Article

Not peer-reviewed version

Enabling Efficient On-Edge Spiking Neural Network Acceleration with Highly Flexible FPGA Architectures

Samuel López-Asunción * and Pablo Ituero

Posted Date: 20 February 2024

doi: 10.20944/preprints202402.1107.v1

Keywords: Neuromorphic Processing; Spiking Neural Networks; FPGA; On-Edge Computing.

Article

Enabling Efficient On-Edge Spiking Neural Network Acceleration with Highly Flexible FPGA Architectures

Samuel López-Asunción* and Pablo Ituero

Laboratorio de Sistemas Integrados, Departamento de Ingeniería Electrónica, Universidad Politécnica de Madrid, 28040 Madrid, Spain; pablo.ituero@upm.es

* Correspondence: samuel.lopez.asuncion@upm.es

† This paper is an extended version of our paper published in 2023 XXXVIII Conference on Design of Circuits and Integrated Systems (DCIS): *Flexible Deep-pipelined FPGA-based Accelerator for Spiking Neural Networks*.

Abstract: Spiking neural networks (SNNs) promise to perform tasks currently done by classical artificial neural networks (ANNs) faster, in smaller footprints and using less energy. Neuromorphic processors are set out to revolutionize computing at a large scale, but the move to edge-computing applications calls for finely-tuned custom implementations to keep pushing towards more efficient systems. To that end, we have examined the architectural design space for executing spiking neuron models on FPGA platforms, focusing on achieving ultra-low area and power consumption. This work presents an efficient clock-driven spiking neuron architecture used for the implementation of both fully-connected cores and 2D convolutional cores, which rely on deep pipelines for synaptic processing and distributed memory for weight and neuron states. With them, we have developed an accelerator for an SNN version of the LeNet-5 network trained on the MNIST dataset. At around 5.5 slices/neuron and only 348 mW, it is able to use 33% less area and 4 times as less power per neuron as current state-of-the-art implementations while keeping low simulation step times.

Keywords: neuromorphic processing; spiking neural networks; FPGA; on-edge computing

1. Introduction

Spiking neural networks (SNNs) are the third generation of artificial neural networks (ANNs), and one of the main approaches to neuromorphic computing [1]. Their structure and behaviour are inspired by that of biological neural systems [2], and try to emulate their physiology to process and convey information. Like real neurons, they use an event-based asynchronous mechanism based on the generation and propagation of action potentials, also called spikes.

These networks are attractive because they are usually arranged in massively-parallel, sparsely connected structures and, due to their characteristics, could process information more efficiently than classical, second-generation ANNs. Lower energy and resource requirements, and new computing and learning paradigms are some of the theoretical improvements to be seen in the future [3]. With several different models in increasing levels of detail that represent the actual chemical reactions happening inside living cells [4], they are also of interest in neuroscience, where simulations of these networks could bring some insight about how information is managed in our brains [5,6].

Real neurons receive spikes of ionic current at their inputs, and generate their own spikes at their output. A similar artificial building block that can spike when presented with a certain stimulus can recreate its behaviour, and we can do so *emulating* its dynamics using specific features of analog electronic devices [7]. On the other hand, digital implementations rely on differential equation solvers to *simulate* the internal state of a neuron. They are inherently resistant to noise, mismatch, power and temperature variations, and they are more flexible when designing, updating or reprogramming network structures, which is useful to quickly test new algorithms and models.

There are some concerns about the actual improvement in efficiency over ANNs. The paradigm shift to spike-based computation still relies heavily on rate-coded spiking implementations and conversion of already trained conventional ANNs, which have not shown those promised improvements [8]. Information encoding is a big challenge and learning algorithms are not quite there yet compared to backpropagation in ANNs. On top of that, the lack of standard frameworks for digital SNN software/hardware development and benchmarking, and the fact that SNN network design and hardware optimization are strongly coupled make research and testing more difficult.

To try and overcome these big hurdles, there is a substantial amount of research effort going into digital SNN architectures [9]. They can be simulated on traditional processors (e.g. CPUs, GPUs [10]) but their competitive advantages regarding energy consumption and parallel processing shine when implemented on custom digital hardware. On one side, dedicated massively-parallel hardware systems like SpiNNaker [11], TrueNorth [12], Loihi [13] and Tianjic [14] represent the group of large digital ASIC platforms for large-scale simulations. On the other side, FPGAs stand as stellar candidates to perform as SNN accelerators [15] for implementations that need smaller sizes and low energy consumption, offering the possibility to design more flexible neuromorphic processors targeting applications on the edge.

This work tries to fill some gaps in FPGA-based SNN development by introducing a new clock-driven neuron processing core architecture. Targeting a low area and low power implementation, these cores are able to run groups of neurons of arbitrary size, only limited by simulation time and device resources. They are also flexible enough to run the most commonly used neurons, from Leaky-Integrate-and-Fire models to more complex conductance-based ones. This allows them to be used as building blocks for future SNN frameworks that map high-level descriptions of spiking networks into digital hardware.

We have designed two different cores implementing fully-connected and 2D convolutional layers of spiking neurons. They feature:

- A high number of neurons in a compact implementation by making heavy use of pipelining and distributed memory, using 33% less FPGA space and power per neuron than current architectures while keeping simulation step times under 140 µs.
- Support for a wide variety of neuron models by following a powerful design pattern, which also makes it an interesting choice to perform automatic mapping from network description languages [16] to FPGA-based SNN accelerators.

As a way to showcase the core implementations, an SNN version of the LeNet-5 network architecture has been developed. Generated by converting an ANN trained on the MNIST handwritten digits dataset, our SNN equivalent implements all its layers with the corresponding SNN cores. We do not focus on network training or inference accuracy, but rather we use it as a way to test our neuron circuits and get area, timing and power estimation measurements and compare them to some of the latest implementations in the state-of-the-art. We try to decouple hardware optimization from network design, so all our design choices while exploring the hardware design space are centered around design optimizations for area, throughput and power of neural model circuits.

Concerning the structure of the document, we first introduce prior work on spiking neuron implementations and some key concepts about the requisites for a model to be supported by the hardware, shown in Section 2. We present our spiking neuron core architecture, showing the fully-connected core in Section 3 and moving onto the 2D convolution core design in 4. We show our chosen design for the LeNet-5 network and some of its features regarding size, memory, and layer correspondence to hardware cores in Section 5. Then, its area, power, and circuit performance characteristics are evaluated and compared against state-of-the-art implementations in Section 6. Finally, Section 7 concludes this work and raises questions for future research.

2. Background

Spiking neural networks are arrangements of neurons commonly organized in different layers. The individual neurons implement a computational model composed primarily by a synaptic processing stage followed by a differential equation solver running the neuron internal state.

When trying to define these models in FPGA-based developments, the network design and training process is typically intertwined with the hardware implementation. As a result, many FPGA-based implementations are ad hoc custom designs for specific network architectures to perform specific tasks. For us to focus on hardware design, we need to decouple the network design process from the circuit implementations. We can use high-level SNN description languages like PyNN [16] to define networks in a platform-independent way, and have them mapped later onto synthesizable hardware descriptions capable of running their neuron models.

2.1. Neuron Model Rules

In order for those high-level model descriptions to be supported by these circuits, we lay down a set of rules [17]:

1. The internal state of a neuron is modelled by a set of differential equations (deterministic or stochastic, ordinary or partial) and continuously evolves with time.
2. Input spikes received through the input synapses trigger changes in those state variables (spikes are binary, discrete events).
3. Output spikes are generated when some condition is satisfied.

Many current neuron models can be described following these rules, shown schematically in Figure 1, such as the commonly used Leaky-Integrate-and-Fire or even more complex models like the Hodgkin-Huxley neuron. For the sake of simplicity, we will focus on the implementation of a hardware optimized version of the LIF model.

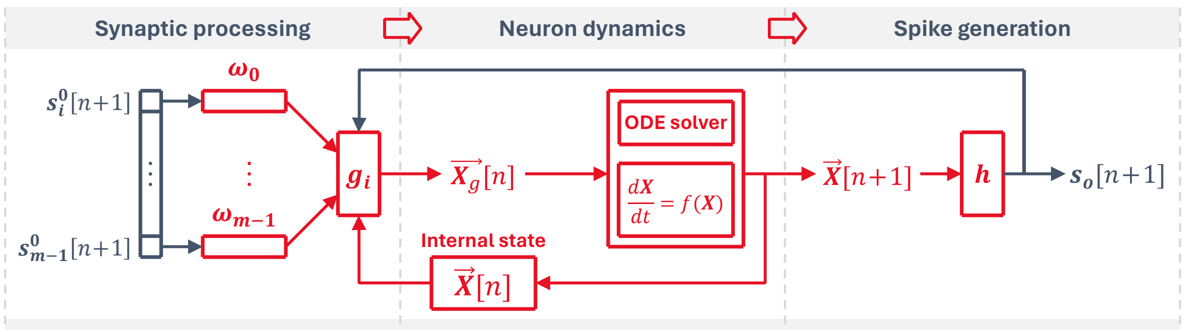

Figure 1. Schematic view of the three neuron model rules.

