# Designing and Accelerating Spiking Neural Network based on High-level Synthesis

Heng Zi
Beijing University of Posts and Telecommunications, Beijing 10013, China
Email: ziheng@bupt.edu.cn

Kang Zhao
Beijing University of Posts and Telecommunications, Beijing 10013, China
Email: zhaokang@bupt.edu.cn

Wei Zhang
The Hong Kong University of Science and Technology, Hong Kong 810000, China
Email: eeweiz@ust.hk

2024 Conference of Science and Technology for Integrated Circuits (CSTIC) | 979-8-3503-6219-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/CSTIC61820.2024.10531920

***Abstract* —The purpose of this paper is to use an advanced spiking neural network algorithm, and accelerate the algorithm with a variety of acceleration methods based on the high-level synthesis platform. Its target is to solve the problem of insufficient performance of the traditional computing platform in real-time electrocardiogram data processing. The experiment shows that the performance is improved significantly, which proves that the proposed method is effective in real-time data processing.**

***Keywords—spiking neural network, electrocardiogram data, acceleration, high-level synthesis***

## I. INTRODUCTION

Spiking Neural Networks (SNN) represents the neural network models inspired by the biological nervous systems. And the central premise of SNN is that neurons communicate with other neurons via voltage action potentials (so-called spikes). The information between neurons is encoded using the time between spikes (frequency) and their amplitudes (weights) [2]. This spike-driven (or event-driven) nature of SNN allows them to consume less energy than other similar networks because communication and synchronization occur only when neurons are spiking. The application of this neural network will play an important role in the processing of large amounts of data. This is especially true in situations where high energy efficiency and performance are required, as they can reduce unnecessary calculations and communications.

ECG (electrocardiogram) data processing is a tedious and time-consuming project. The application of SNN to ECG data processing can improve the real-time and accuracy of heart disease monitoring. However, with the recent increase in power consumption, traditional computing often cannot meet the requirements of strict real-time performance and high real-time performance. Our research focuses on the development of an innovative SNN algorithm and acceleration method designed to meet the problem of insufficient performance when processing ECG data [1].

Field programmable gate arrays (FPGA) is an effective way to accelerate different applications. FPAGs are emerging as a viable alternative to modern CPUs and GPUs. The main benefit of FPGA is that its architecture can be used to applications, such as reducing computational accuracy or streaming intermediate data, rather than expensive communication through external memory.

FPGA has traditionally been programmed using complex low-level hardware descriptive languages (such as VHDL, Verilog), and today FPGA development is assisted by high-level synthesis (HLS) tools. HLS tool allows programmers to describe their applications in a programming language and can automatically convert the application to hardware.

This work was supported in part by National Key R&D Program of China (2022YFB2901100).

In this paper, FPGA is applied to this research and HLS tool is used to simplify the development process of FPGA. A SNN algorithm based on full connection layer function is proposed, and then the SNN algorithm is accelerated step by step on the HLS platform, the performance is improved successfully.

The rest of the paper is organized as follows. In Section Ⅱ, the background of the whole process of SNN is described. Then, Section Ⅲ presents the detailed analysis on SNN algorithm for a better acceleration. And then the acceleration method is proposed in Section Ⅳ. Section Ⅴ introduces the experimental results . Finally, a conclusion is made in Section VI.

## Ⅱ. BACKGROUND

The neurons in SNN mimic biological neurons connect to other neurons via axons and produce spikes when fired. The fully connected layer indicates that in a certain layer of the neural network, there are 32 input neurons and 32 output neurons. The connections between them are fully connected, and each connection has a weight to adjust the connection strength. This layer of the neuron models combines with the concept of biological neurons, enables neural networks to learn and process complex data patterns and features [3].

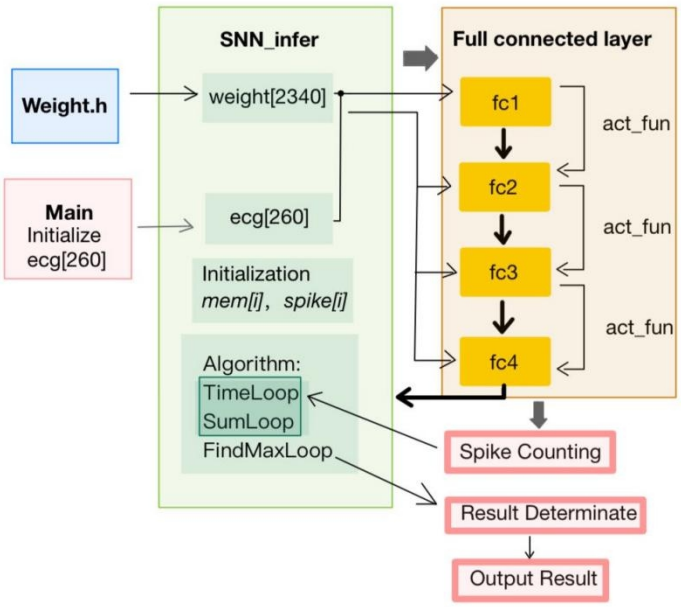


Fig. 1. The inference process based on SNN.

