AN FPGA-BASED NEURAL NETWORK FOR COMPUTER VISION APPLICATIONS

Robin Pottathuparambil and Ron Sass

Reconfigurable Computing Systems Lab, University of North Carolina at Charlotte, NC 28223
Email: {rpottath,rsass}@uncc.edu

ABSTRACT
A neural network is an information processing system that is widely used in various computer vision applications. This paper discusses an low-latency FPGA-based neural network implementation that does direct computation of the sigmoid activation function.

1. INTRODUCTION
Computer vision is a branch of science that deals with extraction of information from images or sequence of images to perform some task. These images or sequence of images are processed with intelligent computing systems to extract the required features. These extracted features are then compared with a set of stored features to further study the image or analyze the image. One of the most common techniques used for comparison is Neural Networks (NN).

NN-based applications are implemented either using software or using a hardware platform. These hardware-based NNs are faster than software-based NNs and are either built using (Application Specific Integrated Circuits) ASICs or (Field Programmable Gate Arrays) FPGAs. The major drawback of an ASIC-based design is the cost of re-fabrication when a change in the NN design is desired. On the contrary, most FPGA-based NN designs are built using a simple look-up table method and any change in the NN design requires a simple re-configuring of the new design.

A simple NN is shown in Fig. 1 (a) where each node’s output behavior is described by an activation function. One of the common and simple activation function is the sigmoid function because of the fact that their response is similar to that of a biological neuron. The evaluation of the sigmoid activation function requires an exponential ($e^{-x}$) computation. These exponential functions when implemented in software are slower than hardware-based implementations. However, in a hardware-based implementation, the exponential function is either approximated using a look-up table or computed at reduced precision. In both cases, the accuracy of the results are lost. The look-up table method consumes on-chip memory resources and when more exponential blocks are required, large on-chip memory resources are used. In order to overcome the large on-chip memory usage, a percentage of off-chip memory bandwidth is also used.

The work discussed in the paper describes an FPGA-based NN whose number of node inputs and the number of nodes in a layer can be easily modified. The NN discussed in the paper is implemented using a double-precision sigmoid activation function which is computed using a direct computation technique called COordinate Rotation Digital Computer (CORDIC). The results from the double-precision exponential core is very accurate and reduces on-chip memory and uses no off-chip memory bandwidth. The latency numbers of the FPGA-based NN design is compared with a ‘C’ implementation. The rest of the paper is organized as follows. Section II gives a brief summary of the existing NN computation techniques. Section III describes the activation function and its design. Section IV compares the results with a CPU-based NN design and the paper concludes with future directions.

2. RELATED WORK
The most common techniques in the literature for computing exponential function ($e^{-x}$) of a sigmoid activation function are CORDIC, table-driven, and polynomial approximation methods. Most of the CORDIC implementations in the literature are either low precision CORDIC [1], or approximation of low-precision CORDIC computation to a double-precision exponential value [2]. The table-driven method discussed in the literature [3] uses look-up table to compute exponential values. In polynomial approximation method, the exponential values are computed using look-up tables and a short Taylor series expansion [4]. Most of the techniques discussed in the literature are either approximation of exponential value or single-precision computation.

The sigmoid activation functions in the literature for NN, are either sigmoid approximators or direct sigmoid implementations. Sigmoid approximators use look-up table to compute sigmoid functions. A 16-bit fixed-point log-sigmoid function implementation using CORDIC is discussed in [5]. Huajan et al. discusses the implementation of the sigmoid function using direct and indirect CORDIC computations [6]. These computations are all fixed-precision computations. Most of the techniques discussed in the literature are either fixed-precision computation or sigmoid approximation. Our work discusses a CORDIC based double-precision floating-point based sigmoid activation function which is...
Fig. 1. (a) A simple 3-2-1 Neural Network; (b) Single node of the Neural Network; (c) Hardware blocks used for computing; (d) CORDIC angle computation (e) CORDIC magnitude computation

highly accurate.

3. DESIGN AND IMPLEMENTATION

The NN layer and its nodes were designed using VHDL and synthesized using Xilinx Platform Studio 10.1. Figure 1 (b) shows a node in the NN. The function \( f(x) \) in Fig. 1 (b) is the sigmoid activation function \( (1/(1+e^{-x})) \). The output of a node is activated when the value of the activation function reaches the threshold value set by ‘T1’. The hardware block diagram of a NN layer is shown in Fig. 1 (c). The number of node inputs in a layer can be easily modified by changing the latency of the first adder. The number of nodes in a layer can be increased by providing the input weight values of the new node and its threshold value. The value ‘\( x \)’ in Fig. 1 (c) after being buffered (because the exponential block has a latency of 64 clock cycles for every 4 inputs) in a FIFO is multiplied by 1/Ln(2) and converted into a fixed-point format (1-bit sign, 6-bit integer and 53-bit fraction). The fixed-point value \( x_{fp} \) is then split into integer \( (x_{int}) \) and fractional \( (x_{frac}) \) values. The fractional value \( x_{frac} \) is then multiplied by Ln(2) to get \( x_f (0 \leq x_f \leq \text{Ln}(2)) \). These values of \( x_{int} \) and \( x_f \) are used to compute the transformed equation of \( e^{-x} = 2^{-x_{int}} \cdot e^{-x_f} \). The value of \( e^{-x_f} \) is then computed using the CORDIC technique as shown in Fig. 1 (d) and (e). The CORDIC computation is done in parallel for four input values. The CORDIC computation involves several iterations. In our case it is 60 iterations (53-bit + 7 guard bits). Once the iterations are done, the computed values (four) are available by setting the switch S4 and S5. As soon as the CORDIC results are available, these values are converted back to double-precision floating-point values and the final values are computed by multiplying \( 2^{-x_{int}} \) (subtracting \( x_{int} \) from the mantissa). The computed exponential value is then added with ‘1’ and then an inverse is computed using a division block as shown in Fig. 1. (c). The output of division is then compared with the threshold value \( (T_1, T_2,T_n) \) to determine the actual output. The entire computational cores used in the design are fully pipelined except the CORDIC computation.

4. RESULTS

The design shown in Fig. 1 (c) was placed and routed on a Virtex 4 FX60 FPGA for an operational frequency of 100 MHz. The design consumed about 90% slices, 60% BRAM blocks and 100% DSP48 slices. The FPGA design was compared with a ‘C’ implementation running on an AMD Opteron, 2.0 GHz, 1MB cache as shown in Table 1. As the size of NN increases the speed-up factor also increases.

5. CONCLUSION

As a part of the future work, the current design will be modified to train the NN with a training dataset.

6. REFERENCES


Table 1. Latency values for FPGA and CPU implementations

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Inputs</th>
<th>Test Cases</th>
<th>FPGA (ms)</th>
<th>CPU (ms)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-4-1</td>
<td>6</td>
<td>1000</td>
<td>2.35</td>
<td>4.75</td>
<td>2.01x</td>
</tr>
<tr>
<td>7-5-1</td>
<td>7</td>
<td>1000</td>
<td>3.10</td>
<td>7.35</td>
<td>2.36x</td>
</tr>
<tr>
<td>8-6-1</td>
<td>8</td>
<td>1000</td>
<td>3.46</td>
<td>8.38</td>
<td>2.41x</td>
</tr>
</tbody>
</table>