Resource Reduction of BFGS Quasi-Newton Implementation on FPGA using Fixed-Point Matrix Updating

Jia Liu and Qiang Liu
Tianjin Key Laboratory of Imaging and Sensing Microelectronic Technology
School of Microelectronics, Tianjin University
Tianjin, China
Email: {jialiu, qiangliu}@tju.edu.cn

***Abstract*—Quasi-Newton (QN) methods are now widely used for NN training due to their high effectiveness. In practice, the iterative process of the QN methods implemented in software is often very time-consuming. To accelerate the training process, floating-point BFGS-QN implementation has been realized on FPGA. By analyzing the performance of the BFGS-QN implementation, it is found that updating the inverse of approximate Hessian matrix B is the most computation and memory intensive part. Therefore, a fixed-point hardware design of B matrix updating is proposed in this paper. The fixed-point representation could lead to overflow and underflow during the computation, which degrade the convergence performance of the training process. To address the issues, matrix property checking and precision scaling schemes are proposed, giving a tradeoff between resource and precision. The experimental results show that compared with the single-precision floating-point BFGS-QN, the mixed precision BFGS-QN with fixed-point B matrix updating design achieves up to 10.9% LUTs, 20.2% FFs and 18.1% BRAMs reduction, while the training speed is not satisfied.**

I. Introduction

Artificial neural networks (NNs) are information processing systems which have the capability to learn any arbitrary input-output relationships from a set of data. However, NNs still face some difficulties which complicate and limite their application. One of the biggest barriers is neural network training. Typical NNs such as multilayer perceptron (MLP) networks, recurrent networks and wavelet networks, do not represent any physical information unless they are trained with training data. The trained neural networks are often referred to as neural models. Let us assume that the input-output relationship embedded in data is $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$. The training process of a neural network is that the neural network learns the relationship $\hat{\boldsymbol{y}} = \hat{\boldsymbol{f}}(\boldsymbol{x})$ from the training data so that the difference between output $\hat{\boldsymbol{y}}$ of the neural model and the real value $\boldsymbol{y}$ is minimized. Therefore, NN training is an optimization problem, which involves a complex and non-convex objective function [1] . Various optimization methods, such as conjugate gradient (CG) approach, quasi-Newton (QN) algorithms and particle swarm optimization (PSO), have been used to address the complexity of the training.

The QN methods rank among the most efficient methods available and are used extensively in various applications. Several distinct QN methods have been developed in the past. The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method is one of the most important methods of this class. BFGS-QN can effectively avoid round-off error and division by zero [2]. The BFGS-QN algorithm involves an iterative process for generating a sequence of approximations for the inverse Hessian. As the number of iterations is increased, the approximation becomes progressively more accurate. As a result, current NN training is typically operated in a offline process for applications with static training data or fixed operation conditions. However, when new data is being added continuously or the operating conditions are dynamic, online training becomes necessary and there is a critical need for efficient training in a limited time period [3] and resource availiability especially for embedded systems.

With the increasing density and large amounts of embedded arithmetic blocks, modern high-capacity Field Programmable Gate Arrays (FPGAs), which can operate at frequency up to hundreds of MHz, have been considered to be a more attractive alternative for high performance scientific computing. There are a number of research works on accelerating optimization algorithms using FPGAs, such as Jacobi [4], [5], least mean square [6], and conjugate gradient [7]. By customizing architecture around algorithms, FPGA-based hardware implementations achieve speed much faster than software implementations. In [8] and [9], backpropagation (BP) learning algorithm was implemented on FPGA for NN online training. Although widely used, the BP algorithm converges slowly, since it is essentially a steepest descent method. In [10], the Davidon-Fletcher-Powell quasi-Newton (DFP-QN) algorithm is implemented on FPGA to accelerate the NN training.

This paper aims at a hardware implementation of the BFGS-QN algorithm on FPGAs. We analyze the data flow of the BFGS-QN algorithm and partition the algorithm into computation modules. The detailed evaluation reveals that (a) $\boldsymbol{B}$ matrix updating module takes a large amount of the computational resources, and (b) $\boldsymbol{B}_{\boldsymbol{n}\times\boldsymbol{n}}$ requires the storage space n^2. To reduce the resource usage, we design the $\boldsymbol{B}$ updating module with fixed-point (FXP) arithmetic logic. The area reduction allows the BFGS-QN algorithm to be implemented on small

FPGAs to further reduce cost.