As shown in Fig1, the *SNN_infer* function is the key part of the reasoning process. It receives the weight data,

the ECG data, and initializes values (membrane potential (*mem[i]*), *spike[i]*, etc.). It includes the output of the last fully connected layer, and contains three important algorithms. The ECG data is transferred from the main function to the neural network for processing, and combined with the weight data for processing. This is the starting point of the whole process. The input ECG data is then fed into the important full connection layer, which in turn passes through four full connection layer functions: *fc1, fc2, fc3*and *fc4*. Each function performs different computational steps to pass the feature data to the next layer. Each fully connected layer includes the updating of the membrane potential, the judgment of pulse state and the application of weights, and updates the membrane potential of neurons according to the connected weights. And the activation function is used in this case to simulate the activation of neurons.

Then the spike counting is performed on the output results of the fully connected layer. During the reasoning process, neurons will generate pulses. By counting the number of pulses per output neuron, it is possible to understand the activity of the network. And then according to the statistical results of the number of pulses, the inference result is determined. The output neuron with the highest number of pulses corresponds to the final classification result. The final inference result is represented as an integer, representing the classification result of the input ECG data. It can be used to determine a particular feature or disease state in the ECG data.

## III. SNN Analysis

### A. Activation Function Analysis

In Fig.2, the code starts with the definition of the activation function used in SNN. This function *act_fun* calculates whether a neuron should generate a spike based on a specified threshold in this case. If the neuron's membrane potential exceeds this threshold, it produces a spike. Otherwise, it remains inactive.

```
1.  inline uint1_t act_fun(float24_t mem) {
2.  return mem > THRESHOLD;   }
```

Fig. 2. The function *act_fun* determines the pulse signal generation

### B. Array Analysis

Arrays play a very important role in the SNN algorithm, so analyzing the detailed functions of the following four arrays is necessary. The array *weight[i]* is a 2340-element float24_t array that stores the weight parameters of the neural network. These parameters are usually pre-trained and imported from an external file. In order to input into the spike neural network to inference, the array *ecg[260]* is used to store 260 data points of ECG (electrocardiogram) data . And the array *mem[i]* is used to store the membrane potential of neurons, which correspond to each of the four fully connected layers. These arrays are updated at each time step according to different calculations. And the *spike[i]* are arrays that store whether the neuron produces a pulse, which is the output of the activation function *act_fun*. They are updated at each time step based on the membrane potential and activation function.

### C. Loop Analysis

1. *TimeLoop*: In Fig.3, his function enters a time loop that represents the time processing of the ECG data. It iteratively calls the fully connected layers (*fc1, fc2, fc3* and *fc4*) to update neuron states and membrane potentials based on the input data, the synaptic weight, and the spike history.

```
    TimeLoop:
1.  for(int time=0; time<130; time++){
2.      fc1(ecg[time], ecg[time+130], mem1, spike1, weight);
3.      fc_layer(spike1, mem2, spike2, weight[96], 32, 32); }
```

Fig. 3. Part of code to call the full connection layer functions in turn.

2. *SumLoop*: In Fig.4, count the number of pulses of each output neuron and find the code part of the neuron with the most pulses to determine the final classification or result of the input data.

```
    SumLoop:
1.  for(i=0; i<4; i++){
2.   #pragma HLS PIPELINE II=2 off rewind
3.        spike_sum[i] += spike_out[i];      }
```

Fig. 4. Count the number of pulses per output neuron.

3. *FindMaxLoop*: In Fig.5, the role is to find the output neuron with the largest average number of pulses, and then store its index in result as the final result of the neural network's reasoning.

```
    FindMaxLoop:
1.      for(i=0; i<4; i++){
2.          cout << spike_sum[i]/130.0 << ' ';
3.           if(max<spike_sum[i]){
4.           max = spike_sum[i];
5.               *result = i;       }
```

Fig. 5. Find the output neuron with the most pulses and determine it as the result.

Further analysis: It is obvious that there is a data dependency between the *Timeloop* and *SumLoop* because the four fully connected layer functions in the *TimeLoop* depend on the result of the previous time step, so the *SumLoop* needs to wait for these functions to complete before calculating the pulse sum. In contrast, the *FindMaxLoop* has no obvious data dependency issues. Each iteration of the loop, it independently compares the total number of pulses of the four output neurons and updates the max and result. This means that it can be executed in parallel between each iteration of the loop without being affected by the results of the previous iteration.

## IV. Acceletration

### A. Function Inline on *act_fun*

Function inline can help the compiler do more compiling optimizations such as data dependency analysis and alias analysis, reducing the overhead of function calls.

