Benchmarking Spiking Neural Networks with Multiprocessing

Isaac Bettendorf, *Student, Virgina Tech,* Osaze Shears, *Student, Virginia Tech*

I. Introduction

THE brain is a massively asynchronous and parallel computational machine [1], [2], [3]. While modern computers become increasingly faster at performing tasks that are easily accomplished with sequential arithmetic and logic operations, they still fall short in performing tasks such as image recognition and abstract thinking with the speed and power efficiency of the human brain [2], [3].

A. Evolution of Modern Computer Architecture

Modern computer architecture evolved from the Von Neumann architecture and their designs have carried with them a bottleneck in the system bus which makes a multiprocessor design very unscalable. One method to increase throughput despite this shortcoming is to utilize specialized processing hardware such as graphic processing units (GPU). A GPU is composed of many specialized arithmetic processing units (ALUs) used for matrix multiplication and can typically perform hundreds of such calculations simultaneously.

B. Artificial Neural Networks

Artificial neural networks (ANN) are a group of algorithms designed to mimic the behavior of biological neural networks, nodes (neuron models) and their connections (synapses) being the ANN's most basic elements. These ANNs are taught to perform tasks by training. During training, inputs, whose expected outputs are known, are fed into the ANN. These inputs propagate through the system producing outputs which may or may not be correct. This process is known as *forward propagation*. The ANN then performs a process known as *back propagation* in which parameters of this network are adjusted to compensate for the difference between the actual output and the desired output. Traditionally in an ANN, it is the forward propagation which utilizes matrix multiplication, a process that can be sped up by using a GPU. More specialized neuromorphic hardware exists where each neuron model is represented in hardware. Neuromorphic hardware is more scalable for power and integrated circuit (IC) real estate compared to the GPU, which suffers from the unscalability of multiplier circuits [4][5]. Neuromorphic hardware often simulates the behavior of spiking neural networks.

C. Spiking Neural Networks

Spiking neural networks (SNN) are a form of ANN that more closely simulates the behavior of natural neural networks [6]. They are inherently asynchronous because SNNs do not have optimizers like traditional ANN. Optimization in ANNs traditionally needs to be synchronous: one synapse's optimizations will have an affect on another synapse's optimization. To contrast, SNN synapses simply wait to receive spikes from nodes within a simulation time step. The state of one synapses is not necessarily influenced by the state of other synapses in a network. A synapse in an SNN is only focused on the nodes whose connections are coming into it; however, there is a synchronous aspect to SNNs in that its elements must wait to continue operating until all elements have reached the end of the time step [4].

SNNs are massively parallel because of the asynchronous nature of the individual nodes. In each layer, a node can potentially be simulated in its own thread. In this article's implementation of a multithreaded SNN, there are groups of nodes (of the same layer) being simulated by a single thread; however, studying the performance on this architecture may provide insight for the development of future architectures.

D. Project Overview

This article is inspired by the work seen in [7]. Ma et al. study the performance of a multithreaded stochastic gradient descent (SGD) training algorithm on a distributed, non-uniform memory access (NUMA) architecture. In our project we add multithreading to an SNN algorithm and benchmark its performance against a multithreaded convolutional neural network (CNN) when learning images in the MNIST dataset.

II. Background

A. Spiking Neural Network Mechanics

Spiking neural networks (SNNs) are fundamentally different from ANNs because their operation occurs over a period of time. At each time step, the neurons in an SNN receive weighted spikes that either increment or decrement the neuron's membrane potential. Once the membrane potential of a neuron exceeds a predetermined threshold value, the neuron emits a spike. After emitting a spike, the neuron resets its membrane potential to its initial value and stops the membrane potential from changing for a period of time called the "refractory period".

There are three components needed to simulate an SNN: (1) an encoding scheme, (2) a neuron model, and (3) a learning rule.

1) Encoding Schemes: Encoding schemes are necessary to specify the way that numerical input data is converted into spike trains. Typically each feature of the input data is mapped to an input neuron which generates spikes based on

the feature's intensity. Our project looks at two different styles of encoding data: rate coding and temporal coding.

Rate coding works by adjusting the *rate* at which input neurons fire based on the intensity of the input data. Input neurons mapped to higher intensities fire more frequently, while input neurons mapped to lower intensities fire less. While rate coding is a noise-resistant way to represent data, it causes the network to consume more power when implemented in hardware.