The main contributions of the work are summarized below.

- A FXP hardware design of the $\boldsymbol{B}$ updating module is proposed, which is combined with the remaining floating-point (FLP) modules to form a mixed precision BFGS-QN implementation on FPGA.
- A matrix property checking and search direction switch scheme is proposed to address the overflow issue, and a precision scaling scheme is proposed to deal with the low precision issue, caused by the reduced wordlength. The two schemes make the FXP design meet the convergence requirement of the training process, giving a tradeoff between resource and precision.
- The experimental results show that compared with the single-precision FLP format, the FXP format $\boldsymbol{B}$ matrix updating design reduces the usage of LUTs, FFs, DSPs and BRAMs of the BFGS-QN implementation by 10.9%, 20.2%, 2.2% and 18.1%, respectively.

The paper is organized as follows. In Section II, the BFGS-QN method is briefly introduced, and the motivation and challenges of designing the FXP $\boldsymbol{B}$ updating module are discussed. Section III proposes the FXP design and the solutions to address the challenges. Section IV presents the designed hardware implementation in detail. Section V evaluates the proposed hardware implementation. Section VI concludes the paper and discusses future works.

II. Problem Statement

The whole process of the BFGS-QN algorithm is summarized in Algorithm 1. It iteratively updates $\boldsymbol{w}$ ($\boldsymbol{w} \in R^n$), which are the NN internal weights, based on training error $E_T(\boldsymbol{w})$ and derivative $\partial E_T/\partial \boldsymbol{w}$ from an initial value $\boldsymbol{w_0}$. Step 0 is an initialization step, where the initial value $\boldsymbol{w_0}$ is randomly chosen and the iteration criterion is set. Step 1 computes the search direction $\boldsymbol{d_k}$. In Step 2, a line search is used to obtain step size λ_k along the search direction. Then, Step 3 finds a new $\boldsymbol{w_{k+1}}$ with the search direction and step size, and computes the gradient. In Step 4, if the gradient is small enough, then the iteration process terminates and returns the solution $\boldsymbol{w^*}$; otherwise the process iterates again by updating the inverse of the approximate Hessian matrix, $\boldsymbol{B}$ in Step 5.

Algorithm 1 shows that $\boldsymbol{B}$ updating involves a number of vector by vector, matrix by vector and matrix by matrix computations, which take significant amount of computation time and resources. In addition, matrix $\boldsymbol{B}$ needs to be stored intermediately for updating itself as well as computing search direction in Step 1. Given $n \times n$ matrix $\boldsymbol{B}$, the storage space required is n^2 which increases quickly as n becomes large.

It is well known that the FXP design can reduce the utilization of resource and memory by the FXP design. Therefore, this paper considers a mixed precision implementation of the BFGS-QN algorithm on FPGA, where the $\boldsymbol{B}$ updating module is implemented using FXP representation and the other modules are implemented using FLP representation. The challenges in implementing $\boldsymbol{B}$ updating using FXP arithmetic are mainly from two aspects:

Algorithm 1 BFGS-QN algorithm.

```
Step 0: Initialization.
        Choose the initial point $\boldsymbol{w_0} \in R^n$, $\boldsymbol{g_0} = \nabla E_T(\boldsymbol{w_0})$.
        Set $\boldsymbol{B_0} = \boldsymbol{I}$, $\varepsilon = 1 \times 10^{-5}$ and $k = 0$.
Step 1: Compute search direction $\boldsymbol{d_k} = -\boldsymbol{B_k g_k}$.
Step 2: Compute step size $\lambda_k$ by solving
        $E_T(\boldsymbol{w_k} + \lambda_k \boldsymbol{d_k}) = \min_{\lambda \geq 0} E_T(\boldsymbol{w_k} + \lambda \boldsymbol{d_k})$.
Step 3: Update $\boldsymbol{w_{k+1}} = \boldsymbol{w_k} + \lambda_k \boldsymbol{d_k}$,
        $\boldsymbol{g_{k+1}} = \nabla E_T(\boldsymbol{w_{k+1}})$.
Step 4: Termination test.
        if $||\boldsymbol{g_{k+1}}|| \leq \varepsilon$ Return $\boldsymbol{w}$.
        else Go to Step 5.
Step 5: Set $\boldsymbol{s_k} = \boldsymbol{w_{k+1}} - \boldsymbol{w_k}$, $\boldsymbol{y_k} = \boldsymbol{g_{k+1}} - \boldsymbol{g_k}$,
        $\boldsymbol{B_{k+1}} = \boldsymbol{B_k} + \frac{1}{\boldsymbol{y_k^T s_k}}[(1 + \frac{\boldsymbol{y_k^T B_k y_k}}{\boldsymbol{y_k^T s_k}})\boldsymbol{s_k s_k^T}$
        $-(\boldsymbol{B_k y_k s_k^T} + \boldsymbol{s_k y_k^T B_k})]$
Step 6: Set $k = k + 1$, go to Step 1.
```

