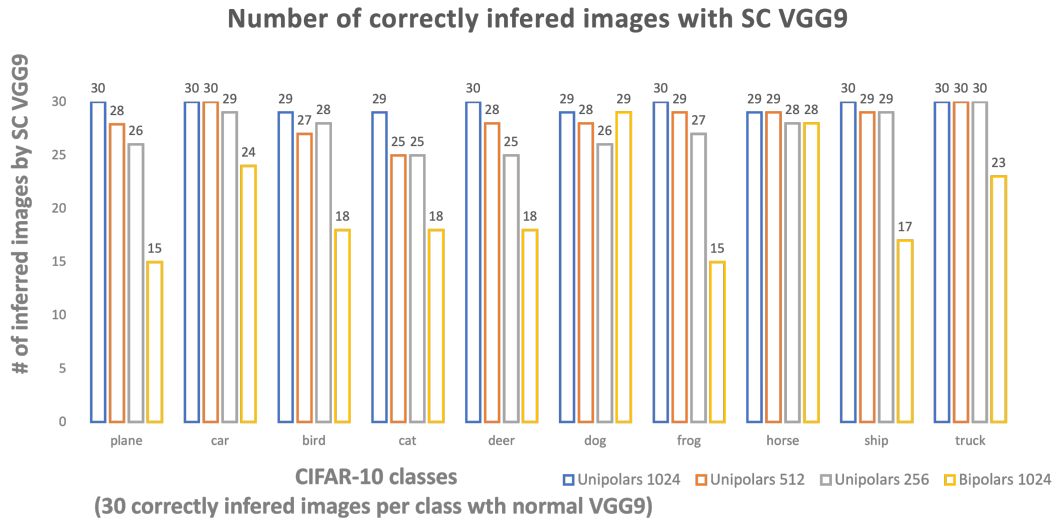


Figure 5.15: Number of correctly inferred images with SC VGG9

The previous showed Number of correctly inferred images with SC VGG9 per class, the following Figure 5.16 condenses the total amount of inferred images (300 in total) per bitstream length. Once again it is possible to observe there exists better accuracy in case of unipolars with a bitstream length of 1024, while bipolars with the same bitstream length cannot even do better that unipolars with an even reduced bitstream length of 256. Below then in Figure 5.16 the average inference time of a single image with SC VGG9 is shown revealing evidently that the best performance goes to the shorter bitstream and the worst is done by bipolars given that multiplications by zero cannot be skipped.

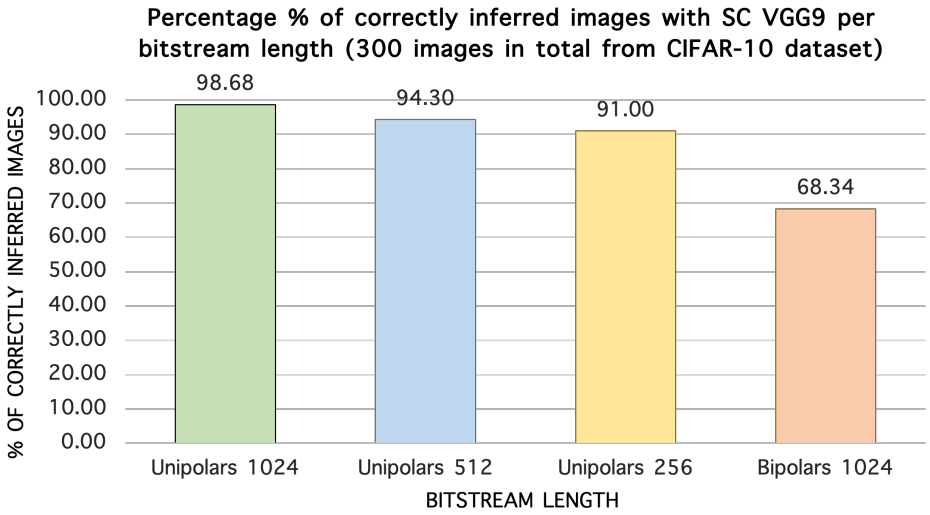


Figure 5.16: Percentage % of correctly inferred images with SC VGG9 per bitstream length (300 images in total from CIFAR-10 dataset)

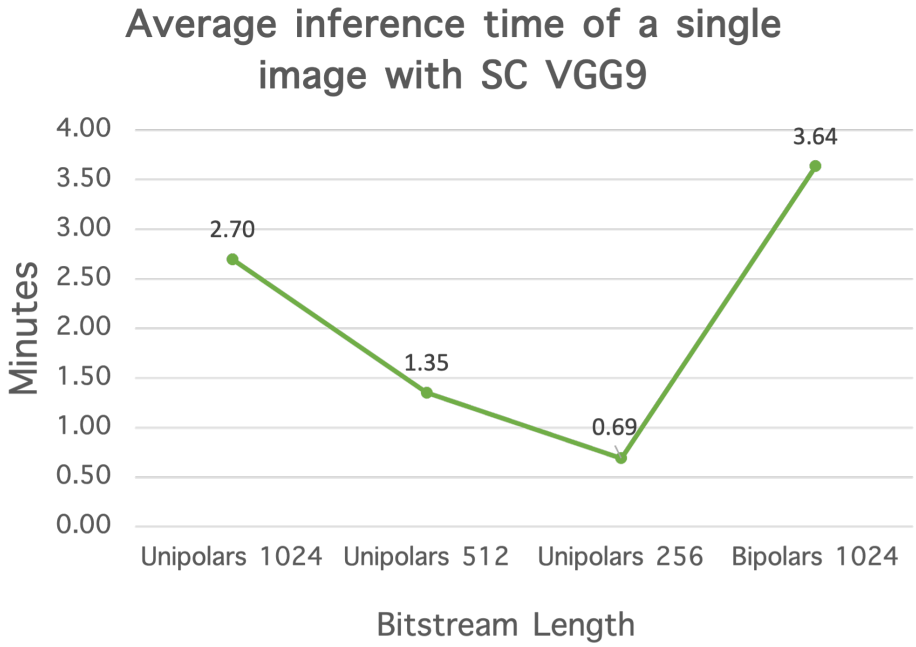


Figure 5.17: Average inference time of a single image with SC VGG9

Then, in Figure 5.18 it is possible to observe that in terms of overall throughput in frames per hour, the SC VGG9 is rather slow because the generation of matrices with random numbers used in the stochastic tensor generators is done by the CPU, not inside the CUDA kernel - NVIDIA provides a random number generator library called cuRAND [46] but with a limited amount of Generator Types mostly based on the Mersenne Twister family of pseudorandom number generators, and in order to properly compare the impact in inference precision of the type of RNG implemented to create the stochastic bitstreams, it is necessary to create the random number matrices from within the C++ CPU code as there are more RNG libraries available. Then this random number matrices can be copied to GPU device. Figure 5.19 presents the throughput in frames per hour per each class in CIFAR-10; the best throughput goes then to the shorter bitstream length (256) and the worst goes to the bipolar functions.