Temporal coding works by adjusting the *time* at which input neurons fire based on the intensity of the input data. Input neurons mapped to higher intensities fire earlier compared to inputs with lower intensities. Temporal coding is more power efficient than rate coding because of its sparsity, however, it is very susceptible to noise [8].

2) Neuron Models: Another component required for simulating SNNs is the neuron model. Similar to the activation function in an ANN, the neuron model uses the inputs provided at the synapses to determine the output value of the neuron. The primary difference between the two is that a neuron model in an SNN will not provide any output if its membrane potential has not reached a certain threshold.

In our project, we conducted experiments using the **leaky integrate-and-fire (LIF)** neuron model. In this model, weighted spikes that arrive at the neuron increase the membrane potential normally. The membrane potential also decays towards a resting value at each timestep. This behavior prevents neurons that are not frequently stimulated from firing [9].

3) Spike-Timing Dependent Plasticity: **Spike-timing dependent plasticity (STDP)** is a common unsupervised learning rule used to train SNNs. In this learning technique, the influence that a pre-synaptic neuron has on a post-synaptic neuron is adjusted based on the time elapsed between their emitted spikes. If a pre-synaptic neuron fires just before a post-synaptic neuron, then the weight of the synapse connecting those two neurons is incremented (i.e., the pre-synaptic neuron firing had a strong correlation to the post-synaptic neuron firing) [10]. STDP is an inherently asynchronous algorithm since each synapse performs its weight updates independently from the other synapses.

B. Convolutional Neural Network

Convolutional neural networks (CNN) are traditional ANNs with added functionality given by specialized layers with the most important of these special layers being the convolutional. The convolutional layer's function is to learn and form filters that best assist in the classification process. CNNs that are composed of 2 dimensional convolutional layers are ideal for image processing.

C. Computer Architecture

This section presents the two main architectures used in this work. The first is Virginia Tech's RLogin which is a NUMA based architecture consisting of 64 Intel Xeon(R) Gold 5218 (at 2.30GHz) CPUs. The second, Virginia Tech's GPU cluster, was composed of one Nvidia Tesla T4 (rev a1) GPU and eight Intel Xeon(R) E5-2630 v3 (at 2.40 GHz) CPUs.

TABLE I
HARDWARE SPECIFICATIONS

Element Name	CPU NUMA	GPU + CPU NUMA
CPU/MP	64	1 GPU + 8 CPU
Cores	16 perCPU	2560CUDA (totalGPU) + 8perCPU
Blocks	–	4 Per MP
Threads	32 perCPU	2560
RAM/global	383 GB	16 GB for GPU + 32 GB for CPU

Table I depicts the architectures' further specifications. Each of the NUMA nodes are connected by high-bandwidth interconnects and DRAM access. Data transfer between the nodes takes longer than locally accessed memory.

The GPU is composed of multiple streaming multiprocessors (MP). Each MP is driven by the CUDA programming model. Thousands of threads can potentially run simultaneously on the GPU. But the work passed to the GPU is specialized and done so in a manner similar to passing a function from the CPU [11]. The passed information consists of a logical thread identifier that allows the threads to work on different sections of the data [11].

D. Related Work

1) Stochastic Gradient Descent on Hardware: In their research, Ma et al. (2019) benchmarked two different styles of multithreaded stochastic gradient descent (SGD) on both CPU and GPU hardware [7]. Their synchronous SGD algorithm allowed the threads to concurrently compute the gradient based on the training samples, and required threads to synchronize after this calculation so that the main thread could perform the weight updates. Alternatively, their asynchronous SGD allowed the threads to both concurrently compute the gradient based on their training samples and update the weights accordingly.

The group found that when synchronous SGD was run using a GPU it sped up the training time by approximately 7X for deep neural networks when compared to the CPU implementation. Additionally, asynchronous SGD performed on a multithreaded CPU outperformed the GPU version by more than 10X.

2) BindsNET: BindsNET is an SNN simulation framework developed in Python by Hazan et al. [12]. It provides users with an efficient way to construct and simulate SNNs in a way the is consistent with PyTorch's tensor objects.

Since BindsNET is built on top of PyTorch, it also supports running the SNNs tensor operations on an NVIDIA GPU using CUDA. BindsNET also provides several neural models and encoding schemes to construct a variety of different SNNs.