1) FXP implementation of $\boldsymbol{B}$ updating may result in overflows in the datapath. The overflows again may lead $\boldsymbol{B}$ to be not positive definite. Matrix $\boldsymbol{B}$ in the BFGS-QN algorithm has a very important property that it is a positive definite matrix, which ensures each iteration of the BFGS-QN algorithm descend towards to the optimal solution [11]. Therefore, it is very important to guarantee the positive definiteness of $\boldsymbol{B}$ in each iteration. In other words, losing the positive definiteness could negatively affect the convergence of the algorithm or even lead to failure.

2) Reduced wordlength reduces the accuracy of the arithmetic computations. Low precision $\boldsymbol{B}$ will affect computation accuracy in finding search direction $\boldsymbol{d_k}$ and step size λ_k, which consequently degrades the convergence rate.

III. Fixed-Point Design of B Updating

The goal of our fixed-point design is to save resources. However, the division operation $\frac{1}{\boldsymbol{y_k^T s_k}}$ in $\boldsymbol{B}$ updating is an exception. It is shown that fixed-point divider consumes computational resources more than single-precision floating-point divider when the bit-width of input is more than 32 bits [12], [13]. Therefore, the B updating module is divided into two regions. In the FLP region, $\frac{1}{\boldsymbol{y_k^T s_k}}$ is calculated. In the FXP region, the remaining computations are calculated. There are FLP-to-FXP converters at the edge of two regions.

Signed two's complement FXP numbers can be represented as $Q(QI, QF)$, in which QI is the bit-width of the integer part including the sign bit, and QF is the bit-width of the fractional part. The range of decimal numbers represented by the format $Q(QI, QF)$ is $[-2^{QI-1}, 2^{QI-1} - 2^{-QF}]$. Given range $[c_{min}, c_{max}]$ and resolution ε ($\varepsilon \leq 1$), a FXP representation of c is determined by the following formulas. QF is determined by the resolution requirement,

$$QF(\varepsilon) = \lceil -log_2 \varepsilon \rceil \tag{1}$$

and QI is determined by the range as below.

$$QI(c_{min}, c_{max}) = \lceil log_2 max(|c_{min}|, |c_{max}|) + 1 \rceil \quad (2)$$

A. *FXP format determination*

This paper targets at a general hardware implementation for various NN training. The goal is to find good FXP format with right precision and range which reduces hardware resource usage while not affecting NN training convergence. Given wide range of NN topologies and training data, the range of variables in B updating is difficult to determine theoretically. Especially, the values of training data could have very different orders of magnitude. The large variation would increase the difficulty in convergence of the training process, leading to low accuracy of the trained NN. Therefore, before starting training, the training data are first scaled to have similar order of magnitude in the typical NN training process. For each input parameter x in the training data, the scaling is defined as [14], $x' = x'_{min} + \frac{x - x_{min}}{x_{max} - x_{min}}(x'_{max} - x'_{min})$. x_{min} and x_{max} are the minimum and the maximum of x in the training data, respectively, and $[x'_{min}, x'_{max}]$ is the input parameter range after scaling.

We first use a simulation-based analysis method [15] to determine the FXP format $Q(QI, QF)$ for each variable. A single-precision FLP version of BFGS-QN algorithm is implemented on MATLAB. Various NN topologies ($n = 32, 64, 128, 256, 512$) are trained using the implementation. The integer bit-width is designed according to the numerical range of the intermediate variables involved in B updating that can be observed conveniently during executions of the MATLAB implementation. For example, the maximum observable value of elements of $\boldsymbol{B}$ is 1023, and thus its integer bit-width is set as 11 bits according to (2).