In this framework, the internal state of a neuron is held in a state variable vector $\boldsymbol{X}[n]$. This state is updated by incoming input spikes via the input synapses, which are defined as units that change the state vector through an arbitrary function g_i

$$\mathbf{X}_{g_i}[n] = g_i(\boldsymbol{X}[n], s_i[n+1]) \tag{1}$$

which is most commonly a weighted addition of the input spike

$$\mathbf{X}_{g_i}[n] = \boldsymbol{X}[n] + \omega_i s_i[n+1] \tag{2}$$

where ω_i is the weight of the input synapse number i and s_i is the input spike at that synapse. The state vector is constantly driven towards a new value based on the natural dynamics of the neuron in every simulation time step. To define a neuron model, we need two elements:

- Neuron dynamics: the set of differential equations stated as the rate of change of the neuron state

$$\frac{d\boldsymbol{X}_g}{dt} = f(\boldsymbol{X}_g) \tag{3}$$

where $\boldsymbol{X}_g$ is a vector holding the state of the neuron after the input synaptic processing.

- Equation solver: since the simulation is discrete in time, we need a numerical method to solve the next step of the neuron state. An example of this is the Forward Euler method

$$\boldsymbol{X}[n+1] = \boldsymbol{X}_g[n] + \frac{d\boldsymbol{X}_g}{dt} \cdot t_{step} \tag{4}$$

where t_{step} is the simulation step time. This method is accurate enough to showcase the LIF model.

Lastly, the neuron fires an output spike when the state vector satisfies the threshold condition

$$s_o[n+1] = h(\boldsymbol{X}[n+1]) \tag{5}$$

where h is 1 or 0 if the condition is reached or not, respectively. Neurons with discontinuous dynamics, such as those having state resets like LIF neurons, can be incorporated into these rules by having the output spike from the previous time step fed back into itself through an additional virtual synapse

$$\boldsymbol{X}_v[n] = g_v(\boldsymbol{X}[n], s_o[n]) \tag{6}$$

which in the particular case of a state reset would be described as

$$\boldsymbol{X}_v[n] = \begin{cases} \boldsymbol{0}, & s_o[n] = 1 \\ \boldsymbol{X}[n], & s_o[n] = 0 \end{cases} \tag{7}$$

This virtual synapse is key when developing flexible circuits that are able to run arbitrary neuron models using standard numerical methods for continuous dynamics.

2.2. Clock-Driven vs. Event-Driven Simulations

There are some simulation techniques for event-driven models, where the state is updated only when input spikes arrive. They may reduce the amount of processing since spike events are supposed to be sparse [18]. Clock-driven methods, however, are widely used because they can be more easily described and simulated [19], and many neuron models in this class can be constrained to follow these rules, which are meant for simulators that run through every time step.

Since we are implementing hardware with support for arbitrary models, we have chosen a clock-driven approach to implement the circuits. Besides, they are the only solution if we need to continuously monitor some variable in the network, such as specific membrane voltages, in order to analyze their behaviour, which may be of interest in some cases such as neuroscience-related applications.

2.3. Motivation and Previous Works

FPGA-based accelerators are based on specialized hardware to run simulations of neuron models. Taking advantage of the fact that neurons process information independently of the rest of the network, and leveraging the features and resources in these devices, the hardware designer can use parallelization and pipelining to accelerate these computations. Their flexibility is also a bonus when designing edge applications, compared to both small and large-scale spiking neural network ASICs [9].

One of the first FPGA systems is Minitaur [18], which features an event-driven spiking neuron implementation, a strategy also followed by more recent works [20,21]. In these systems, neurons only run when there are spikes at its input, which can help achieve better efficiencies. There are also many clock-driven FPGA implementations, which extensively use parallelization in large FPGA devices to get high speeds in relatively low area and power [22,23].

Some approaches are also moving towards end-to-end generation of accelerators from hardware-agnostic descriptions or even classical ANN models [24,25]. We firmly believe that this decoupling is necessary for the proliferation of SNN hardware accelerators, and we have also taken a similar route, trying to develop neuron circuits that can be easily mapped from high-level languages.

However, we take a different approach in that we have tried to spread the weight and neuron state data across the FPGA memory resources as much as possible to increase the data bandwidth without sacrificing time step latency. We have heavily relied on the FPGA block memories, distributed RAM and LUTs to implement a deeply pipelined architecture of the mentioned neuron model rules able to run a large number of neurons using very little area and power, which enables us to use lower-end devices.

3. Fully-Connected Core

Our implementation is based on the concept of *neuron core* as a single unit that can simulate groups of neurons in a pipelined manner, rather than individual neurons.

This is a natural consequence of doing a pipelined hardware implementation of a single spiking neuron, like that of Figure 2a. Grouping together a set of N neurons with M synapses which are connected to the same presynaptic outputs allows us to use the same circuit with some additional control hardware to run all neurons sequentially, as represented on Figure 2b. By feeding the pipeline with information (i.e. neuron state and weights) from each neuron every cycle we can process them all to generate a vector of output spikes, essentially simulating a fully-connected layer of spiking neurons. A pipelined core takes advantage of some key aspects of spiking neural networks:

- In high-level SNN description languages, neurons are usually defined in groups or populations that are easily mapped into neuron cores. These can then be sized accordingly based on the number of neurons and synapses.
- All processing inside a neuron model is self-contained, meaning that the information inside each neuron does not depend on other neurons. This, together with the fact that simulation step times (0.1 ms to 10 ms) are usually orders of magnitude larger than typical FPGA clock speeds, makes hardware pipelining an interesting choice.
- Since internal states and synaptic weights are accessed sequentially and independently, we can use distributed memory blocks, which are readily available on FPGAs.

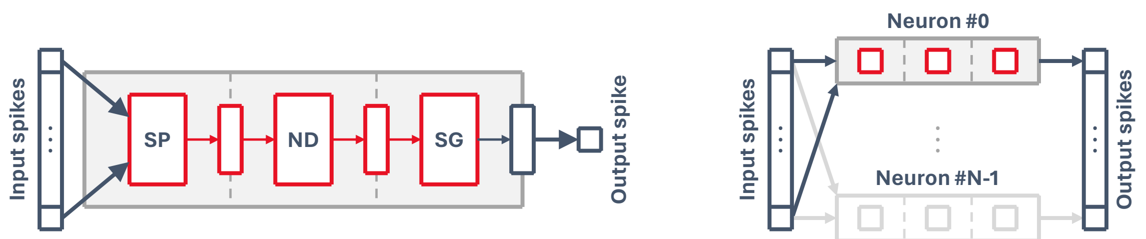

Figure 2. (**a**) Simplified view of a pipelined implementation of a generic neuron model. A single neuron accepts an array of input spikes connected one-to-one to the neuron synapses and generates a single output spike. *SP = Synaptic Processing, ND = Neuron Dynamics, SG = Spike Generation*. (**b**) Representation of the core neuron group: a core can process N neurons that have the same M input spikes, generating N outputs.

3.1. Architecture

Our circuit, shown in Figure 3, implements the three main operations: synaptic processing, neuron dynamics and spike generation. Firstly, the virtual synapse is placed at the beginning of the synaptic processing stage, which fetches from memory the previous neuron state and the output spike information from the last time step, pushing the updated state into the pipeline. After that, each synaptic weight circuit occupies one pipeline stage, pulling data from individual memory elements, where each memory element i holds ω_i for all N neurons. This implements Equation (2) for each one of the neuron synapses.

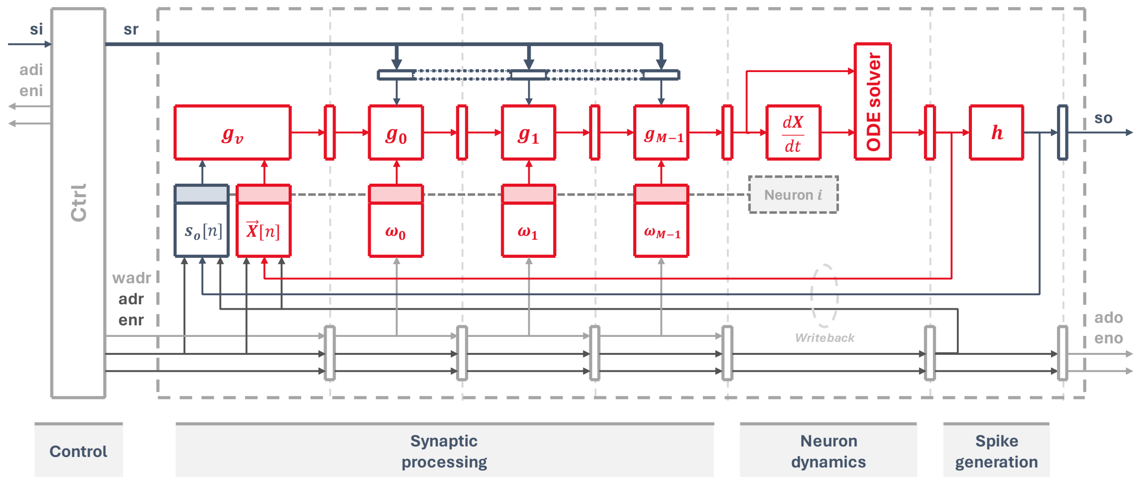

Figure 3. Hardware schematic of the pipelined neuron cores, showing the synaptic processing, dynamics and spike generation stages.

The next stage calculates the updated state for the next time step using a differential equation solver, as per Equations (3) and (4). Lastly, a threshold circuit checks this new state and generates an output spike when the threshold condition is reached, as defined in Equation (5).