3) Fast Spiking Neural Network Architecture for Low-cost FPGA Devices: The source [13] implements an SNN on a Xilinx Spartan 3 FPGA. Modeling a real biological system requires thousands of neurons. This sources proposes an SNN architecture that is able to be implemented using relatively

little hardware resources but still represent biological networks. This was accomplished by a combination of both serial and parallel structures that are used to optimize the neurons' required computation time. The presented results illustrates an architecture whose performance is comparable to similar implementations but reduces resource consumption by 70%.

III. Experimental Design

In order to capture any improvements provided by the addition of multithreading to an SNN implementation, an SNN architecture that classifies the MNIST dataset was chosen. This architecture consists of four layers whose dimensions are respectively 784x320x50x10 LIF neuron models.

For comparison, a CNN was utilized. Just as with the SNN, the CNN's architecture was chosen based on its ability to classify the MNIST dataset. This architecture consisted of six layers: two convolutional (size 10x20), a drop-out, and three traditional ANN layers (320x50x10).

The architectures were contracted using PyTorch in Python 3.6. The SNN also utilizes the BindsNET framework for simulation. The experiments measured throughput which is defined as the number of forward and back propagations performed per second.

A. Multithreaded Convolutional Neural Network Algorithm

Many different methods for multithreading the CNN were experimented with before using the final architecture and method. PyTorch methods such as `DataParallel` and `DistributedDataParallel` on different NUMA architectures such as Google CoLab were experimented with before settling on the Python method `process` and Virginia Tech's RLogin to take the final data points.

Algorithm 1 Multithreaded CNN Pseudocode

for ii **in** range(num_processes) **do**
  Process(target=train,args=(ii,args,model,dataset))
  pr.start()
  prArray.append(pr)
end for
for pr **in** prArray **do**
  pr.join()
end for

B. Multithreaded Spiking Neural Network Algorithm

Before attempting to add parallelism to the SNN, we first used cProfile to determine where the the algorithm was spending most of its execution time. **Figure 1** shows the results of running the 320x50x10 SNN in cProfile. The results indicate that that 78% of the execution time is spent performing updates at the network's layers (40%), synapses (22%) and weights (16%). For each of these tasks, the network spent the most time performing matrix multiplication.

Given these results, we focused on modifying the algorithm so that the layer, synapse and weight updates could be performed in parallel using multiple threads. The initial approach

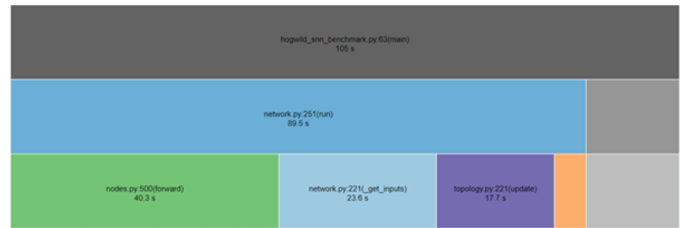

Fig. 1. Results from running cProfile on a single-threaded SNN implementation.

taken utilized Python's `Thread` and `Queue` classes to provide tasks for the worker threads to perform. The throughput results collected in our project were taken using these classes. Later in the project we transitions to using `ThreadPool` to improve the execution of our multithreaded algorithm. This implementation is shown in **Algorithm 2**.

Algorithm 2 Multithreaded SNN Pseudocode

for $t = 0$ **to** num_timesteps **do**
  for $c = 0$ **to** num_connections **do**
    ThreadPool.apply(connection[c].get_inputs())
  end for
  for $l = 0$ **to** num_layers **do**
    ThreadPool.apply(layer[l].forward())
  end for
  for $c = 0$ **to** num_connections **do**
    ThreadPool.apply(connection[c].update_weights())
  end for
end for

C. Experimental Procedure

A Python virtual environment was created on both of Virginia Tech's CPU and GPU clusters and the CNN and SNN were trained using the MNIST dataset. On the CPU cluster, both the CNN and the SNN were executed with varying thread counts from 1 to 40 in steps of 4. This was repeated for each given batch size of 32, 64 and 128. This processes was repeated for the SNN given the encoding schemes of rate coding and temporal coding. Because of the limited number of CPUs, the same process above was executed on the GPU cluster, however, the thread count varied from 1 to 16 and the step size was either 1 or 2.