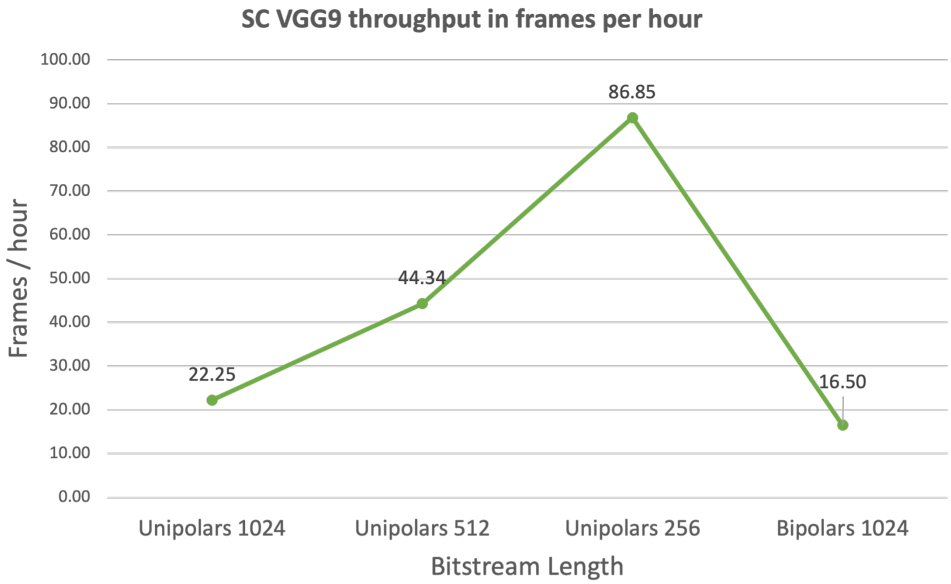


Figure 5.18: SC VGG9 throughput in frames per hour

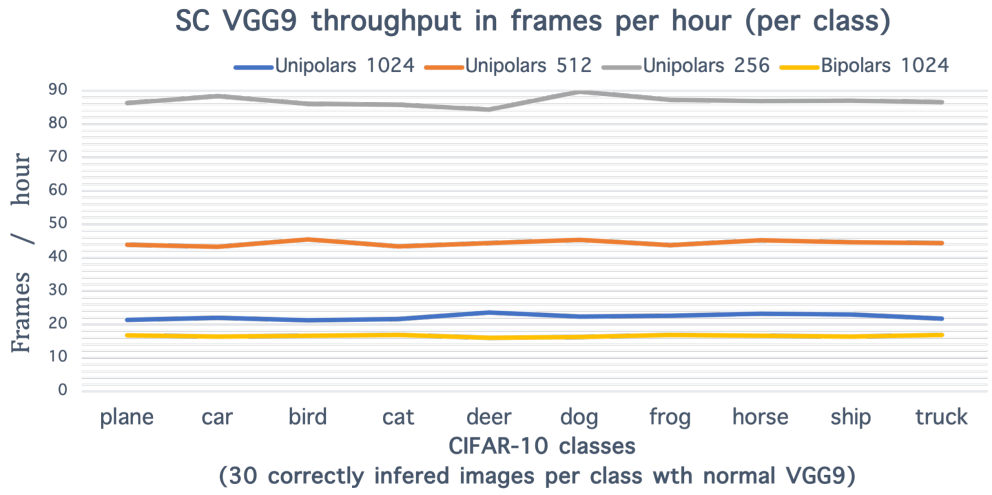


Figure 5.19: SC VGG9 throughput in frames per hour (per class)

5.5 SC VGG9 Inference Accuracy per RNG

Lastly, this section is the main reason why the random numbers matrices are created in CPU outside the CUDA kernel and the second reason why a PyTorch extension that would support Stochastic Computing is required (the first being to evaluate SC CNNs under different bitstream lengths with either unipolar or bipolar stochastic numbers). The overall precision that stochastic computing can achieve when implemented in DCNNs inference is mostly given and limited by the source of the random numbers: the type of RNG, therefore it is required to evaluate the accuracy of the forward propagation under different types of RNGs and then estimate whether there is any considerable accuracy improvement with one or another RNG type. To do this 4 different type of RNGs were tested with the SC PyTorch extension in the SC VGG9; these are the Mersenne Twister embedded in the C++ standard library [32], an allegedly new type of RNG called PCGs "...hard to predict." [47], an XORSHIFT* [48] and finally a 16 bit LFSR. In Figure 5.20 it is possible to observe the average inference execution time of 300 images from CIFAR-10 with SC VGG9 using each of these 4 different types of RNGs.