The input spikes are read from an input memory, addressed by the signals *adi* and *eni*. To manage the flow of data and feed the pipeline with information from all neurons, a control state machine is laid out at the first stage. It acts as a sort of program counter that generates the input addresses to read the input spikes. It also generates the output neuron addresses, propagating them through every stage to fetch all the weights and state of each neuron. At the end of the pipeline, they are used to update neuron states and output spikes which are written back into their corresponding memory elements, and to store the outputs into an external spike memory through the signals *ado* and *eno*.

When the time step input signal arrives, which marks the start of one simulation time step, all input spikes are processed sequentially by the synaptic processing stage, then the equation solver updates the neuron states and finally all output spikes are generated, as shown in Figure 4.

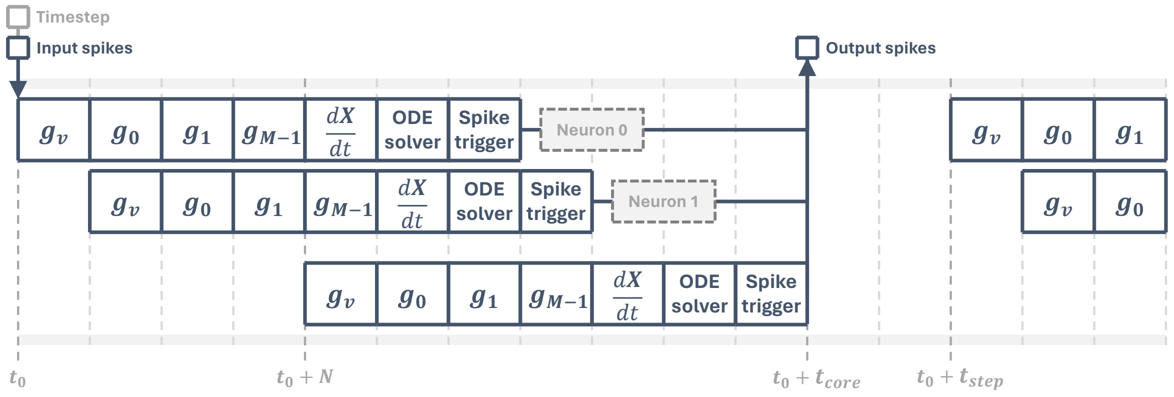

Figure 4. Core processing timeline. Following the first simulation time step at t_0, the core feeds the pipeline with neuron data and processes the input spikes, generating an output spike vector at $t_0 + t_{core}$, where t_{core} is the core latency. In this diagram, it is assumed that all operations take one clock cycle.

3.2. LIF Core Implementation

Arguably, the LIF neuron is the most used neuron model in SNN design. As illustrated in Figure 5, we implement a simplified LIF neuron model where the state variable of each neuron is defined as a scalar value $\boldsymbol{X}[n] \equiv u[n]$ holding the membrane voltage, and the core pipeline is implemented as follows:

1. Synaptic processing: the core implements the synaptic model of Equation (2), where static weights are used for training and inference. To simplify the implementation, weights can not be updated on the fly and the simulation relies on offline training. The virtual synapse implements the LIF state reset:
$$u_v[n] = \begin{cases} u_{rest}, & s_o[n] = 1 \\ u[n], & s_o[n] = 0 \end{cases} \tag{8}$$
taking the previous membrane voltage and resetting it to its rest potential if the neuron has spiked in the previous time step. After that, the following synaptic stages take that updated potential and sequentially process the input spikes
$$u_{g_i}[n] = u[n] + \omega_i s_i[n+1] \tag{9}$$
2. Neuron dynamics: the state equations are hard-coded into the core definition, although they can be modified before synthesis if the model needs to be changed. The rate of change for this simplified LIF model is a linear function of the difference between the membrane voltage and its resting potential
$$\frac{du}{dt} = (u_g[n] - u_{rest}) \cdot 2^{-\alpha}, \quad \alpha = 5 \tag{10}$$
where u_g is the membrane potential after the synaptic processing and $2^{-\alpha}$ is a parameter that determines how fast the membrane goes back to its resting potential. This coefficient is a power of two to implement the multiplication as a bit-shift, but the core can incorporate multipliers if the model needs more generic parameters. The equation solver simply adds this value to the previous state, assimilating the t_{step} value into $2^{-\alpha}$.
$$u[n+1] = u_g[n] + \frac{du}{dt} \tag{11}$$
3. Spike output: the spike output condition function of Equation (5) is implemented in this case as a simple threshold which is compared against the state variable
$$s_o[n+1] = \begin{cases} 0, & u[n+1] < u_{th} \\ 1, & u[n+1] \geq u_{th} \end{cases} \tag{12}$$
where u_{th} is an arbitrary constant threshold potential.

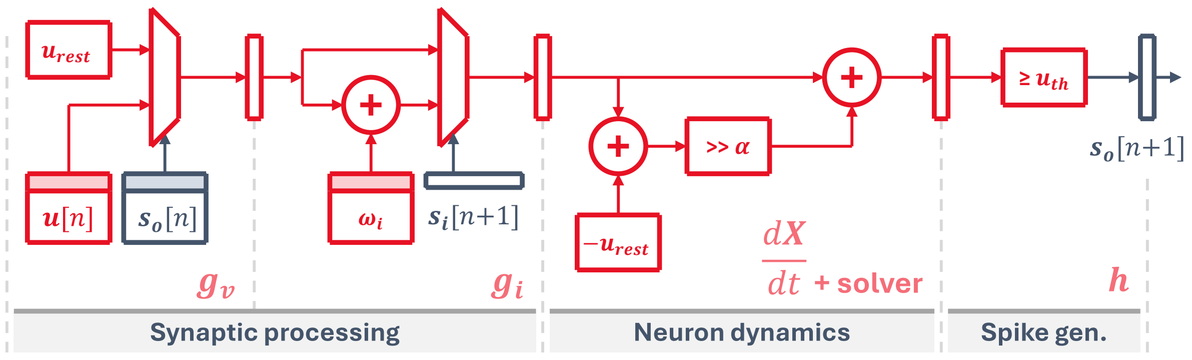

Figure 5. Simplified Leaky-Integrate-and-Fire implementation of a fully-connected core, with the virtual synapse g_v, synaptic stages g_i, neuron dynamics $\frac{dX}{dt}$ + solver and spike generation stage h.

3.3. Latency

Cores need to perform all neuron operations before the next simulation time step arrives. Their latency depends on the number of synapses M, determining the pipeline length, and the total number of neurons N. In this LIF implementation all synaptic and state update operations happen in a single cycle, so we have

$$t_{core} = [(M+4) + (N-1)] \cdot t_{clk} \tag{13}$$

where t_{clk} is the system clock period and t_{core} is the core latency, shown in Figure 4. For the simulation to be able to run, all cores must satisfy $t_{core} < t_{step}$.

Neurons are isolated processing units, which means they do not need information from other neurons to work. A pipelined implementation like this one is then free of data hazards and, since all synapses have their own stage, control or structural hazards are not a problem either. This allows us to split the core into smaller groups of neurons and parallelize the computations as much as we want. If we divide the total number of neurons by a constant parallelization parameter d and distribute the neurons evenly between cores we get a lower latency value

$$t_{core} = [(M+4) + (\frac{N}{d} - 1)] \cdot t_{clk} \tag{14}$$

and if we do it so that each core only runs one neuron (i.e. $d = N$), we get a minimum value for $t_{core} = (M+4) \cdot t_{clk}$. This is really useful when mapping high-level descriptions into hardware as it lets us map big groups into smaller cores that are able to keep up with the simulation time step.

3.4. Quantization and Memory

Each neuron holds information about its state $u[n]$, its weights ω_i —where i is the synapse number— and their previous output spike $s_o[n]$. Synapses and neuron dynamics are split up into pipeline stages, so we need distributed memory to fetch all that information separately and continuously. This way we can overlap the calculations for all neurons along the pipeline.

LIF neuron model implementations can vary, having different numeric representations and specific operations. We have focused on fixed-point 16-bit values to hold the membrane potential and weights, as we consider it large enough to be representative. Also, according to the neuron model rules, all spike events are stored as bits, which is convenient as many commercial FPGAs can implement 1-bit wide block RAMs.

If a core is sized to run N neurons, and assuming that all neurons are fully connected and have M synapses, the needed storage space is

$$m = N \cdot (2M + 2 + \frac{1}{8})\ \mathrm{B} \tag{15}$$

where m is the total memory usage in bytes. For instance, a core running 64 LIF neurons with 64 possible different synapses needs roughly 4.2 kB of block memory.

4. Convolutional Core

A convolutional layer can be implemented using a fully-connected core with N neurons, corresponding to the total number of output pixels, and M synapses per neuron, corresponding to the number of input pixels. It is obvious that this is a very wasteful approach since all neurons have the same weights, and each output neuron only covers a small region of the input feature map the size of the kernel. This means that only the synaptic weights around that region are meaningful while the remaining weights, and in fact most of them, are zero.

It is worth noting that the concept of spiking convolutional layers emerges from classical CNNs, where biological plausibility is not a concern. Kernels sweep along and across the input feature maps to generate smaller output maps that encode more complex information. However, they have to move around in order to do so, that is, the kernel synapses are not spatially bound to the presynaptic neurons, which does not seem like something that actually happens in biological systems. Because of that, in some ANN-to-SNN converters [26] convolutional layers are mapped into dense layers that have static synaptic connections, but are way larger than it is required to store kernel information.