IV. Results

A. CNN Results

Figure 2 and **Figure 3** illustrate the results collected from the multithreaded CNN executed on both CPU and GPU systems respectively. While the multithreading did provide improvements, it fell short of utilizing the architecture's full potential. In **Figure 2**, it can be seen that for all three executions, the peak throughput is reached at about 20 threads. After, as thread count increases, the throughput began to decrease. The greatest throughput measured is when the batch size is 128 at 507 ops/s. For this batch size, the throughput at one thread is 252 ops/s. This translates to a speedup of

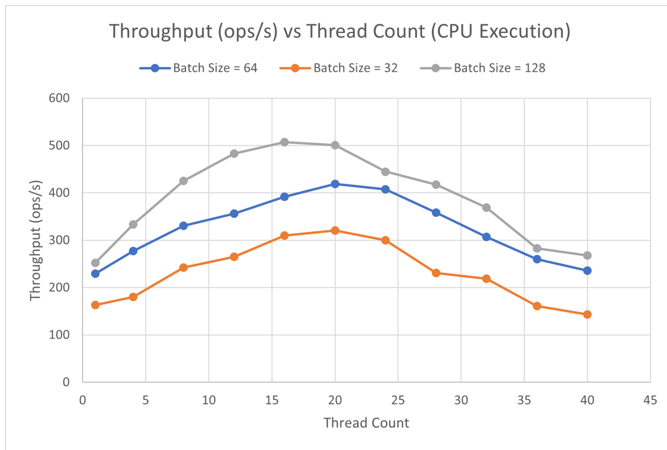

Fig. 2. Results from running the CNN Variations on the CPU

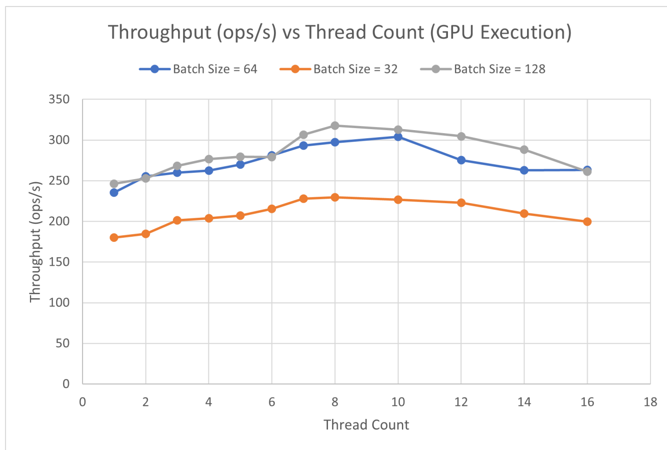

Fig. 3. Results from running the CNN Variations on the GPU

about 2. Using Amdahl's law, it is calculated that, at this CNN's implementations best, the percentage of instructions being parallelized is about 52.5%.

The GPU executions (**Figure 3**) did not see any improvements in throughput when compared to the CPU executions. It should be understood that this is an unfair comparison as the processors in the clusters were different, with the CPU NUMA having the superior processors. When only the CPUs are utilized on the GPU NUMA and compared with the throughput when both CPU and GPU are utilized, slight performance gains are observed.

In both **Figure 2** and **Figure 3**, it can be seen that the CNN execution with the batch size of 64 clearly outperforms the execution with batch size of 32. A similar but less significant performance increase can be seen between the batch size 128 execution and the batch size 64 execution. Each thread processes a batch independently so this trend can be explained by the additional overhead of having to processes twice as many batches in the batch size 32 execution compared to the batch size 64 execution.