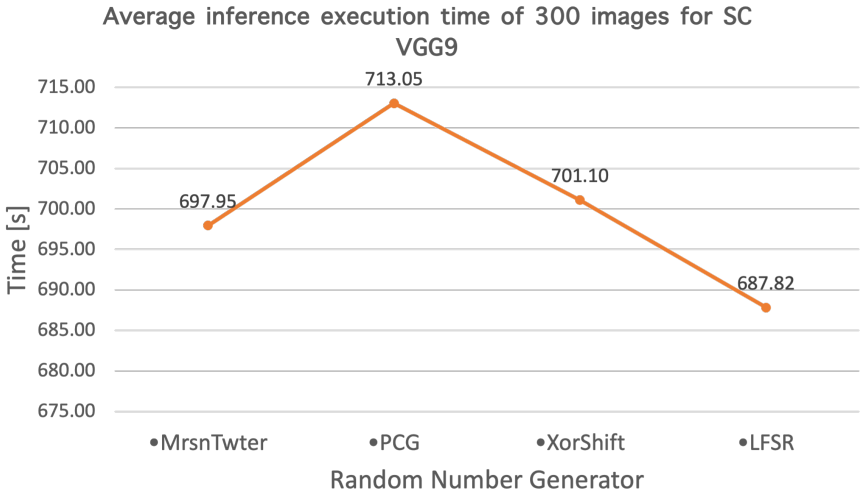


Figure 5.20: Average inference execution time of 300 images for SC VGG9

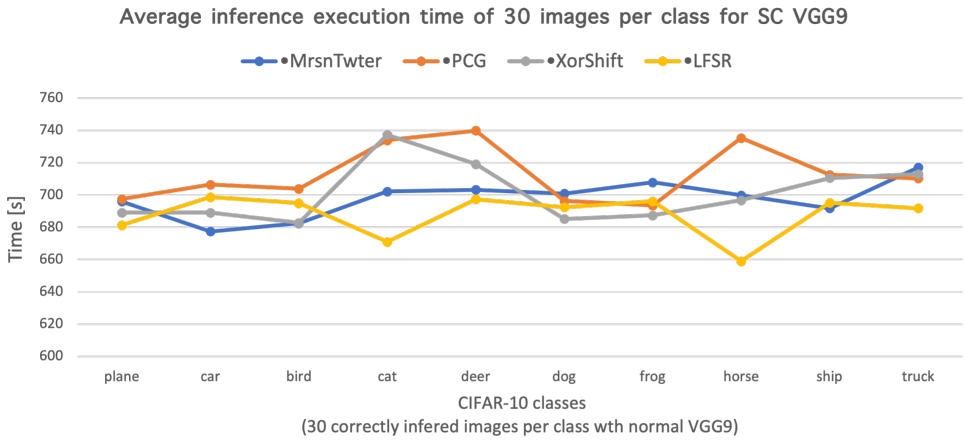


Figure 5.21: Average inference execution time of 30 images per class for SC VGG9

From Figure 5.20 and Figure 5.21 it is then possible to estimate that the 'fastest' RNG to infer 300 images of the CIFAR-10 dataset is the 16bit LFSR, and the slowest one is the so called PCG C++ library, however, as it will be seen next, the 16-bit LFSR presents a major disadvantage over these other 3 in terms of inference accuracy and output probabilities precision.

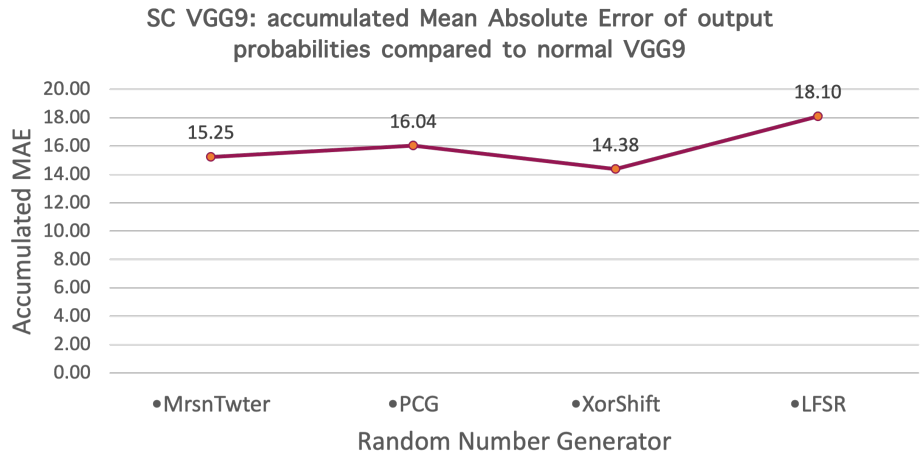


Figure 5.22: SC VGG9: accumulated Mean Absolute Error of output probabilities compared to normal VGG9

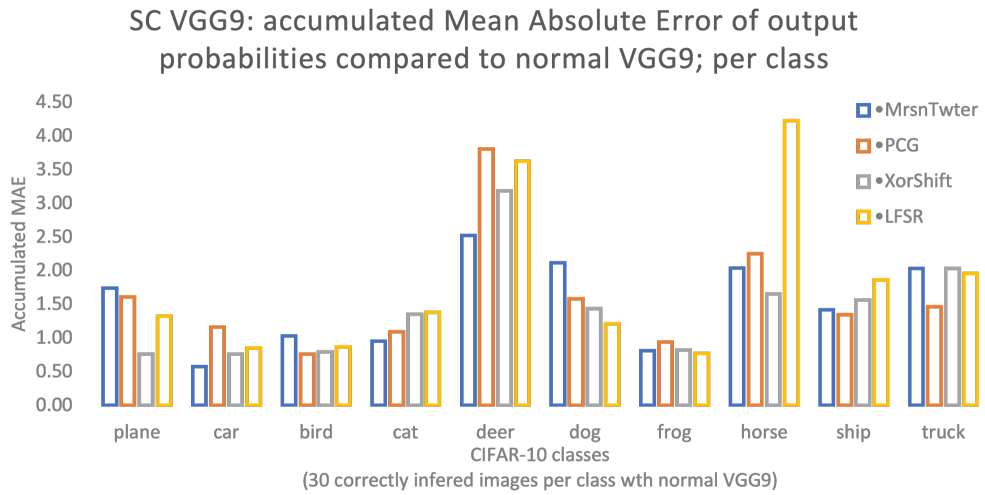


Figure 5.23: SC VGG9: accumulated Mean Absolute Error of output probabilities compared to normal VGG9; per class

Figure 5.22 and Figure 5.23 present the accumulated Mean Absolute Error of the output probabilities of the SC VGG9 when compared to the normal VGG9. It is possible to observe then that the worse RNG is the 16-bit LFSR and the one that appears to be slightly better than the others is XORSHIFT* [48], nonetheless these results are not really determinant enough so to conclude once and for all that XORSHIFT* is better than the others 3 for a larger dataset like ImageNet with a deeper SC DCNN like the VGG19 would be required to be implemented to assert this and maybe even test more types of software generated RNGs.

Finally, Figure 5.24 and Figure 5.25 show how each of these different RNGs inferred 300 images in total from CIFAR-10 dataset. Considering the previous analysis it is therefore expected to see that XORSHIFT* could in fact infer correctly more images than the other 3 RNGs, while LFSR had the lowest numbers and percentages.