### B. Performance Optimizations

1. We pipelined the processing cycle with the

**"#pragma HLS PIPELINE II=2 off rewind"** directive to improve performance. For the *SumLoop* loop, *PIPELINE* indicates that it is pipelined so that iterations of the loop can be executed in parallel. Here "II=2" means that an iteration is performed every 2 clock cycles, which speeds up the execution of the loop. off and rewind are used to deactivate the rollback feature to ensure that the pipe does not cause an error.

2. The "**#pragma HLS INLINE**" directive in *Timeloop* is used to tell the compiler to expand function calls inline as much as possible at compile time to reduce the overhead of function calls. In this code, it is applied to call the function *fc1* and function *fc_layer* to optimize the use of hardware resources.

3. We applied the "**#pragma HLS UNROLL**" directive to reset the *spike_sum*, telling the compiler to expand the loop when generating the hardware, that is, to expand the statements in the loop into multiple copies for parallel execution. In this case, the loop is expanded into four separate reset operations, each responsible for the reset of one *spike_sum* array element.

### C. Combining *fc2, fc3, fc4* into *fc_layer*

Researchers merged the three *fc_functions* and replace the call of the original function in the *SNN_infer* function. Since the *fc2, fc3* and *fc4* have the same code structure, in Fig. 6, we combined these three functions into a single *fc_layer* function to reduce redundancy and increase flexibility.The new *fc_layer* function takes the parameters *in_size* and *out_size* to determine the size of the input and output, making it adaptable to different layers. This increases the versatility of the code and makes it easy to deal with network structures of different sizes without having to write separate functions for each layer.

```
1.  void fc_layer(uint1_t spike_in[32], float24_t *mem,
2.      uint1_t *spike_out, const float24_t *weight, int
3.      in_size, int out_size) {
4.  for(int i=0; i<out_size; i++){
5.      mem[i] = mem[i] * DECAY * (1-spike_out[i]);
6.  for(int j=0; j<in_size; j++){
7.      mem[i] += spike_in[j]*weight[j+i*in_size]; }
8.      mem[i] += weight[in_size*out_size + i];
9.      spike_out[i] = act_fun(mem[i]);  }
```

Fig. 6. The *fc_layer* function after combining.

### D. Resetting the array *spike_sum*

At the end of the function *SNN_infer*, one loop is reported to reset the array of spikes' sum (*spike_sum[i]*) in the preparation for the next call.

```
1.    for(int i = 0; i < 4; i++) {
2.      #pragma HLS UNROLL
3.            spike_sum[i] = 0;    }
```

Fig. 7. Reset the code for the array *spike_sum*.

## V. Experiment

The experiment is performed on Xilinx Vitis HLS 2023.1 running on the Ubuntu Linux 20.04 LTS operating system. And Xilinx Vivado 2023.1 is also used to evaluate the resources after routing. During the simulation, the clock period is set as 10ns.

TABLE I
The performance estimations after each acceleration methods.

| Step | Latency (cycles) | Interval | BRAM | DSP | FF | LUT |
|---|---|---|---|---|---|---|
| A | 311494 | 311495 | 24 | 4 | 7676 | 21612 |
| B | 307464 | 307465 | 24 | 4 | 7664 | 21492 |
| C | 307724 | 307325 | 18 | 3 | 5415 | 15461 |
| D | 307723 | 307325 | 18 | 3 | 5409 | 15461 |

In Table I, as we mentioned in Section III, the performance gets improvement gradually during four steps. The step-*A* stands for declaring the *act_fun* function with the **inline** attribute, the step-*B* uses pragma/directive for acceleration, the step-*C* stands for merging three functions into one function, and *D* means resetting the array *spike_sum*.

The performance/resource estimations have been shown in Table I. In four acceleration methods from *A to D*, not only the initial reduction in latency and interval are observed, but more importantly, a significant reduction in resource usage is also presented. Especially on flip-flops (FF) and lookup tables (LUT), the reductions are significant, highlighting the efficiency and effectiveness of the proposed method. The four optimizations have brought higher efficiency and lower resource consumption to the FPGA design, which has been proved to be a successful performance improvement.

## VI. Conclusion

The SNN model shown in this paper is useful for processing time series data, such as electrocardiogram, especially when the real-time classification is required. The results show that the performance has been significantly improved. We believe that the development of this technology can also continue to meet the needs of other applications, providing a potential optimization strategy for the processing of real-time data.

## References


[1] S. Kiranyaz, T. Ince and M. Gabbouj, "Real-time patient-specific ECG classification by 1-D convolutional neural networks", IEEE Trans. Biomed. Eng., vol. 63, no. 3, pp. 664-675, Mar. 2016.

[2] A. Podobas and S. Matsuoka, "Designing and accelerating spiking neural networks using OpenCL for FPGAs," 2017 International Conference on Field Programmable Technology (ICFPT), Melbourne, VIC, Australia, 2017, pp. 255-258, doi: 10.1109/FPT.2017.8280154.

[3] Fwb04 (2023, Mar 14). "SNN_HLS" [Online]. GitHub. Available: https://github.com/fwb04/SNN_HLS.git.