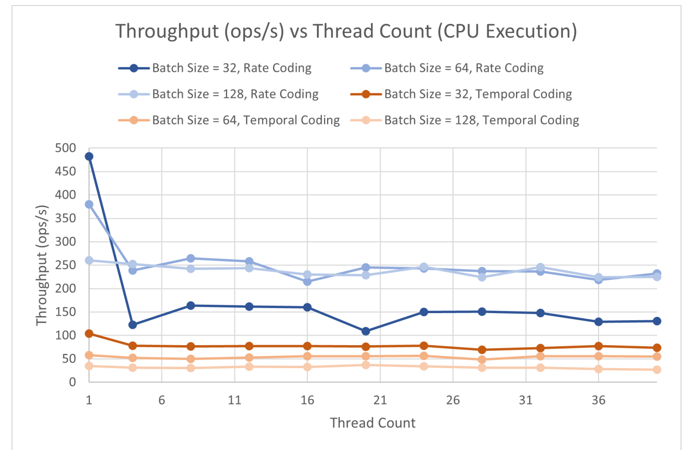

Fig. 4. Results from running the SNN Variations on the CPU

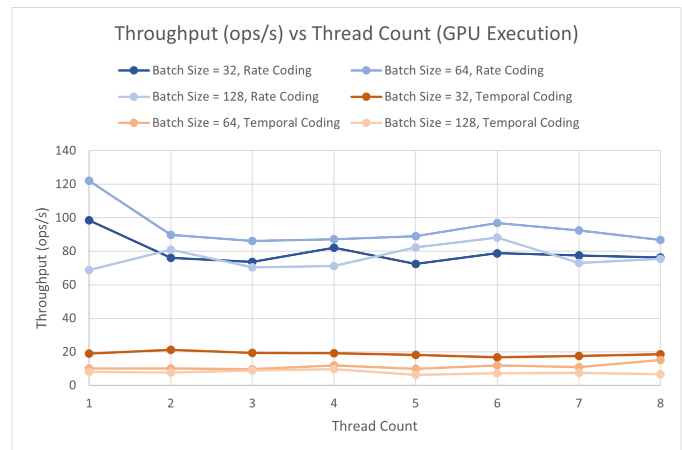

Fig. 5. Results from running the SNN Variations on the GPU

B. SNN Results

Figure 4 and **Figure 5** show the results from running the multithreaded SNN algorithm on the CPU and GPU systems respectively. As seen in each of the figures, adding threads to perform layer, synapse and weight updates substantially decreased the throughput of the system. On the CPU-based system, the best multithreaded execution is approximately 50% slower than the single threaded execution. Similarly, on the GPU-based system, the best multithreaded execution is approximately 20% slower than the single threaded execution.

C. Multithreaded Analysis

We investigate the reason for this decrease in the SNN's throughput by individually timing the threads instantiated by the `ThreadPool` object. **Figure 6** shows the results of measuring (1) the average time it takes for a thread to get its job from the pool, (2) the average time it takes for a thread to execute this task, and (3) the average amount of time it takes for all tasks to complete. These measurements were recorded for the layer update and weight update tasks. It can be observed that the initialization of a thread takes up nearly 50% of the execution time for both layer and weight updates. We suspect that because Python is a higher level

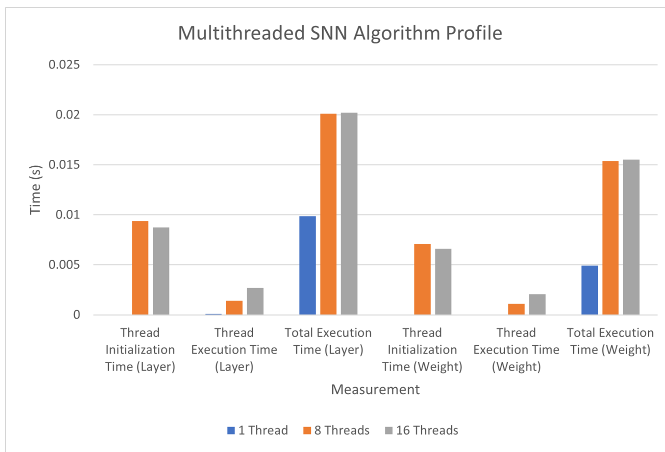

Fig. 6. Results from benchmarking ThreadPool tasks (Layer Updates and Weight Updates)

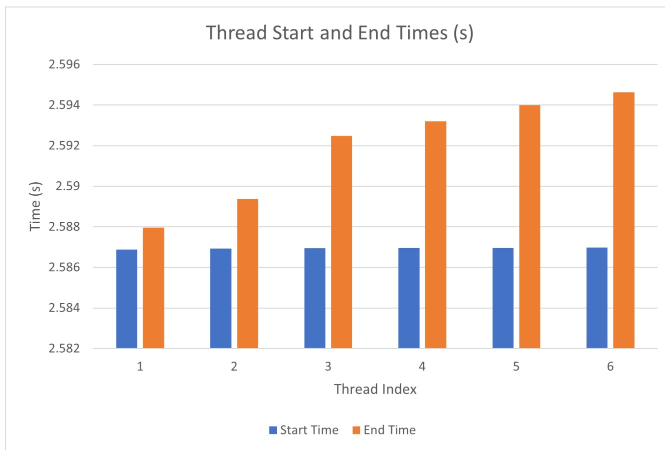

Fig. 7. Results from measuring ThreadPool task start and end times (Layer Updates)

scripting language, the multithreading features of the language have not been well optimized. Python further does not allow users to specify the priority of threads. Using a lower level programming language such as C or C++ may be able to improve the performance here by reducing the time it takes for threads to receive their tasks.