The fraction bit-width is chosen with respect to the tradeoff between resource usage and precision required to achieve the demanding training error. The process is the following. In the first step, the fraction bit-width is selected on the base of computational resource. Table I lists the resource usage of FXP multiplier and adder with different input bit-widths from Xilinx IP cores. For each variable with the determined QI, the wordlength is set as the closest input bit-width of the corresponding operator. For example, if $\boldsymbol{B}$ is an operand to a multiplier, its wordlength can be set as 18 or 25 bits, *i.e.,* the fraction bit-width is 7 or 14 bits. If the wordlength of another operand is 25 or 18 bits, one DSP is used. The initial fraction bit-width of all intermediate variables are selected following the same way. In the second step, the accuracy of the FXP $\boldsymbol{B}$ updating is validated. The BFGS-QN MATLAB implementation is executed with the FXP $\boldsymbol{B}$ updating module. If the training error meets the requirement, then the FXP design is done; otherwise moving to the next step. In the third step, the variables which significantly affect the accuracy are searched, such as those which suffer from underflows. For these variables, their wordlength is increased to the next level in Table I by increasing the fraction bit-width. Continuing with the previous example, if $\boldsymbol{B}$ is the one whose precision

TABLE I
RESOURCE USAGE OF FXP MULTIPLIER AND ADDER.

Mult (bit-width)	# DSPs	Add (bit-width)	# LUTs	# FFs
18×25	1	25+25	50	75
25×35	2	35+35	70	105
40×35	4	40+40	100	160

negatively affect the training error, the wordlength of $\boldsymbol{B}$ takes 25 or 35 bits and two DSPs are used. Afterwards, the process goes to the second step again.

B. *Matrix property checking and search direction switch*

In the previous subsection, the integer bit-width of intermediate variables in $\boldsymbol{B}$ updating is determined on the base of FLP simulation under a set of training data and NN structures. The real numerical range of the intermediate variables under other inputs could be different and thus overflows occur in practice. The observation is that when the overflows occur, the resultant $\boldsymbol{B}$ may not be positive definite. As a result, $\boldsymbol{d_k}$ computed in Step 1 carries wrong search direction, making the convergence process diverge from the expected descent route.

One possible solution to this issue is to check the positive definiteness of matrix $\boldsymbol{B}$. A symmetric $n \times n$ real matrix $\boldsymbol{M}$ is said to be positive definite if $\boldsymbol{z^T M z}$ is strictly positive for every non-zero column vector $\boldsymbol{z}$ of n real numbers. In the $\boldsymbol{B}$ updating formula in Step 5 of Algorithm 1, there is a term $\boldsymbol{y_k^T B_k y_k}$, which can be used to check the positive definite property of matrix $\boldsymbol{B}$ during the updating process without introducing extra computational resource cost. If the property holds, the updating formula continues computation; otherwise the identity matrix $\boldsymbol{I}$ is assigned to $\boldsymbol{B_{k+1}}$. Identity matrix assignment makes the training process continues from the current solution with the negative gradient direction $-\boldsymbol{g_k}$, which belongs to the classical gradient-based methods.

C. *Precision scaling*

As mentioned in Section III-A, the fraction bit-width is first chosen to save computational resources and then increases if the training precision is not satisfied. Another way of meeting the precision requirement is to apply precision scaling. We particularly pay attention to variables $\boldsymbol{s_k}$ and $\boldsymbol{y_k}$. In the $\boldsymbol{B}$ updating formula, $\boldsymbol{s_k}$ is the input of three multipliers and $\boldsymbol{y_k}$ is the input of two multipliers. The wordlength of $\boldsymbol{s_k}$ and $\boldsymbol{y_k}$ directly affect the computation precision and resource consumption of the matrix updating module. Therefore, precision scaling is applied to $\boldsymbol{s_k}$ and $\boldsymbol{y_k}$.