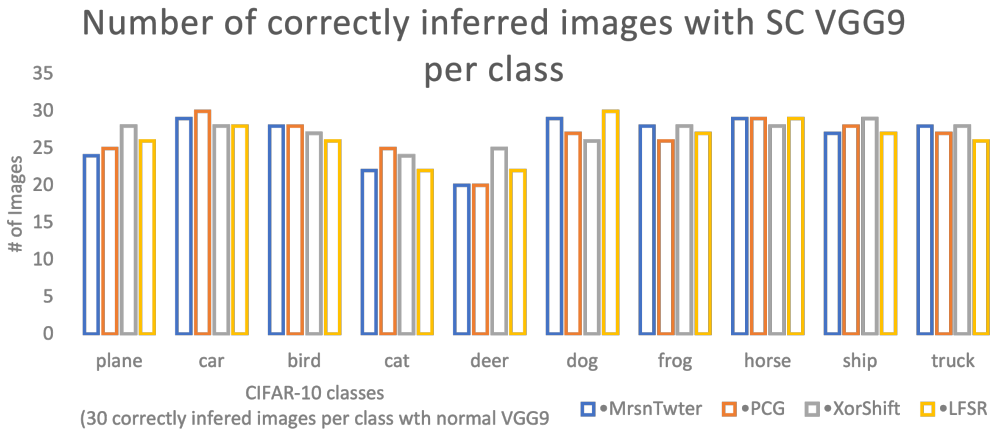


Figure 5.24: Number of correctly inferred images with SC VGG9 per class

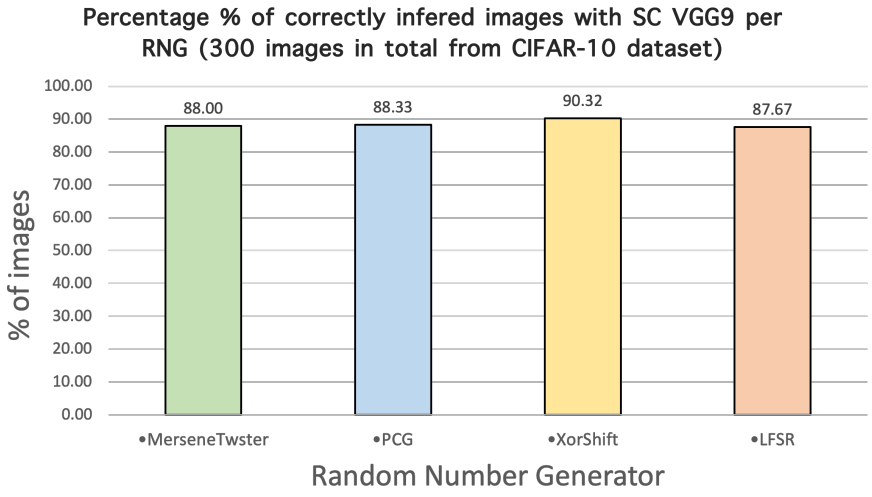


Figure 5.25: Percentage % of correctly infered images with SC VGG9 per RNG (300 images in total from CIFAR-10 dataset)

As it has already been commented previously, the purposes of this SC PyTorch framework extension could be extended to test more types of RNGs with different bitstream lengths and ideally very deep convolutional neural networks like VGG19, VGG16, GoogleNet or others, ideally with larger datasets. The overall results showed effectively that Unipolar representation in SC DCNNs inference is better in terms of accuracy, throughput and outputs precision than the bipolar representation.

6 Conclusions

Besides the already mentioned additional purposes that could be implemented using the SC PyTorch framework extension developed in this work, one important and quite interesting additional purpose could be found in the application of the SC functions to evaluate what would be the impact of implementing SC in the very recent transformer network used in large language models. If the results in terms of overall inference accuracy and outputs precision are good enough so to make SC worth implementing in transformer hardware designs, then the power and energy efficiency that could be gained would be substantial compared to traditional binary transformer implementations [49].
From the results obtained in this work, it is possible to assert that unipolar functions are more precise than bipolars because zeros are actually zeros and not approximations. In addition, shorter stochastic unipolar bitstreams are able to represent probabilities with more decimals, unlike bipolar numbers, which need larger bitstream lengths to represent less decimals and lastly, unipolar functions can compute more operations per second in the GPU implementation because it is possible to skip multiplications by zeros given that the accumulators do not need to be increased.
When comparing the 4 types of RNGs in here tested, it is reasonable to deduce that the 16 bit LFSR be the fastest RNG but also the less precise compared to the other 3 RNGs, whereas XorShift* was the RNG with less error and highest percentage of correctly inferred images.
In terms of performance, throughput is limited physically by the GPU device bandwidth and by the computation and memory latencies of the slowest thread, this means that the last thread to finish its computations in either SC 2D convolution or the SC FCLayer is the one that determines the end of the CUDA kernel execution. It is of course necessary to mention it would be possible to improve the performance and consequently general throughput of the SC functions, however for the purposes of this work it is deemed to say that a proper implementation of the extension was achieved with the development of a SC LeNet-5 and a SC VGG9.
There exist advancements in several other fields of DNN hardware acceleration such as Deep neural network quantization and systolic arrays which are two important techniques used to improve the efficiency of neural network computations, especially in hardware accelerators designed for deep learning. Quantization in deep neural networks (DNNs) involves reducing the precision of the numerical values (weights, activations, and biases) in a model. Typically, DNNs are trained with floating-point values (e.g., 32-bits), but quantization reduces these to lower-precision formats like 8-bit integers. This reduction in precision reduces computational and memory requirements with minimal impact on accuracy; on the other hand, a systolic array is a hardware architecture commonly used to accelerate matrix multiplications, a fundamental operation in deep learning. The systolic array is highly parallel, enabling data to flow rhythmically across processing elements (PEs) in a coordinated pattern, like a heartbeat (hence, "systolic"). Both of these approaches are great as far as traditional binary computations are concerned, nonetheless, stochastic computing could also carry even better advantages in terms of deep learning acceleration, especially in resource-constrained environments, because of the reduced Hardware complexity that simple AND and XNOR gates carry instead of complex arithmetic units, leading to much lower power consumption and hardware costs. Despite these benefits, SC in DCNNs comes with challenges, such as potentially slower processing due to large bitstream lengths. However, advances in stochastic computation techniques and architectures can make these trade-offs more favorable, especially as demand for efficient deep learning grows in edge and IoT applications. [50]