Nevertheless, they are computationally useful. In order to design an efficient spiking convolutional core, we can greatly reduce memory usage and area by making a couple modifications to the fully-connected neuron core architecture:

- Removing all unnecessary zeros: every output neuron only processes information about a small region of the input space. That means that the rest of them are zero and can be ignored. A fully-connected layer would have $M = (n_x n_y)^2$, while a layer with no zero-valued weights would have $M = k_x k_y N$.
- Storing kernel weights only once: all neurons share the same kernel weights, so the number of synaptic weights drops again to $M = k_x k_y$.
- Emulating kernel stride/movement: by carefully arranging the kernel weights and getting the input information in a certain order, we can take advantage of the input synaptic pipeline to "move" the kernel around. This is done all while minimizing the amount of intermediate storage needed inside the pipeline.

4.1. 2D Convolution

As can be seen in Figure 6, convolutional layers may have a single 2D input feature map which generates another 2D output map. If we have more than one kernel, the output is a stack of 2D maps. On the other hand, if we have a single kernel, but a stack of 2D input features, then the kernel is 3-dimensional, but we still have a single 2D output feature because the kernel z-dimension is equal to the number of inputs, and it only moves along the x and y axis. Lastly, we may have a stack of 2D input features and more than one 3D kernel, which would generate another stack of 2D output maps.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 February 2024 doi:10.20944/preprints202402.1107.v1

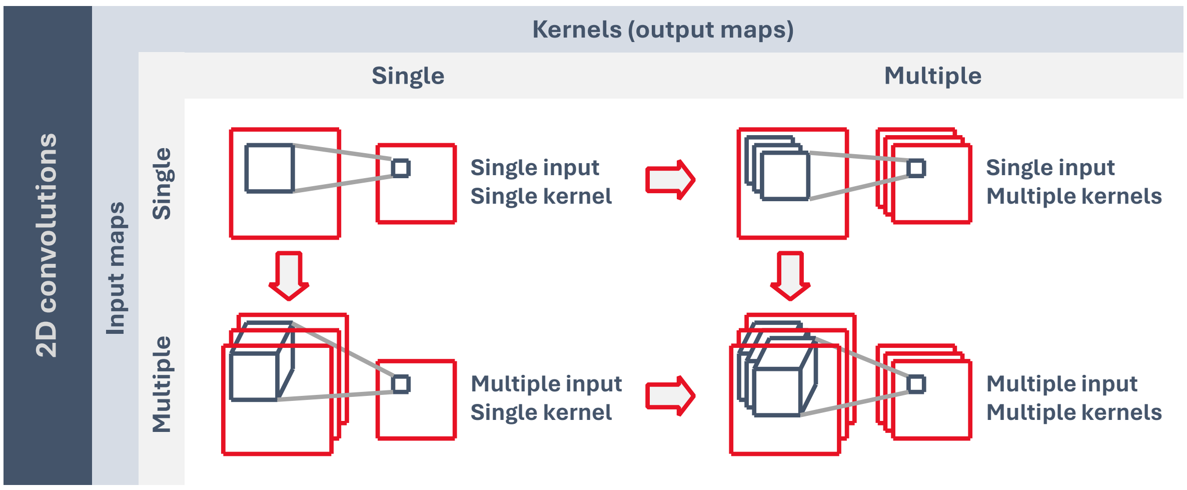

Figure 6. 2D convolutional layer configurations based on the number of inputs and kernels (outputs).

The architecture is understood more clearly if we start from the ground up by building a 2D single-input single-kernel non-pipelined implementation and progressively build our way up to a 2D multi-input multi-kernel pipelined convolutional core.

4.2. Single-Input Single-Kernel 2D Convolution

Input spikes are read from the presynaptic output memory and, in the fully-connected core, we read and latch the whole input vector into the synaptic pipeline sequentially. This time, the control unit manages a FIFO-like structure called a line buffer where, based on the kernel size, only the necessary information from the input map is available to the pipeline at any time.

Figure 7a shows a representation of the line buffer against an input image, where we can see the pixels that need to be stored at any given time for a single output pixel. When the kernel moves —to the right in this case—, we simply shift the new pixel into the line buffer, which is implemented as a long shift register. By doing this, we shift the input data in instead of physically move the kernel around. This way can get the kernel input image region by taking it from specific locations on the buffer, and we can directly apply the corresponding kernel weights as constant values.

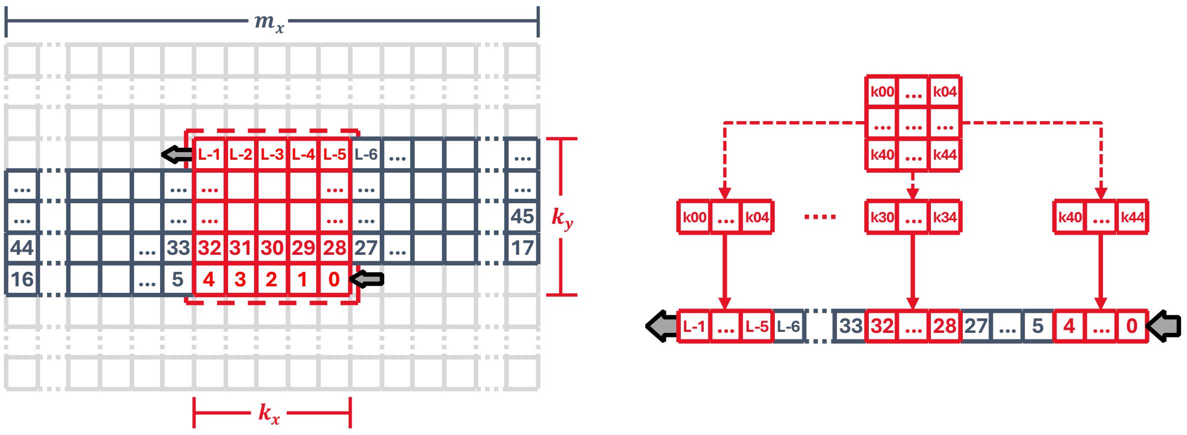

Figure 7. (**a**) Line buffer against a 28x28 2D image. (**b**) 5x5 kernel correspondence to the line buffer pixels.

Since the spikes are binary, it is feasible to implement the line buffer as a register rather than with RAM memory. Its length depends on the kernel size (k_y, k_x) as well as the input map width as:

$$L = (k_y - 1)m_x + k_x \tag{16}$$

By taking our line buffer and representing it as a straight line, as it is shown in Figure 7b, we see that it holds k_y segments of k_x bits, which correspond to the kernel inputs. As the input data flows in, those segments hold the data for the next stride as if the kernel was moving itself. For a non-pipelined implementation, the output calculation is just a matter of multiplying all the inputs with their corresponding kernel weights and waiting for the next input spike to come in.

4.2.1. Pipelining

In the next step, we pipeline the implementation into one stage per kernel weight. To that end, we need to copy the line buffer kernel segments into the pipeline, so that each stage gets all input bits at the right moment, as illustrated in Figure 8. It can be shown however that those copies are not needed at all, since they can be fetched from the very line buffer at specific positions up ahead the shift register as long as the input is read continuously. Every row has its own access point, whose position in the line buffer is given by

$$r_i = ((k_y - 1) - i)m_x + (k_x - 1) \tag{17}$$

$$t_i = r_i + k_x i \tag{18}$$

where r_i is the last position in the line buffer for a specific row i, and t_i is the tap point for each row.

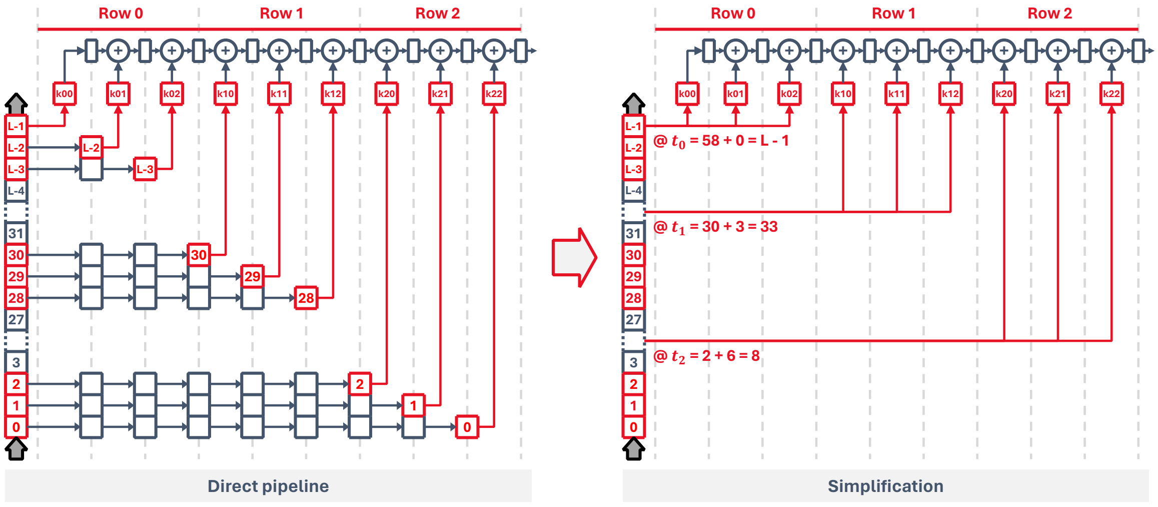

Figure 8. Single-input single-kernel 2D convolution implementation, 3x3 kernel on 28x28 input image, pre- and post-simplification.