Additionally, threads initialized in the multithreaded algorithm take a longer amount of time to perform their task than the single threaded version. We believe that the policy employed by the scheduler is partially the reason for this, since worker threads may compete with other processes running on the system. To further understand this issue, we analyzed the start and end times of threads performing layer updates as shown in **Figure 7**. The results of this experiment show all of the threads begin processing their layer updates at approximately the same time. However, the time at which a given thread completes its update occurs later if the thread was initialized at a later time (i.e., a higher thread index). At this time we cannot conclude why this behavior was observed, but we suspect that it is again scheduler related.

V. Conclusion and Future Work

Our CNN experiments showed that although GPUs have the capability to improve matrix multiplication speed during forward propagation, certain parts of the backpropagation algorithm could only occur on the CPU and thus bottlenecked the execution. The reason for this is because machines on GPU cluster featured only 8 lower end CPUs compared to those on the dedicated 64 CPU cluster. Using higher batch sizes with the CNN also resulted in greater throughput since threads were capable of processing more data at each step.

The SNN experiments showed adding multithreading functionality to the network created a large overhead which doubled the execution time in most cases. This overhead was likely caused by the the time it took to provide tasks to the threads in the pool, in addition to the scheduling policy of the CPU. Furthermore, using temporal coding, a more sparse encoding scheme compared to rate coding, reduced the throughput of the SNN by about 80%.

Future work on this topic could attempt to look at SNN implementations on lower level programming languages such as C and C++. Brian2GENN appears to be a promising alternative since the framework was built with multiprocessing in mind [14]. Lastly, emulating the SNN on custom hardware (i.e., FPGA and ASIC) may yield better throughput.

References

[1] S. Zeki, "A massively asynchronous, parallel brain," *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 370, no. 1668, p. 20140174, 2015.
[2] J. Hawkins and S. Blakeslee, *On intelligence*. Macmillan, 2004.
[3] M. Shanahan, *The technological singularity*. MIT press, 2015.
[4] M. Bouvier, A. Valentian, T. Mesquida, F. Rummens, M. Reyboz, E. Vianello, and E. Beigne, "Spiking neural networks hardware implementations and challenges: A survey," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 15, no. 2, apr 2019.
[5] G. Li, L. Deng, Y. Chua, P. Li, E. O. Neftci, and H. Li, "Spiking neural network learning, benchmarking, programming and executing," *Frontiers in Neuroscience*, vol. 14, 2020.
[6] W. Maass, "Networks of Spiking Neurons: The Third Generation of Neural Network Models," Tech. Rep. 9, 1997.
[7] Y. Ma, F. Rusu, and M. Torres, "Stochastic gradient descent on modern hardware: Multi-core cpu or gpu? synchronous or asynchronous?" in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 1063–1072.
[8] S. Thorpe and J. Gautrais, "Rank order coding," in *Computational neuroscience*. Springer, 1998, pp. 113–118.
[9] A. N. Burkitt, "A review of the integrate-and-fire neuron model: I. homogeneous synaptic input," *Biological cybernetics*, vol. 95, no. 1, pp. 1–19, 2006.
[10] J. Sjöström and W. Gerstner, "Spike-timing dependent plasticity," *Scholarpedia*, vol. 5, no. 2, p. 1362, 2010, revision #184913.
[11] P. Gupta, "Cuda refresher: The cuda programming model," 2020. [Online]. Available: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/
[12] H. Hazan, D. J. Saunders, H. Khan, D. Patel, D. T. Sanghavi, H. T. Siegelmann, and R. Kozma, "BindsNET: A Machine Learning-Oriented Spiking Neural Networks Library in Python," *Frontiers in Neuroinformatics*, vol. 12, p. 89, dec 2018. [Online]. Available: https://www.frontiersin.org/article/10.3389/fninf.2018.00089/full
[13] T. Iakymchuk, A. Rosado, J. V. Frances, and M. Batallre, "Fast spiking neural network architecture for low-cost fpga devices," in *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, 2012, pp. 1–6.
[14] M. Stimberg, D. F. Goodman, and T. Nowotny, "Brian2genn: accelerating spiking neural network simulations with graphics hardware," *Scientific Reports*, vol. 10, no. 1, pp. 1–12, 2020.