7 Future Work

There is still a lot that can be done and should be done regarding stochastic computing deep convolutional neural networks implementation and optimization in terms of overall inference accuracy and possibly even neural network training, as well as processing times and energy or power consumption of hardware implementations.
Specifically speaking in terms of the framework extension that in this work is presented, SC DCNNs with more layers like the VGG16 or VGG19 can be tested, but that would imply an optimization of the current CUDA parallel algorithms oriented towards an improved GPU memory management since the actual SC-VGG9 utilizes quite a lot of GPU memory, and as the number of parameters increases, not only does used memory increase too but also processing runtimes, and in consequence, throughput decreases proportionally.
In order to improve the current parallel implementations of the stochastic functions in terms of memory occupancy, splitting the fully connected layer data in batches can be tried, and regarding the 2D convolution, different already existent approaches like a Grouped convolution, Depth-wise, Point-wise, Depth Separable, etc., can also be tried so to try to decrease memory usage while maintaining or even improving the current processing times and throughputs.
Another rather interesting implementation of the current stochastic computing extension could be found in the implementation of a SC transformer network, however this would imply the necessary inclusion of yet more types of stochastic multiply-accumulate operations that are comprised within the transformer embedding, encoding and decoding layers -mainly the multi-headed self- and cross-attention and masked multi-headed attention sub layers-.
Finally, the current SC framework extension could be adapted to receive external random number matrices from other real random number generators found in system-on-chips so as to test faster whether these RNGs could potentially mean an improved precision of the neural network output probabilities or not, given that the main source of error in stochastic bitstreams generation comes from the type of RNG used. This would save a considerable amount of time in the research of new hardware developments as software is more flexible and therefore faster when it comes to testing new approaches or ideas regarding stochastic computing implementations.

8 Appendix

```
uint16_t LFSR_StatesGenerator(uint16_t state) {
    // XOR the bits according to the feedback polynomial
    // Polynomial: x^16 + x^15 + x^13 + x^4 + 1 => Tap mask: 0xD008
    bool bit = ((state >> 15) ^ (state >> 14) ^ (state >> 12) ^ (state >> 3)) & 1;
    state = (state << 1) | bit; // Shift and add the feedback bit
    return state;
}
```

Listing 8.1: 16-bit LFSR

```
    // Constructor with random weights and bias
ScFcLayer::ScFcLayer(int input_size, int output_size,
  const int bitstreamLength,  RandomNumberGenType type, BitstreamRepresentation mode)
    : input_size(input_size), output_size(output_size),
      bitstreamLength(bitstreamLength), type(type), mode(mode) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(-1.0, 1.0);

    weights.resize(input_size, std::vector<double>(output_size));
    for (int i = 0; i < input_size; ++i) {
        for (int j = 0; j < output_size; ++j) {
            weights[i][j] = dis(gen); } }
    bias.resize(output_size);
    for (int i = 0; i < output_size; ++i) {
        bias[i] = dis(gen); }

    scWeight = StochasticTensor(weights, bitstreamLength, type, mode);
    scBias = StochasticTensor({bias}, bitstreamLength, type, mode);
}
```

Listing 8.2: Sc Fc 2nd constructor

```
std::vector<uint8_t> concatenateSCVectors(const std::vector<uint8_t>& vector1,
    const std::vector<uint8_t>& vector2) {
    if (vector2.size() > 1 && std::any_of(vector2.begin(), vector2.end(),
        [](uint8_t i) { return i == 1; })) {
        std::vector<uint8_t> result = vector1;
        result.insert(result.end(), vector2.begin(), vector2.end());
        return result; }
    return vector1;
}

double calculatePx(const std::vector<uint8_t>& bstream1, BitstreamRepresentation mode,
    const std::vector<uint8_t>& bstream2) {
    std::vector<uint8_t> combined = bstream1;
    if ((!bstream2.empty()) && (bstream1.size() == bstream2.size())) {
        combined.insert(combined.end(), bstream2.begin(), bstream2.end()); }

    int totalOnes = 0;
    totalOnes = std::count(combined.begin(), combined.end(), 1);
    int totalLength = combined.size();

    double px = static_cast<double>(totalOnes) / totalLength;

    if (mode == UNIPOLAR) { return px;
    } else if (mode == BIPOLAR) {
        return (2 * px) - 1; } else {
        throw std::invalid_argument("Invalid mode"); }
}
```

Listing 8.3: ScFcLayer class auxiliary functions

```cpp
// // ScCudaTorch.cpp
#include <iostream>
#include <cstdio>
#include <random>
#include <cstdint>
#include <vector>

#include "pcg_random.hpp"
#include "xorshift.hpp"

#include <pybind11/pybind11.h>

// Function prototype
std::vector<std::vector<double>> ScCudaConv2d(
    const std::vector<std::vector<double>>& polarInputData,
    const std::vector<std::vector<double>>& polarKernelData,
    const std::vector<std::vector<double>>& randomMatrix_Input,
    const std::vector<std::vector<double>>& randomMatrix_Kernel);

std::vector<float> ScCudaFcLayer(
    const std::vector<float>& inputs,
    const std::vector<std::vector<float>>& weights,
    const std::vector<float>& biases,
    const std::vector<std::vector<float>>& randomMatrix_input,
    const std::vector<std::vector<float>>& randomMatrix_weights,
    const std::vector<std::vector<float>>& randomMatrix_biases,
    const int num_Outputs);

enum RandomGeneratorType {
    MT19937,
    LFSR_16,
    PCG,
    XORSHIFT };

class LFSR16 {
public:
    LFSR16(uint16_t seed) : state(seed) {}
    uint16_t next() {
        // Polynomial: x^16 + x^15 + x^13 + x^4 + 1 => Tap mask: 0xD008
        bool bit = ((state >> 15) ^ (state >> 14) ^ (state >> 12) ^ (state >> 3)) & 1;
        state = (state << 1) | bit;
        return state; }

    uint16_t operator()() {
        return next(); }

private:
    uint16_t state; };

std::vector<std::vector<double>> generate_random_matrix(int R, int C,
    RandomGeneratorType generator_type) {
    std::random_device rd;  // Random device for generating seeds
    std::vector<std::vector<double>> matrix(R, std::vector<double>(C));
    std::uniform_real_distribution<double> dis(0.0, 1.0);

switch (generator_type) {
        case MT19937: {
            for (int i = 0; i < R; ++i) {
                uint32_t seed = rd();  // Generate a new seed for each row
                std::mt19937 gen(seed);  // Mersenne Twister random number generator
                for (int j = 0; j < C; ++j) {
                    matrix[i][j] = dis(gen); } }
            break; }
            case LFSR_16: {
            for (int i = 0; i < R; ++i) {
                uint16_t seed = rd() & 0xFFFF;
                LFSR16 lfsr(seed);
                for (int j = 0; j < C; ++j) {
                    matrix[i][j] = dis(lfsr); } }
            break; }
```