The kernel weight order is arbitrary and can be changed, leading to some other configurations with different properties. For instance, if we place the weights in reverse (with respect to the previous implementation) we see that we can also simplify the pipeline, but the line buffer must be $k_x k_y - 1$ bits longer. However, this configuration has a fanout of just one stage per line buffer bit, as opposed to the k_x stages per bit on the forward configuration, which may be of use in high speed setups.

4.2.2. Control and Padding

Since access to the input feature map is sequential, the core starts by filling in the line buffer with its first row. The control logic uses two counters x and y representing the input map coordinates and generates an input address to read the flattened image from an input memory as

$$a_i = m_y y + x \tag{19}$$

As illustrated in Figure 9, we are not able to generate an output until the line buffer has enough bits so that the kernel is aligned with the top-left corner, which causes the pipeline to generate dummy values for the first cycles. During this time the control logic generates a rejection signal and the outputs are discarded. In a similar way, when the kernel reaches the right edge of the input feature map we need to wait k_x cycles for the kernel to wrap around and get back to the left side. During this time, the outputs are also ignored. Both conditions can be combined into

$$x \geq (k_x - 1) \wedge y \geq (k_y - 1) \implies \text{Kernel is aligned / output is valid} \tag{20}$$

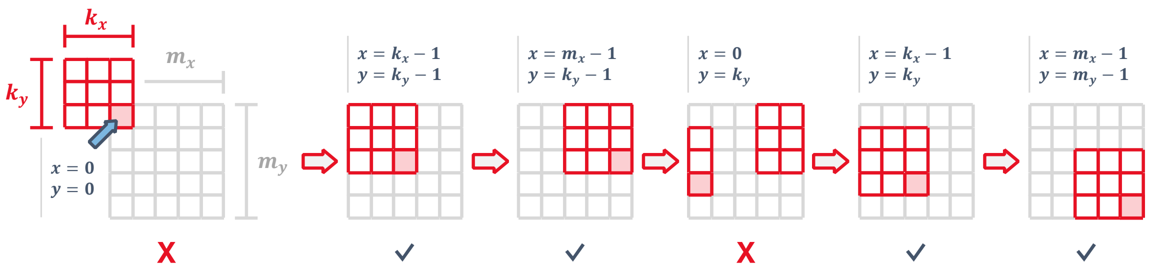

Figure 9. Kernel alignment of the single-input single-kernel 2D convolution. A control signal is generated when the kernel is within the image boundaries and the outputs are valid.

The output map has a size of $n_x = m_x - k_x + 1$ by $n_y = m_y - k_y + 1$. Ouptut addresses are generated which can be expressed as a function of the current x, y input pixel as

$$\begin{aligned} x_o &= x - (k_x - 1) \\ y_o &= y - (k_y - 1) \\ a_o &= n_y y_o + x_o \end{aligned} \tag{21}$$

We focus on generating a convolution with no padding since it is the most straightforward to implement. However, the line buffer could be filled with zeros at the correct times to perform a convolution with input padding.

4.3. Multiple-Input Single-Kernel 2D Convolution

When jumping onto convolutions of multiple 2D input maps, the kernel turns into a 3D object that runs along the x and y axes. Generally, to move the kernel along a specific axis j and compute its output one needs to read a new set S of input pixels. This set contains the pixels in the plane next to the current position in the direction of the axis. The number of new pixels is

$$|S| = \prod k_i \quad \forall i \neq j \tag{22}$$

where i represents all axes except for the one the kernel is moving along, and k_i is the kernel size along axis i. Since we have chosen to move through the x axis first, we would need to get $|S| = k_z * k_y$ input pixels every time we want to compute a new output pixel. However, thanks to the line buffer we already have some of the previous values ready to go, so the actual number of pixels to be read every time drops to $|S| = k_z$. We also keep the previous strategy of discarding the outputs while the kernel is rolling over the edges in the x dimension.

The control logic assumes that the input maps are stored sequentially, so their pixel values are not interlaced. The input address generation then changes to

$$a_i = m_x m_y z + m_y y + x \tag{23}$$

The fact that we have to wait for all pixels in the z dimension to generate a single output is not a problem as long as the control logic keeps track of the kernel alignment. While the input pixels are loaded in, the circuit rejects all generated output values until the line buffer contents are properly aligned with the kernel weights, that is, every k_z reads. The alignment condition turns into

$$\begin{matrix} x \geq (k_x - 1) \,\wedge\, y \geq (k_y - 1) \\ z = (k_z - 1) \end{matrix} \implies \text{Kernel is aligned / output is valid} \tag{24}$$

3D Line Buffer

Now, instead of reading in single pixels, we read all input map values for every i, j pixel and store them sequentially into the line buffer, as shown in Figure 10, so for every pixel we store a vector of k_z values. It is the most convenient way of going through all input maps: the kernel z dimension is equal to the number of maps, so as soon as we compute an output pixel we can completely discard the last values in the line buffer without having to read them again.

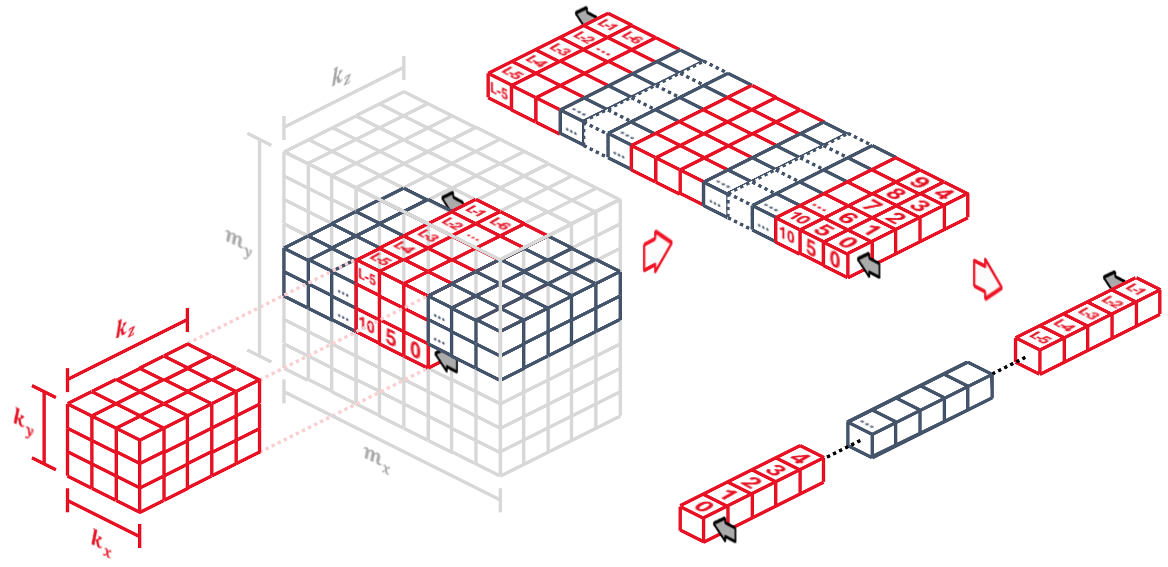

Figure 10. 3D line buffer against a stack of five 2D input maps for a 3x3x5 kernel. The 3D buffer unfolds to be implemented as a linear shift register.

Since the line buffer holds k_z values for every pixel, the buffer length is now

$$L = ((k_y - 1)m_x + k_x)k_z \tag{25}$$

The row and tap positions also change slightly, but they still keep the convenient property of being static locations in the line buffer regardless of the kernel z dimension, as long as the kernel values are properly connected along the synaptic pipeline. They become

$$r_i = ((k_y - 1) - i)k_z m_x + (k_z k_x - 1) \tag{26}$$
$$t_i = r_i + k_z k_x i \tag{27}$$

4.4. Multiple-Input Multiple-Kernel 2D Convolution

To extend this architecture to support multiple kernels so that the output is a stack of 2D feature maps, we look back to the fully-connected synaptic architecture. While the previous convolutions had constant kernel weights, in this case we add a small array of weights holding all kernels along the pipeline stages, as shown in Figure 11.

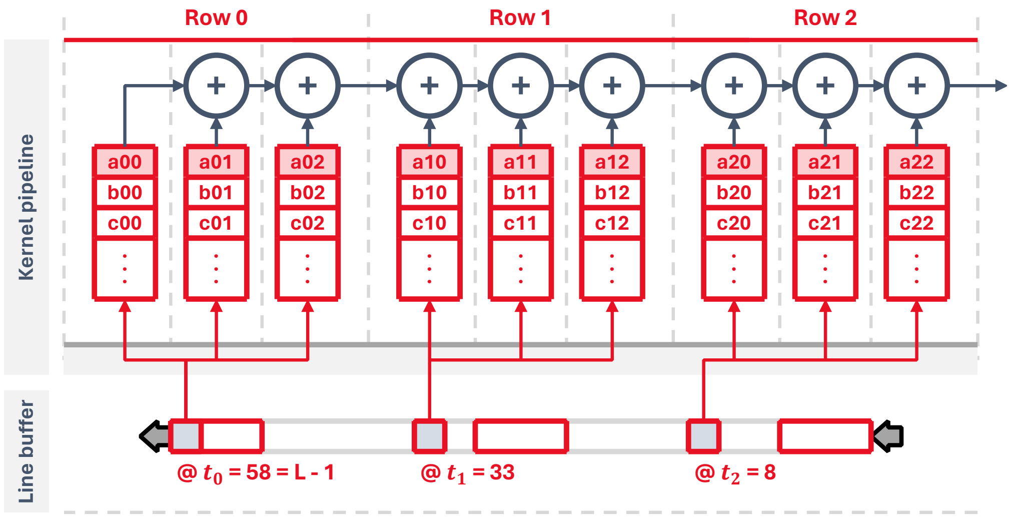

Figure 11. Single-input multiple-kernel 2D convolution implementation with 3x3 kernels. The same architecture is used in multiple-input multiple-kernel implementations.