In the $\boldsymbol{B}$ updating formula, if $\boldsymbol{s_k}$ and $\boldsymbol{y_k}$ are multiplied by the same value, the intermediate computation result shown below remains the same.

$$\frac{1}{\boldsymbol{y_k^T s_k}}[(1 + \frac{\boldsymbol{y_k^T B_k y_k}}{\boldsymbol{y_k^T s_k}})\boldsymbol{s_k s_k^T} - (\boldsymbol{B_k y_k s_k^T} + \boldsymbol{s_k y_k^T B_k})] \quad (3)$$

For FXP arithmetic, multiplication by 2^b without overflow is equivalent to shift b bits to the left. If the most significant $b+1$ bits in the FXP format of numerator and denominator is all 0 or 1, the extension of the signed bit, left shifting b bits of both

TABLE II
FXP FORMAT OF INTERMEDIATE VARIABLES IN $\boldsymbol{B}$ UPDATING.

Variable	Format	Variable	Format
$\boldsymbol{s_{k_{fi}}}$	(2,16)	$\boldsymbol{y_{k_{fi}}}$	(1,17)
$\boldsymbol{B_k}$	(11,14)	$\boldsymbol{a = \frac{1}{y^T_{k_{fi}} s_{k_{fi}}}}$	(25,15)
$\boldsymbol{b = y^T_{k_{fi}} B_k}$	(8,17)	$\boldsymbol{c = b \bullet y_{k_{fi}}}$	(7,28)
$\boldsymbol{d = a * c}$	(7,18)	$\boldsymbol{e = 1 + d}$	(7,18)
$\boldsymbol{f = s_{k_{fi}} s^T_{k_{fi}}}$	(4,31)	$\boldsymbol{g = e * f}$	(6,34)
$\boldsymbol{h = b * s^T_{k_{fi}}}$	(6,34)	$\boldsymbol{i = s_{k_{fi}} * b^T}$	(6,34)
$\boldsymbol{j = h + i; k = g - j}$	(6,34)	$\boldsymbol{l = k * a}$	(11,14)

numerator and denominator will not affect the result, *i.e.,* the most significant b bits can be omitted. As a result, the format $Q(QI, QF)$ could approximately represent $QI + QF + b$ bits precision. The process of the presented precision scaling is shown in the following example. Let compute $a = \frac{0.00987}{-0.00123}$. Using $Q(1, 23)$ format for the numerator and denominator, respectively, $a = \frac{000000010100000001111001}{111111111010111110110010}$. Omitting the most significant 6 bits of both numerator and denominator resulting in $a = \frac{010100000001111001}{111101011110110010}$, in which the FXP value of a does not change. Therefore, we could use format $Q(1, 17)$ with scaling instead of $Q(1, 23)$ for the computation.

We apply the precision scaling to Step 5 of Algorithm 1. Given the termination condition $||\boldsymbol{g_{k+1}}|| \leq 10^{-5}$, the required bit-width of $\boldsymbol{s_k}$ and $\boldsymbol{y_k}$ is 24 bits. It is desirable to set the bit-width to 18 bits to save computational resources as shown in Table I. To realize the goal, before updating $\boldsymbol{B}$, the values of $\boldsymbol{s_k}$ and $\boldsymbol{y_k}$ are checked. If the most significant b $(1 \leq b \leq 6)$ bits of both $\boldsymbol{s_k}$ and $\boldsymbol{y_k}$ are all 0 or 1, the most significant b bits of $\boldsymbol{s_k}$ and $\boldsymbol{y_k}$ are omitted in the following computation, or $\boldsymbol{s_k}$ and $\boldsymbol{y_k}$ are input directly into the updating formula. In this way, 18 bits can represent the precision close to 24 bits.

The finalized bit-width of the intermediate variables in the FXP region of the $\boldsymbol{B}$ updating module is listed in Table II. To verify the design of the FXP matrix updating, we compare the training error achieved by the mixed precision BFGS-QN with that achieved by the FLP BFGS-QN. Both versions of the BFGS-QN algorithm are given the same initial solution and training data, and run the same number of iterations. The comparison result is presented in Table III. The result shows that the mixed precision BFGS-QN with the FXP matrix updating achieves the training error similar to the FLP version. The overflows happen about 1 time per 100 iterations on average, which has little effect on the final training error by using the matrix property checking and search direction switch method. These demonstrate that the FXP design of matrix updating has good computation accuracy and satisfies the convergence quality requirement of the training process.