Listing 8.4: Python-CUDA/C++ Binder Script part 1

```
        case PCG: {
           for (int i = 0; i < R; ++i) {
                pcg_extras::seed_seq_from<std::random_device> seed_source;
                pcg32 rng(seed_source);
                for (int j = 0; j < C; ++j) {
                    matrix[i][j] = dis(rng); } }
            break; }
        case XORSHIFT: {
            for (int i = 0; i < R; ++i) {
                uint32_t seed32 = rd();
                using rng32_type = xorshift_detail::xorshiftstar<xorshift32plain32a, uint16_t,
                    0xb2e1cb1dU>;
                rng32_type rng32(seed32);
                for (int j = 0; j < C; ++j) {
                    matrix[i][j] = dis(rng32); } }
            break; }
        default:
            std::cerr << "Invalid generator type" << std::endl; }
    return matrix;
}

PYBIND11_MODULE(sc_cuda_torch, m) {
    pybind11::enum_<RandomGeneratorType>(m, "RandomGeneratorType")
        .value("MT19937", RandomGeneratorType::MT19937)
        .value("LFSR_16", RandomGeneratorType::LFSR_16)
        .value("PCG", RandomGeneratorType::PCG)
        .value("XORSHIFT", RandomGeneratorType::XORSHIFT)
        .export_values();

    m.def("ScCudaFcLayer", &ScCudaFcLayer,
        "A function that does ScCudaFcLayer with weights, bias and input using CUDA");
    m.def("ScCudaConv2d", &ScCudaConv2d,
        "A function that does ScConv2D an input and kernel using CUDA");
    m.def("generate_random_matrix", &generate_random_matrix,
        "A function that generates_random_matrix with different types of RNG");
}
```

Listing 8.5: Python-CUDA/C++ Binder Script

```
def calculate_mean_absolute_error(y_true, y_pred):
    mae = torch.mean(torch.abs(y_pred - y_true))
    return mae

def tensor_similarity(tensor1, tensor2, threshold=0.90):
    assert tensor1.shape == tensor2.shape
    similarity = torch.nn.functional.cosine_similarity(tensor1.flatten(), tensor2.flatten(), dim=0)
    return similarity # Skibidi_Toilet == Sigma_Giga_Chad

def calculate_mean_percentage_error(y_true, y_pred):
    epsilon = 1e-8
    percentage_error = (y_true - y_true) / (y_true + epsilon)
    mpe = torch.mean(percentage_error)
    return mpe
```

Listing 8.6: SC CNN accuracy evaluation metrics

```
std::vector<float> ScCudaFcLayer(
    const std::vector<float>& inputs,
    const std::vector<std::vector<float>>& weights,
    const std::vector<float>& biases,
    const std::vector<std::vector<float>>& randomMatrix_input,
    const std::vector<std::vector<float>>& randomMatrix_weights,
    const std::vector<std::vector<float>>& randomMatrix_biases,
    const int num_Outputs){
    // Create CUDA streams
    cudaStream_t stream1, stream2, stream3;
    cudaStreamCreate(&stream1);
    cudaStreamCreate(&stream2);
    cudaStreamCreate(&stream3);
    std::vector<float> RM_flat = flatten(randomMatrix_input);
    ////STG__INPUT NEURONs__KERNEL Variables/////
    int inputData_size = inputs.size();
    int RM_cols = randomMatrix_input[0].size();
    int output_size = inputData_size * RM_cols;
    float* d_inputData;
    float* d_RM;
    int8_t* d_output;
    // Define grid and block dimensions
    int blockSize = 256;
    int numBlocks = (output_size + blockSize - 1) / blockSize;
    // Allocate input device memory
    cudaCheckError(cudaMalloc(&d_inputData, inputData_size * sizeof(float)));
    cudaCheckError(cudaMalloc(&d_RM, inputData_size * RM_cols * sizeof(float)));
    cudaCheckError(cudaMalloc(&d_output, output_size * sizeof(int8_t)));
    ////STG__WEIGHTS__KERNEL Variables//////
    std::vector<float> RM_flatW = flatten(randomMatrix_weights);
    std::vector<float> weights_data = flatten(weights);
    int inputData_sizeW = weights.size()*weights[0].size();
    int output_sizeW = inputData_sizeW * RM_cols;
    float* d_inputDataW;
    float* d_RMW;
    int8_t* d_outputW;
    // Define grid kernel and block kernel dimensions
    int blockSizeW = 256;
    int numBlocksW = (output_sizeW + blockSizeW - 1) / blockSizeW;
    // Allocate kernel device memory
    cudaCheckError(cudaMalloc(&d_inputDataW, inputData_sizeW * sizeof(float)));
    cudaCheckError(cudaMalloc(&d_RMW, inputData_sizeW * RM_cols * sizeof(float)));
    cudaCheckError(cudaMalloc(&d_outputW, output_sizeW * sizeof(int8_t)));
    ////STG__BIASES__KERNEL Variables//////
    std::vector<float> RM_flatB = flatten(randomMatrix_biases);
    float* d_inputDataB;
    float* d_RMB;
    int8_t* d_outputB;
    int inputData_sizeB = biases.size();
    int output_sizeB = inputData_sizeB * RM_cols;
    // Define grid kernel and block kernel dimensions
    int blockSizeB = 256;
    int numBlocksB = (output_sizeB + blockSizeB - 1) / blockSizeB;
    // Allocate kernel device memory
    cudaCheckError(cudaMalloc(&d_inputDataB, inputData_sizeB * sizeof(float)));
    cudaCheckError(cudaMalloc(&d_RMB, inputData_sizeB * RM_cols * sizeof(float)));
    cudaCheckError(cudaMalloc(&d_outputB, output_sizeB * sizeof(int8_t)));
    ///////OUTPUT SCFC__KERNEL//////
        // Number of output neurons
    int output_sizeFc =  num_Outputs;
        // Number of input neurons
    const int input_sizeFc = inputs.size();
    std::vector<float> h_output(output_sizeFc, 0);
    float* d_outputFc;
    cudaCheckError(cudaMalloc(&d_outputFc, output_sizeFc * sizeof(float)));
    int blockSizeFc = 256;
    int numBlocksFc = (output_sizeFc + blockSizeFc - 1) / blockSizeFc;
```