By combining the two previous strategies we can build the most general case where multiple input maps are processed with multiple 2D kernels to generate a stack of 2D output feature maps. The controller reads all input maps once per kernel, sequentially computing the corresponding output maps, whose dimensions are $n_x = m_x - k_x + 1$ by $n_y = m_y - k_y + 1$. The output addresses to store the output maps in memory are generated based on the x, y input pixel coordinates and the kernel index f as

$$\begin{aligned} x_o &= x - (k_x - 1) \\ y_o &= y - (k_y - 1) \\ a_o &= n_x n_y f + n_y y_o + x_o \end{aligned} \tag{28}$$

5. Network

To showcase the performance of the developed modules, we implemented an SNN version of a LeNet-5 network for the MNIST handwritten digits dataset. It is a convenient way to demonstrate an application that fits in a low-power device on the edge. This network has been trained as a classical ANN in Keras and converted into an SNN using the SNN toolbox framework [26].

The network, shown in Figure 12, is made up of several neuron layers, including 2D convolutions, dense layers —also known as fully-connected— and pooling layers, which subsample the input features and average their pixels out to generate smaller feature maps.

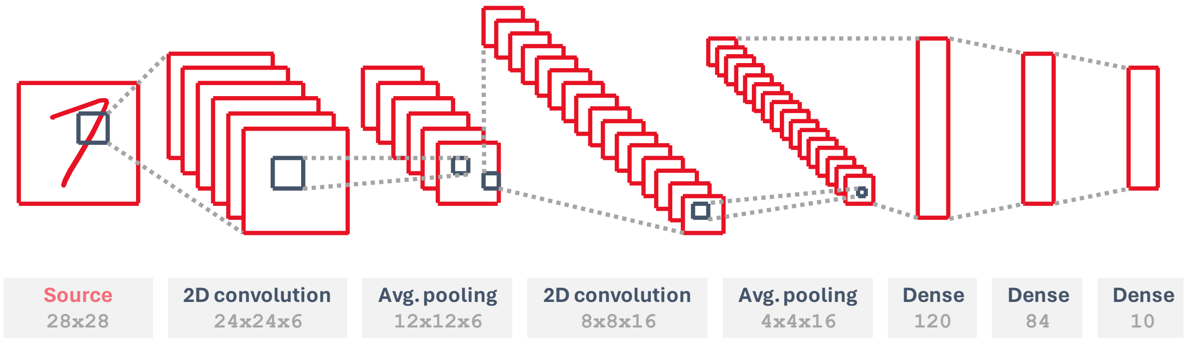

Figure 12. LeNet-5 architecture for a 28x28 MNIST input image and 10 output classes.

The input layer converts input images into Poisson rate-coded spikes. We map all other layers into several cores, using both the fully-connected and 2D convolutional versions. To implement the pooling layers, which have a 2x2 pooling mask, we use a single-input single-kernel 2D convolutional layer with all weights equal to $w = 1/(p_x p_y) = 0.25$ and stride $s = p_x = 2$, aplied as many times as input feature maps in the previous convolutional layer. The rest of them have several sets of trainable weights, whose number is shown in Table 1.

Table 1. Number of neurons, synapses and weights per layer.

	2D conv. (24x24x6)	Pooling (12x12x6)	2D conv. (8x8x16)	Pooling (4x4x16)	Dense (120)	Dense (84)	Dense (10)	**Total**
Neurons	3 456	864	1 024	256	120	84	10	**5 814**
Synapses	86 400	3 456	153 600	1 024	30 720	10 080	840	**286 120**
Weights	150	4	2400	4	30 720	10 080	840	**44 198**

All layers start their processing when the global time step signal arrives, reading their input spikes and storing the generated output spikes back into a series of dual-port memories that chain all layers together, as illustrated in Figure 13. The spike memories are instantiated by the synthesizer, which is free to choose any device primitive. However, due to their relative small size compared to the synaptic weights and the fact that they hold binary values, they are implemented as distributed RAM that can be read and written by all layers at the same time.

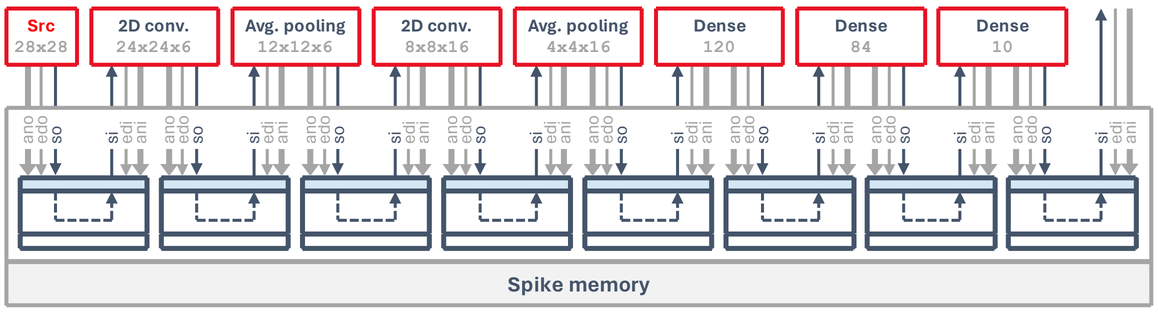

Figure 13. SNN network architecture: input, convolutional, pooling and fully-connected neuron cores attached to the spike memory.

For example, the first 2D convolution reads the 784 bits from the input memory through the *si*, *adi* and *eni* interface. After processing, it writes the 3 456 output bits (binary spikes) to the next memory block through *so*, *ado* and *eno*. The total number of spike memory bits for the whole network is equal to the number of neurons in the network, which in this case is $N = 5\,814$.

Regarding weight and state memory usage, the total number of bits is given by

$$m = \underbrace{N \cdot \text{bits/state}}_{\text{Total state memory}} + \underbrace{W \cdot \text{bits/weight}}_{\text{Total weight memory}} \tag{29}$$

where N is the total number of neurons, W is the total number of weights in the network and m is the memory requirement in bits. For this network, $m = 800\,192\,\text{bit} = 781.4\,\text{kB}$.

The input layer module is loaded with an MNIST image to generate the network input spikes, which are processed by the following layers. The final output spikes are available through the read interface of the last memory block after all the processing is complete.

6. Results

The cores and network have been developed in VHDL using Xilinx Vivado 2022.1 and GHDL, and have been synthesized and tested on a Xilinx Artix-7 XC7A100T FPGA. We opted for a relatively compact device as our focus is on low-power and low-area applications.

Table 2 shows the primitive utilization of the whole network with 16-bit-wide weights. To have as reusable a design as possible, all HDL code has been written using no vendor or device-specific instances. Primitive inference is then left to the synthesizer, which has control over what type of memory it instantiates for weight, neuron state and spike storage. It can be seen that different layers use different elements: the convolutional and pooling layers rely on a mix of distributed memory, LUTs and block RAM, while the two biggest fully-connected ones use up the available block RAM tiles. We have seen that this distribution is consistent even in bigger FPGAs with more available memory, so block RAM utilization is not limited by device size but rather by the synthesis optimization process.

Table 2. FPGA resource utilization for a LeNet-5 architecture with 16-bit wide weights *(Artix-7 XC7A100TCSG324-1)*.

	Slice LUTs	Slice Registers	Slices	LUT as Logic	LUT as Memory	Block RAM [1]
Input	21	38	13	21	0	0
2D conv. (24x24x6)	564	709	184	547	17	2.5
Pooling (12x12x6)	98	129	51	87	11	1
2D conv. (8x8x16)	3420	3984	1044	2973	447	0
Pooling (4x4x16)	81	117	42	67	14	0.5
Dense (120)	5890	7184	1986	5878	12	68.5
Dense (84)	2128	3062	789	2118	10	60.5
Dense (10)	2051	2783	654	2033	18	0
Spike Memory	15	4	6	1	14	2
Total	**14266**	**18010**	**4744**	**13723**	**543**	**135**
Total (%)	22.5 %	14.2 %	29.9 %	21.7 %	2.9 %	100 %
Available	63400	126800	15850	63400	19000	135

[1] Full block RAM tiles are 36-bit wide and can be split, half figures represent 18-bit-wide half tiles.

We have tested the network architecture for different weight quantization bit widths, from 4 to 16 bits. While the cores remain the same with no remarkable difference in resource usage, the weight storage distribution changes and in Figure 14, it can be seen that the synthesizer switches to block RAM for weight storing as bit widths go past 8 bits. None of the implementations uses any of the available DSPs in the FPGA.