IV. FPGA Hardware Implementation

The hardware design of the $\boldsymbol{B}$ updating module is outlined in Fig. 1. As mentioned earlier, the module includes the FLP region and the FXP region. To keep $\boldsymbol{B}$ in the FXP format, computation of search direction $\boldsymbol{d_k}$ is also merged into the FXP region of the $\boldsymbol{B}$ updating module.

TABLE III
COMPARISON RESULT BETWEEN THE MIXED PRECISION BFGS-QN AND THE FLP BFGS-QN.

NN topology (3-layer MLP)	n	Training error (%)	
		FLP	Mixed precision
3-8-1	32	0.2	0.35
7-8-1	64	1.97	2.00
6-8-10	128	1.90	2.02
4-16-12	256	0.78	0.82
15-32-1	512	0.89	0.95

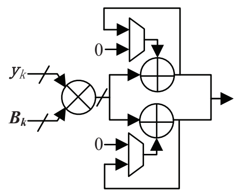

Fig. 2. Hardware design of MVM operation.

In the FLP region, $\boldsymbol{s_k}$ is obtained directly from Step 3, $\boldsymbol{y_k}$ is computed by a subtracter. A deeply pipelined dot-product(DP) unit [16] is implemented for the computation of $\boldsymbol{y_k^T s_k}$ with $T_{DP} = n + L_{mul} + L_{add}\lceil \log_2 L_{add} + 2 \rceil$, where L_{mul} and L_{add} are the latency of multiplier and adder, respectively. Then the result of $\frac{1}{\boldsymbol{y_k^T s_k}}$ is obtained. The intermediate computation results are passed to the FXP region through FLP-to-FXP converters. During the conversion, precision scaling is applied to FXP $\boldsymbol{s_{k_{fi}}}$ and $\boldsymbol{y_{k_{fi}}}$. In hardware, the precision scaling just introduces extra comparators and multiplexers.

In the FXP region, a fully pipelined matrix by vector multiplication (MVM) is designed, where an element of the matrix enters into the multiplier per cycle. The hardware structure is presented in Fig. 2. The output of the multiplier is passed to two adders under a ping-pong control, so that the multiplication result is accumulated in turn. This structure can avoid the pipeline pause which occurs in implementing MVM by separated vector by vector multiplications. As a result, the latency of the MVM operation is $T_{MVM} = L_{mul} + L_{add} + n^2$. What's more, the column vector by row vector multiplication (CRM) operation is implemented using a pipelined multiplier.

After calculation of $\boldsymbol{y^T_{k_{fi}} B_k y_{k_{fi}}}$, the positive definiteness of $\boldsymbol{B}$ is evaluated. Then, $\boldsymbol{s_{k_{fi}} s^T_{k_{fi}}}$, $\boldsymbol{s_{k_{fi}} y^T_{k_{fi}} B_k}$, $\boldsymbol{y^T_{k_{fi}} B_k s^T_{k_{fi}}}$ and $\boldsymbol{1 + \frac{y^T_{k_{fi}} B_k y_{k_{fi}}}{y^T_{k_{fi}} s_{k_{fi}}}}$ are computed in parallel by matching the latency of multipliers and adders and under control of state machine. The whole design is pipelined and $\boldsymbol{B}$ is updated one element per cycle.

In the hardware design, five on-chip RAMs are exploited to temporally store the intermediate computation results $\boldsymbol{y_k}$, $\boldsymbol{s_{k_{fi}}}$, $\boldsymbol{y_{k_{fi}}}$, $\boldsymbol{y_{k_{fi}} B_k}$ and $\boldsymbol{B_{k+1}}$, which are used in multiple subsequent calculations. The on-chip storage for $\boldsymbol{B_{k+1}}$ needs $\boldsymbol{n^2}$ elements while the other four are $\boldsymbol{n}$ elements.