Listing 8.7: C++ kernel-wrapper function ScCudaFcLayer Part 1

```
                //time measure
                cudaEvent_t start, stop;
                cudaEventCreate(&start);
                cudaEventCreate(&stop);
                cudaEventRecord(start);
    // Copy INPUT data to device
    cudaCheckError(cudaMemcpyAsync(d_inputData, inputs.data(), inputData_size * sizeof(float),
        cudaMemcpyHostToDevice, stream1));
    cudaCheckError(cudaMemcpyAsync(d_RM, RM_flat.data(), inputData_size * RM_cols * sizeof(float),
        cudaMemcpyHostToDevice, stream1));
    // Copy WEIGHTS data to device
    cudaCheckError(cudaMemcpyAsync(d_inputDataW, weights_data.data(),inputData_sizeW*sizeof(float),
        cudaMemcpyHostToDevice, stream2));
    cudaCheckError(cudaMemcpyAsync(d_RMW, RM_flatW.data(), inputData_sizeW*RM_cols* sizeof(float),
        cudaMemcpyHostToDevice, stream2));
    // Copy BIAS data to device
    cudaCheckError(cudaMemcpyAsync(d_inputDataB, biases.data(), inputData_sizeB * sizeof(float),
        cudaMemcpyHostToDevice, stream3));
    cudaCheckError(cudaMemcpyAsync(d_RMB, RM_flatB.data(),inputData_sizeB*RM_cols* sizeof(float),
        cudaMemcpyHostToDevice, stream3));
    cudaMemcpyToSymbol(input_sizeFcL, &input_sizeFc, sizeof(int));
    cudaMemcpyToSymbol(outSizeFcL, &output_sizeFc, sizeof(int));
    cudaMemcpyToSymbol(N_fcl, &RM_cols, sizeof(int));
    ////STG__INPUT__KERNEL//////
    stochasticTensorGenerator<<<numBlocks, blockSize, 0, stream1>>>(d_inputData, d_RM, d_output,
        inputData_size, RM_cols);
    cudaCheckError(cudaGetLastError());
    ////STG__WEIGHTS__KERNEL//////
    stochasticTensorGenerator<<<numBlocksW, blockSizeW, 0, stream2>>>(d_inputDataW, d_RMW,
        d_outputW, inputData_sizeW, RM_cols);
    cudaCheckError(cudaGetLastError());
    ////STG__BIAS__KERNEL//////
    stochasticTensorGenerator<<<numBlocksB, blockSizeB, 0, stream3>>>(d_inputDataB, d_RMB,
        d_outputB, inputData_sizeB, RM_cols);
    cudaCheckError(cudaGetLastError());
    cudaStreamSynchronize(stream1);
    cudaStreamSynchronize(stream2);
    cudaStreamSynchronize(stream3);
    ////__SCFCLAYER__KERNEL//////
    forward_pass<<<numBlocksFc, blockSizeFc>>>(d_output, d_outputW, d_outputB, d_outputFc);
    cudaCheckError(cudaGetLastError());
    //cudaDeviceSynchronize();
    cudaCheckError(cudaMemcpy(h_output.data(), d_outputFc, h_output.size() * sizeof(float),
        cudaMemcpyDeviceToHost));
                //STOP time measure
                cudaEventRecord(stop);
                cudaEventSynchronize(stop);
                float milliseconds = 0;
                cudaEventElapsedTime(&milliseconds, start, stop);
    // Free device memory
    cudaFree(d_inputData);
    cudaFree(d_RM);
    cudaFree(d_output);
    cudaFree(d_inputDataW);
    cudaFree(d_RMW);
    cudaFree(d_outputW);
    cudaFree(d_inputDataB);
    cudaFree(d_RMB);
    cudaFree(d_outputB);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
    cudaFree(d_outputFc);
    return h_output;
}
```

Listing 8.8: C++ kernel-wrapper function ScCudaFcLayer Part 2

List of Figures

List of Tables

List of Listings

Acronyms

CNNs Convolutional Neural Networks 20

FCNNs Fully Connected Neural Networks 20

FP Floating Point 20

ILP Instruction-Level Parallelism 17

RNG Random Number Generator 20

SC Stochastic Computing 19

SC CNN Stochastic Computing Convolutional Neural Network 20

SC DCNN Stochastic Computing Deep Convolutional Neural Network 52

SC DCNNs Stochastic Computing Deep Convolutional Neural Networks 2

STG Stochastic Tensor Generator 27

TLP Thread-Level Parallelism 17

References

[1] Vincent C. Gaudet Warren J. Gross. *Stochastic Computing: Techniques and Applications*. Springer Nature Switzerland AG, 1. edition, 2019.

[2] Y. Liu, S. Liu, Y. Wang, F. Lombardi, and J. Han. A survey of stochastic computing neural networks for machine learning applications. *IEEE Transactions on Neural Networks and Learning Systems*, 32(7):2809–2824, 2021.

[3] A. Alaghi and J. P. Hayes. Survey of stochastic computing. *ACM Trans. Embed. Comput. Syst.*, 12, May 2013.

[4] Y. Wang Y. Liu and F. Lombardi. A stochastic computational multi-layer perceptron with backward propagation. *IEEE Trans. Comput.*, 67(9):1273–1286, Sep 2018.

[5] L. Liu S. Liu and J. Han. Gradient descent using stochastic circuits for efficient training of learning machines. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 37(11):2530–2541, Nov. 2018.

[6] Amos R Omondi. *FPGA implementations of neural networks*. Springer, 2006.

[7] W. Luk C. Luo and C. Guo. Towards efficient deep neural network training by fpga-based batch-level parallelism. *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 45–52, 2019.

[8] Yudong Tao, Rui Ma, Mei-Ling Shyu, and Shu-Ching Chen. Challenges in energy-efficient deep neural network training with fpga. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, pages 400–401, 2020.

[9] Joonsang Yu, Kyounghoon Kim, Jongeun Lee, and Kiyoung Choi. Accurate and efficient stochastic computing hardware for convolutional neural networks. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 105–112, 2017.