Using switching activity data from post-implementation simulations we have done a power analysis to produce some estimated energy consumption figures, which can be found in Table 3. Shown in Figure 15, we can see this switch to RAM drives energy consumption up from to 0.244 W for 4 bits to 0.348 W for 16 bits.

Table 3. Power data and max. frequency values for weight bit widths from 4 to 16 bits.

Bits	Logic	Signals	Clocks	BRAM	Dynamic	Static	**Total[1]** [W]	Max. freq. [MHz]
4	0.010	0.011	0.074	0.008	0.103	0.141	**0.244**	141.80
6	0.019	0.017	0.070	0.008	0.114	0.141	**0.254**	148.96
8	0.016	0.018	0.080	0.052	0.166	0.151	**0.317**	132.53
12	0.010	0.019	0.075	0.092	0.196	0.151	**0.347**	134.31
16	0.010	0.019	0.076	0.092	0.197	0.151	**0.348**	139.31

[1] Power estimation data calculated with a 100 MHz system clock and $T = 25\,^{\circ}\mathrm{C}$.

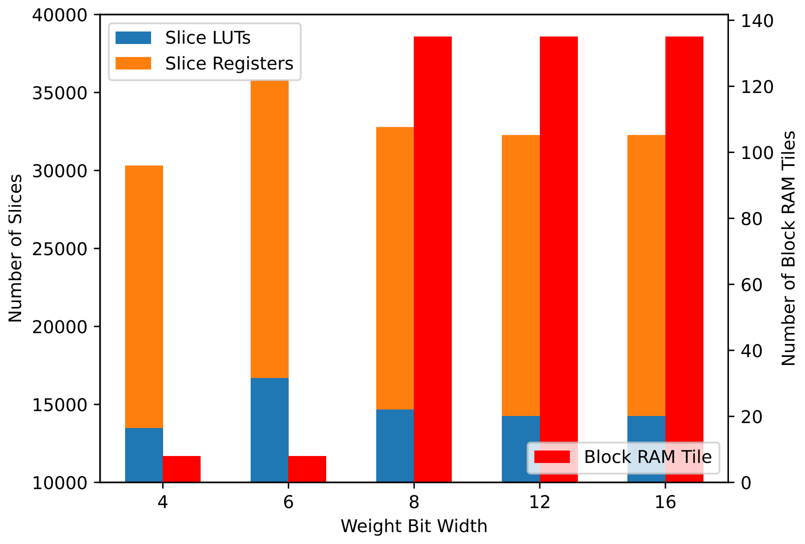

Figure 14. Slice LUT, register and block RAM utilization.

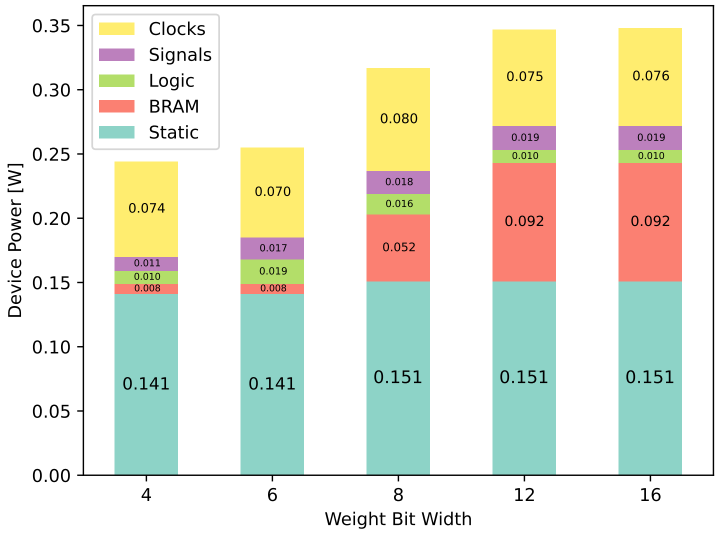

Figure 15. Power estimation data @ 100 MHz and $T = 25\,^{\circ}$C.

6.1. Latency and Minimum Timestep

Since all layers process data simultaneously, the minimum timestep the network simulation can achieve is dependent on the maximum latency among all of them. In this case, the longest processing time is reached by the second 2D convolutional layer, mostly because of the number of kernels and kernel size. At 13 978 cycles, it translates to $t_{step} = 105.9\,\mu s$ for a system clock frequency of 132 MHz.

Both fully-connected and convolutional cores can take advanatage of the data independence between neurons to be parallelized. This could be exploited to split each layer into several cores, each of which would process a subset of the neurons in that layer. This way we could shorten the maximum latency values, trading in area for speed to achieve shorter simulation time steps, or to fit larger layers in a lower-speed device.

6.2. Benchmarking

Table 4 shows a comparison between the previously mentioned state-of-the-art FPGA SNN accelerators. We considered a representative set of relevant implementations based on LIF neurons. It can be seen that, while block RAM and distributed memory usage is similar, our cores manage to reach the lowest resource utilization per neuron among the considered architectures, at at 2.45 and 3.10 LUTs and FFs per neuron respectively. Our minimum step time of 105.9 µs is small enough to fit the lowest times commonly employed in these simulations, which are usually in the hundreds of microseconds, and the whole network runs at only 348 mW or 59.86 µW/neuron when using the largest weight and state bit widths, making it the least power-hungry implementation.

Table 4. Comparison of relevant state-of-the-art FPGA SNN hardware accelerators.

	Minitaur [18]	**Wang et al.** [22]	**Darwin** [20]	**Gupta et al.**[21]
Year	2014	2016	2017	2020
Device	XC6SLX150	XC6VLX240T	XC6SLX45	XC6VLX240T
Clock	75 MHz	120 MHz	25 MHz	100 MHz
Algorithm	Event	Clock	Event	Event
Model	LIF	LIF	LIF	Simplified LIF
Arch.	784-500-500-10	-	784-500-500-10	784-16
Weights	Off-chip DRAM	On-chip BRAM	Off-chip DRAM	On-chip BRAM
Bit width	16/16 bit	-/8 bit	32/16 bit	24/24 bit
Neurons	1785	1591	1794	800
Synapses	647000	638208	647000	12544
BRAM	200	-	72	16
DSP	-	-	32	64
LUTs	-	69265	27288	56230
FFs	-	50688	54576	23238
Slices	22000	119953	81864	79468
LUTs/neuron	-	43.54	15.21	70.29
FFs/neuron	-	31.86	30.42	29.05
Slices/neuron	12.32	75.39	45.63	99.34
Power	1.5 W	1.07 W	-	-
Power/neuron	840.34 µW	672.53 µW	-	-
	E3NE [24]	**NHAP** [23]	**Spiker** [25]	**This work**
Year	2021	2022	2022	2024
Device	XCVU13P	XC7K325T	XC7Z020	XC7A100T
Clock	200 MHz	200 MHz	100 MHz	100 MHz
Algorithm	Clock	Mixed	Clock	Clock
Model	LIF	LIF/IZ	LIF	Simplified LIF
Arch.	28x28-6c5-p2-16c5-p2-120-84-10	1024-1024-10	784-400	28x28-6c5-p2-16c5-p2-120-84-10
Weights	On-chip BRAM	Both	On-chip BRAM	On-chip BRAM
Bit width	-/3 bit	16/16 bit	16/16 bit	16/16 bit
Neurons	5814	2058	1384	5814
Synapses	286120	1059600	313600	286120
BRAM	-	83.5	45	135
DSP	0	64	0	0
LUTs	27000	12218	29145	14266
FFs	24000	10325	26853	18010
Slices	51000	22543	55998	32276
LUTs/neuron	4.64	5.94	21.06	**2.45**
FFs/neuron	4.13	5.02	19.40	**3.10**
Slices/neuron	8.77	10.95	40.46	**5.55**
Power	1.2 W	0.535 W	-	**0.348** W
Power/neuron	206.40 µW	259.96 µW	-	**59.86** µW

Making comparisons between FPGA-based SNN implementations is not straightforward. Almost every one of them runs similar realizations of the Leaky-Integrate-and-Fire neuron model, but they do so on many different FPGA platforms with largely varying available resources and speed capabilities. To make things worse, they implement different network architectures with different quantization levels and memory organization, which significantly impact the implementation results. For instance, a network with more convolutional layers, will see a decrease in LUTs and FFs per neuron since many neurons are simulated using a single set of kernel weights.

More importantly, state-of-the-art design analyses usually put their emphasis on network accuracy or fitness, which are application-derived metrics rather than architectural ones. Since our focus is on hardware optimization, we have provided as much characterization data as possible to compensate for the lack of algorithmic-level comparative figures. Our goal is to construct a circuit architecture that can accommodate any high-level description, independent of the neural model and network topology —which will determine the accuracy-related metrics—, while being optimized for efficient and light-weight close-to-the-edge systems. To achieve ultra-low area and power consumption, as well as low simulation step times, we leverage the strengths of clock-driven architectures and employ two key strategies: extensive and extreme hardware pipelining, specially convenient for FPGA platforms, distributed memory to increase weight access bandwidth, and a fixed connectivity scheme that relies on efficient spike memory banks.

7. Conclusions

This work introduced a new architecture for acceleration of SNNs, using clock-driven neuron processing cores on FPGAs. It is able to reach fast simulation step times of 105.9 µs and, using 16-bit-wide neuron states and weights, it has the lowest device resource utilization among all compared works, at 5.55 slices per neuron respectively, as well as the lowest power figure at 59.86 µW per neuron.