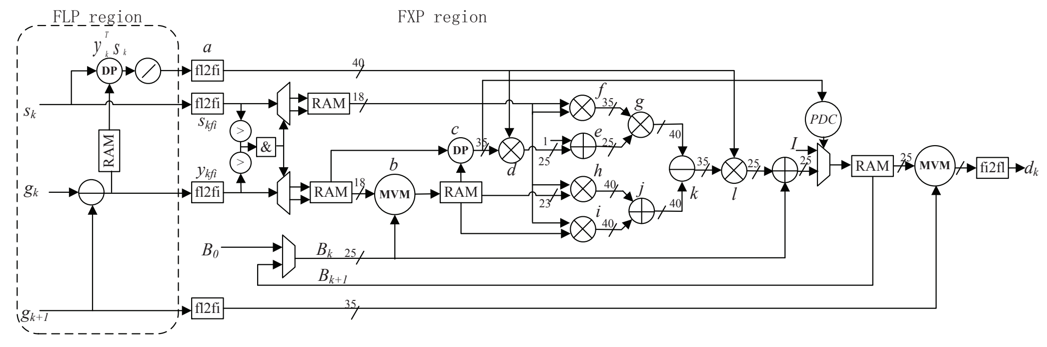

Fig. 1. Hardware design of $\boldsymbol{B}$ updating module. MVM: matrix by vector multiplication. DP: dot product. PDC: positive definiteness checking.

TABLE IV
RESOURCE COMPARISON OF THE $\boldsymbol{B}$ UPDATING MODULE WITH FLP AND FXP FORMATS. WHEN COUNTING BRAM, 36K RAM IS 1 BLOCK AND 18K RAM IS 0.5 BLOCK.

	LUT		FF		DSP		RAM	
n	FLP	FXP	FLP	FXP	FLP	FXP	FLP	FXP
32	7880	3533	12706	5034	24	20	5	4
64	7891	3610	12717	5074	24	20	8	6
128	7969	3617	12736	5056	24	20	18	14.5
256	7787	3859	12252	5110	24	20	62	49
512	8986	4404	12776	5116	24	20	236	187

V. Experimental Results

The mixed precision BFGS-QN implementation is applied to train five 3-layer perceptron NNs, each learning the input-output relationship of a different polynomial function in the form of $y_k = x_1^{a_{1_k}} + x_2^{a_{2_k}} + \ldots + x_{N_1}^{a_{N_{1_k}}}$. The five NNs have different sizes as shown in Table VII. The proposed design is synthesized and implemented on the Net-FPGA SUME (xc7vx690tffg1761-3) board, running at 250 MHz. The evaluation is carried out from two aspects: FXP $\boldsymbol{B}$ updating vs FLP $\boldsymbol{B}$ updating and mixed precision BFGS-QN VS three other QN implementations including a C++ version of BFGS compiled on an Intel Core i5-4590 CPU at 3.3GHz with 8GB RAM, FPGA-DFP [10] and FLP FPGA-BFGS implmented on the same FPGA board. The comparison metrics include resource usage, execution time and power consumption.

A. FXP $\boldsymbol{B}$ updating vs FLP $\boldsymbol{B}$ updating

The resource usage of the $\boldsymbol{B}$ updating module with FLP and FXP formats is reported in Table IV. Compared with the FLP format, the FXP design reduces LUT, FF, DSP, and BRAM usage by up to 55.2%, 60.4%, 16.7% and 20.9%, respectively. As n increases, the computational resources usage is almost the same, while the BRAM usage increases quickly and the reduction becomes significant.

The effect of the precision scaling on resource reduction is also evaluated. The resource usage of the FXP $\boldsymbol{B}$ updating module without and with precision scaling is shown in Table V. Note that the scaling is applied to s_k and y_k and 18 bits are used instead of 24 bits. As mentioned earlier, precision scaling introduces comparators and MUXs, which correspond to the extra 6.6% LUT and 1.6% FF in Table V. The benefits are 4 DSPs and 1 BRAM reduction. Given the scarcity of DSP and BRAM, the effect is positive and promising.

TABLE V
EFFECT OF PRECISION SCALING. NS: NO SCALING. S: SCALING.

	LUT		FF		DSP		RAM	
n	NS	S	NS	S	NS	S	NS	S
32	3354	3533	5008	5034	24	20	5	4
64	3376	3610	5020	5074	24	20	7	6
128	3381	3617	5012	5056	24	20	15.5	14.5
256	3571	3859	5029	5110	24	20	50	49
512	4163	4404	5055	5116	24	20	188	187

TABLE VI
EXECUTION TIME COMPARISON OF THE $\boldsymbol{B}$ UPDATING MODULE WITH FLP AND FXP FORMATS.

n	FLP (ms)	FXP (ms)	Reduction
32	0.027	0.009	66.7%
64	0.070	0.036	48.9%
128	0.205	0.137	33.5%
256	0.672	0.535	20.4%
512	2.392	2.118	11.5%

The FXP design also improves the speed of the $\boldsymbol{B}$ updating process. Table VI shows the execution time of the module. The execution time in the FXP design is reduced by up to 66.7% due to the less pipeline stages of fixed point IP and deeply pipelined design. As n increases, the gap between the FLP and FXP designs decreases. This is because the computation complexity is $O(n^2)$ and the execution time of both designs is dominated by n^2 for large n.