[10] MathWorks. What is a convolutional neural network?: 3 things you need to know. `https://www.mathworks.com/discovery/convolutional-neural-network.html?s_tid=srchtitle_site_search_1_convolutional%2520`, Jul 2024.

[11] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[12] GeeksforGeeks. Weight initialization techniques for deep neural networks. `https://www.geeksforgeeks.org/weight-initialization-techniques-for-deep-neural-networks/`, Jul 2022.

[13] What are Convolutional Neural Networks? — IBM — ibm.com. `https://www.ibm.com/topics/convolutional-neural-networks`. [Accessed 03-09-2024].

[14] Cross-correlation – from Wolfram MathWorld — mathworld.wolfram.com. `https://mathworld.wolfram.com/Cross-Correlation.html`. [Accessed 05-09-2024].

[15] Cross-correlation - wikipedia — en.wikipedia.org. `https://en.wikipedia.org/wiki/Cross-correlation`. [Accessed 05-09-2024].

[16] Conv2d 2014; pytorch 2.4 documentation - pytorch.org. https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html#torch.nn.Conv2d. [Accessed 05-09-2024].

[17] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2018.

[18] Maxpool2d 2014; pytorch 2.4 documentation — pytorch.org. https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html#torch.nn.MaxPool2d. [Accessed 05-09-2024].

[19] Convolutional neural network - wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Convolutional_neural_network. [Accessed 05-09-2024].

[20] Linear x2014; pytorch 2.4 documentation — pytorch.org. https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear. [Accessed 07-09-2024].

[21] Diego Unzueta. Fully connected layer vs convolutional layer: Explained — built in — builtin.com. https://builtin.com/machine-learning/fully-connected-layer. [Accessed 07-09-2024].

[22] Michael A. Nielsen. *Neural networks and deep learning*. Determination Press, 2015.

[23] Tom M. Mitchell. *Machine learning*. McGraw Hill, 1997.

[24] Cs 230 - convolutional neural networks cheatsheet — stanford.edu. https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks#hyperparameters. [Accessed 09-09-2024].

[25] Introduction to cuda programming and performance optimization — gtc 24 2024 — nvidia on-demand — nvidia.com. https://www.nvidia.com/en-us/on-demand/session/gtc24-s62191/. [Accessed 11-09-2024].

[26] NVIDIA Corporation. *CUDA C++ Programming Guide*. NVIDIA Corporation, 2024.

[27] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[28] Jason Sanders, Edward Kandrot, and Jack J. Dongarra. *Cuda by example: An introduction to general-purpose GPU programming*. Addison-Wesley/Pearson Education, 2015.

[29] Cornell virtual workshop; understanding gpu architecture > gpu memory > memory levels — cvw.cac.cornell.edu. https://cvw.cac.cornell.edu/gpu-architecture/gpu-memory/memory_levels. [Accessed 14-09-2024].

[30] Vasily Volkov. *Understanding latency hiding on gpus*. PhD thesis, University of California, Berkeley, 2016.

[31] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into deep learning*. Cambridge University Press, 2024.

[32] Mersenne twister - wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Mersenne_Twister. [Accessed 19-09-2024].

[33] Sangyoon Lee. https://www.evl.uic.edu/sjames/cs525/final.html.

[34] krunal1313. 2d-convolution-cuda/convolution.cu at master · krunal1313/2d-convolution-cuda. https://github.com/krunal1313/2d-Convolution-CUDA/blob/master/convolution.cu.

[35] CoffeeBeforeArch. 2d_constant_memory. https://github.com/CoffeeBeforeArch/cuda_programming/blob/master/05_convolution/2d_constant_memory/convolution.cu.

[36] https://pybind11.readthedocs.io/en/stable/index.html.

[37] Peter Goldsborough. Custom c++ and cuda extensions. https://pytorch.org/tutorials/advanced/cpp_extension.html.

[38] https://pytorch.org/tutorials/beginner/introyt/introyt1_tutorial.html.

[39] https://pytorch.org/tutorials/beginner/introyt/trainingyt.html.

[40] Pradeep Gupta. Cuda refresher: The cuda programming model. https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/, Jun 2023.

[41] OneFlow. How to choose the grid size and block size for a cuda kernel? https://oneflow2020.medium.com/how-to-choose-the-grid-size-and-block-size-for-a-cuda-kernel-d1ff1f0a7f92, Jan 2022.

[42] Nvidia nsight compute. https://developer.nvidia.com/nsight-compute.

[43] Fang Cabrera. The cuda parallel programming model - 5. memory coalescing - fang's notebook. https://nichijou.co/cuda5-coalesce/.

[44] Hongyi Wang, Mikhail Yurochkin, Yuekai Sun, Dimitris Papailiopoulos, and Yasaman Khazaeni. Federated learning with matched averaging, 2020.

[45] https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html.

[46] NVIDIA. https://docs.nvidia.com/cuda/curand/introduction.html#introduction.

[47] M.E. O'Neill. Pcg, a family of better random number generators. https://www.pcg-random.org/, Aug 2014.

[48] M.E. O'Neill. A family of truncated xorshift* prngs. https://gist.github.com/imneme/9b769cefccac1f2bd728596da3a856dd.

[49] Beom Jin Kang, Hae In Lee, Seok Kyu Yoon, Young Chan Kim, Sang Beom Jeong, Seong Jun O, and Hyun Kim. A survey of fpga and asic designs for transformer inference acceleration and optimization. *Journal of Systems Architecture*, 155:103247, 2024.

[50] Yang Yang Lee and Zaini Abdul Halim. Stochastic computing in convolutional neural network implementation: A review, Nov 2020.

Plagiarism Scan Result Report

Scanned Text

Input File	*template_thesis_students_javier.pdf*
Author	*n/a*
Scan Date	*2024-11-05*
Project	*n/a*
Remarks	*n/a*

Result Overview

Copied/Total Words: 1729/26358
Suspicious Sources: 35
Accepted Sources: 0
Excluded Sources: 0

Percentage of Copied Words / Total Words

Layout of copied paragraphs

Legend and Explanations

Original Text	Text without any identified relevant online or library sources.
Suspicious source	Text containing identified matches in online or library sources. The text might have been rephrased.
Accepted source	Text with correctly quoted and accepted citations in online or library sources. The text may have been rephrased.

Searched Sources and Settings

Maximum phrase distance	125	Searched in online sources	Yes
Minimum number of words per source	50	Searched in library sources	Yes
Minimum characters per phrase	15	Minimal similarity	35%