It is based on the concept of neuron core as a unit capable of simulating groups of spiking neurons, which:

- Takes advantage of pipelining and neuron data independence to accelerate simulations and reduce hardware usage, using around 33% less slices per neuron than the best current implementation.
- Achieves very low-power operation, nearly 4 times as less power per neuron as the most efficient state-of-the-art accelerator.
- Manages to keep simulation step times low, which contributes to making more accurate simulations and leaves room for bigger networks.
- Makes it easy to map high-level descriptions of SNNs into hardware.

We have defined control subsystems for fully-connected and 2D convolutional layers, spanning from single-map single-kernel instances to fully-fledged multimap 2D convolutions with multiple kernels.

The presented cores are flexible circuits designed to run arbitrary models. They are meant to be used as building blocks in automatic mapping from network description languages to FPGA-based SNN accelerators, enabling designers to easily generate fast and efficient hardware from any supported neuron model. Our simplified LIF neuron is a good starting point for implementing new supported models, performing accuracy tests or developing different connectivity schemes for many types of neuron layers.

Using SNN toolbox for ANN-to-SNN conversion, we have developed an implementation of the LeNet-5 architecture trained on the MNIST handwritten digit dataset to showcase our convolutional and fully-connected cores.

We are in the process of developing a complete software module for SNN HDL generation that relies on our core architecture. It is a Python-based framework based on VHDL template instantiation, mapping high-level network descriptions written either in PyNN or our intermediate representation interface into HDL. It enables quick and easy testing with varying parameters, neuron models and quantization levels, being able to give performance and accuracy values on real hardware tailored for close-to-the-edge applications on low-resource devices.

Author Contributions: Conceptualization, S.L. and P.I.; methodology, S.L.; software, S.L.; validation, S.L. and P.I.; formal analysis, S.L.; investigation, S.L.; resources, S.L. and P.I; data curation, S.L.; writing—original draft preparation, S.L. and P.I.; writing—review and editing, S.L. and P.I.; visualization, S.L.; supervision, P.I.; project administration, P.I.; funding acquisition, S.L. and P.I. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the MCIN/AEI/10.13039/501100011033 Spanish Ministry of Science and Innovation within projects PID2022-141391OB-C21, PDC2022-133657-I00, and grant PGC2018-097339-B-I00.

Data Availability Statement: The data is available on request to the corresponding authors.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Maass, W. Networks of spiking neurons: The third generation of neural network models. *Neural Networks* **1997**, *10*, 1659–1671.
2. Kasabov, N.K. *Time-Space, Spiking Neural Networks and Brain-Inspired Artificial Intelligence*; Springer Berlin Heidelberg, 2019.
3. Yamazaki, K.; Vo-Ho, V.K.; Bulsara, D.; Le, N. Spiking Neural Networks and Their Applications: A Review. *Brain Sciences* **2022**, *12*, 863.
4. Frenkel, C.; Bol, D.; Indiveri, G. Bottom-Up and Top-Down Neural Processing Systems Design: Neuromorphic Intelligence as the Convergence of Natural and Artificial Intelligence, 2021.
5. Bogdan, P.A.; Marcinnò, B.; Casellato, C.; Casali, S.; Rowley, A.G.; Hopkins, M.; Leporati, F.; D'Angelo, E.; Rhodes, O. Towards a Bio-Inspired Real-Time Neuromorphic Cerebellum. *Frontiers in Cellular Neuroscience* **2021**, *15*.
6. Pei, J.; Deng, L.; Song, S.; Zhao, M.; Zhang, Y.; Wu, S.; Wang, G.; Zou, Z.; Wu, Z.; He, W.; Chen, F.; Deng, N.; Wu, S.; Wang, Y.; Wu, Y.; Yang, Z.; Ma, C.; Li, G.; Han, W.; Li, H.; Wu, H.; Zhao, R.; Xie, Y.; Shi, L. Towards artificial general intelligence with hybrid Tianjic chip architecture. *Nature* **2019**, *572*, 106–111.
7. Indiveri, G. Computation in Neuromorphic Analog VLSI Systems. In *Perspectives in Neural Computing*; Springer London, 2002; pp. 3–20.
8. Davidson, S.; Furber, S.B. Comparison of Artificial and Spiking Neural Networks on Digital Hardware. *Frontiers in Neuroscience* **2021**, *15*.
9. Basu, A.; Deng, L.; Frenkel, C.; Zhang, X. Spiking Neural Network Integrated Circuits: A Review of Trends and Future Directions. 2022 IEEE Custom Integrated Circuits Conference (CICC), 2022, pp. 1–8.
10. Golosio, B.; Tiddia, G.; Luca, C.D.; Pastorelli, E.; Simula, F.; Paolucci, P.S. Fast Simulations of Highly-Connected Spiking Cortical Models Using GPUs. *Frontiers in Computational Neuroscience* **2021**, *15*.
11. Furber, S.B.; Lester, D.R.; Plana, L.A.; Garside, J.D.; Painkras, E.; Temple, S.; Brown, A.D. Overview of the SpiNNaker System Architecture. *IEEE Transactions on Computers* **2013**, *62*, 2454–2467.
12. Akopyan, F.; Sawada, J.; Cassidy, A.; Alvarez-Icaza, R.; Arthur, J.; Merolla, P.; Imam, N.; Nakamura, Y.; Datta, P.; Nam, G.J.; Taba, B.; Beakes, M.; Brezzo, B.; Kuang, J.B.; Manohar, R.; Risk, W.P.; Jackson, B.; Modha, D.S. TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **2015**, *34*, 1537–1557.
13. Davies, M.; Srinivasa, N.; Lin, T.H.; Chinya, G.; Cao, Y.; Choday, S.H.; Dimou, G.; Joshi, P.; Imam, N.; Jain, S.; Liao, Y.; Lin, C.K.; Lines, A.; Liu, R.; Mathaikutty, D.; McCoy, S.; Paul, A.; Tse, J.; Venkataramanan, G.; Weng, Y.H.; Wild, A.; Yang, Y.; Wang, H. Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro* **2018**, *38*, 82–99.
14. Deng, L.; Wang, G.; Li, G.; Li, S.; Liang, L.; Zhu, M.; Wu, Y.; Yang, Z.; Zou, Z.; Pei, J.; Wu, Z.; Hu, X.; Ding, Y.; He, W.; Xie, Y.; Shi, L. Tianjic: A Unified and Scalable Chip Bridging Spike-Based and Continuous Neural Computation. *IEEE Journal of Solid-State Circuits* **2020**, *55*, 2228–2246.
15. Pham, Q.T.; Nguyen, T.Q.; Hoang, P.C.; Dang, Q.H.; Nguyen, D.M.; Nguyen, H.H. A review of SNN implementation on FPGA. 2021 International Conference on Multimedia Analysis and Pattern Recognition (MAPR). IEEE, 2021.
16. Davison, A.; Brüderle, D.; Eppler, J.; Kremkow, J.; Muller, E.; Pecevski, D.; Perrinet, L.; Yger, P. PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics* **2009**, *2*.

17. Brette, R.; others. Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of Computational Neuroscience* **2007**, *23*, 349–398.
18. Neil, D.; Liu, S.C. Minitaur, an Event-Driven FPGA-Based Spiking Network Accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **2014**, *22*, 2621–2628.
19. Mo, L.; Tao, Z. EvtSNN: Event-driven SNN simulator optimized by population and pre-filtering. *Frontiers in Neuroscience* **2022**, *16*.
20. Ma, D.; Shen, J.; Gu, Z.; Zhang, M.; Zhu, X.; Xu, X.; Xu, Q.; Shen, Y.; Pan, G. Darwin: A neuromorphic hardware co-processor based on spiking neural networks. *Journal of Systems Architecture* **2017**, *77*, 43–51.
21. Gupta, S.; Vyas, A.; Trivedi, G. FPGA Implementation of Simplified Spiking Neural Network. 2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS). IEEE, 2020.
22. Wang, Q.; Li, Y.; Shao, B.; Dey, S.; Li, P. Energy efficient parallel neuromorphic architectures with approximate arithmetic on FPGA. *Neurocomputing* **2017**, *221*, 146–158.
23. Liu, Y.; Chen, Y.; Ye, W.; Gui, Y. FPGA-NHAP: A General FPGA-Based Neuromorphic Hardware Acceleration Platform With High Speed and Low Power. *IEEE Transactions on Circuits and Systems I: Regular Papers* **2022**, *69*, 2553–2566.
24. Gerlinghoff, D.; Wang, Z.; Gu, X.; Goh, R.S.M.; Luo, T. E3NE: An End-to-End Framework for Accelerating Spiking Neural Networks With Emerging Neural Encoding on FPGAs. *IEEE Transactions on Parallel and Distributed Systems* **2022**, *33*, 3207–3219.
25. Carpegna, A.; Savino, A.; Carlo, S.D. Spiker: an FPGA-optimized Hardware accelerator for Spiking Neural Networks. 2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). IEEE, 2022.
26. Rueckauer, B.; Lungu, I.A.; Hu, Y.; Pfeiffer, M.; Liu, S.C. Conversion of Continuous-Valued Deep Networks to Efficient Event-Driven Networks for Image Classification. *Frontiers in Neuroscience* **2017**, *11*.