B. BFGS-QN(mixed) VS three other QN implementations

Table VII reports resource usage of the whole BFGS-QN implementations. When taken into the whole implementation, the FXP $\boldsymbol{B}$ updating module results in resource reduction in LUT, FF, DSP and BRAM by up to 10.9%, 20.2%, 2.2% and 18.1%, respectively. In addition, computational resource requirements of the implementations for different NNs are almost the same, showing good scalability. In contrast, the BRAM usage increases as the network size grows, due to the on-chip storage of matrix $\boldsymbol{B}$.

We also evaluate the power consumption of the FXP design. Table VIII shows that the dynamic power of the BFGS-QN implementation is reduced by up to 10.1%.

TABLE VII
RESOURCE COMPARISON OF THE FLP BFGS-QN AND THE MIXED PRECISION BFGS-QN.

NN	LUT		FF		DSP		RAM	
(n)	FLP	Mixed (Reduction)	FLP	Mixed (Reduction)	FLP	Mixed (Reduction)	FLP	Mixed (Reduction)
3-8-1 (32)	39668	35515(10.5%)	56856	49625(12.7%)	182	178 (2.2%)	27	26(3.7%)
7-8-1 (64)	39939	35587(10.9%)	57926	50660(12.5%)	182	178 (2.2%)	32	30(6.3%)
6-8-10 (128)	39380	35748(9.2%)	58768	46925(20.2%)	182	178 (2.2%)	43	40(7.0%)
4-16-12 (256)	39532	35658(9.8%)	60514	53664(11.3%)	182	178 (2.2%)	89	76(14.6%)
15-32-1 (512)	39914	36147(9.4%)	78662	71809(8.7%)	182	178 (2.2%)	272	223(18.1%)

TABLE VIII
DYNAMIC POWER CONSUMPTION COMPARISON.

NN	FLP BFGS-QN	Mixed precision BFGS-QN (Reduction)
3-8-1	2.297 W	2.115 W (8.0%)
7-8-1	2.137 W	1.921 W (10.1%)
6-8-10	2.290 W	2.087 W (9.7%)
4-16-12	2.361 W	2.165 W (8.3%)
15-32-1	2.473 W	2.360 W (4.6%)

TABLE IX
COMPARISON OF EXECUTION TIME PER ITERATION AMONG CPU, FPGA-DFP, FPGA-BFGS(FLP), FPGA-BFGS(MIXED).

NN	CPU (ms)	FPGA-DFP (ms)	FPGA-BFGS(FLP) (ms)	FPGA-BFGS(Mixed) (ms)
3-8-1	32.93	2.54	1.18	1.16
7-8-1	63.58	5.01	1.52	1.48
6-8-10	139.99	9.63	2.03	1.96
4-16-12	361.70	18.97	3.76	3.62
15-32-1	844.05	36.82	8.05	7.78

Table IX shows that the mixed precision FPGA-BFGS implementation is up to 108 times faster than the CPU-BFGS and 5 times faster than the FPGA-DFP. Although the execution time of FXP B updating module is reduced, the speed of whole BFGS-QN keeps almost same with the FLP version. There are two reasons. Firstly, the clock frequency is not improved because the critical path is between DSP and Block RAM which is not in the B updating module. Secondly, it is the line search module which repeatedly evaluates the objective function that dominates the execution time.

VI. CONCLUSION

This paper presents a mixed precision BFGS-QN implementation on FPGA. The resource-intensive module of $\boldsymbol{B}$ updating is implemented using the FXP format. A matrix property checking and search direction switch method is proposed to ensure the overflows do not affect the convergence of the implementation in training NNs. In addition, a precision scaling scheme is developed to use short wordlength to represent high precision. The experimental results show that the mixed precision BFGS-QN design brings up to 10.9% LUT, 20.2% FF, 2.2% DSP and 18.1% BRAM reduction, respectively.

In future, we would like to find ways to further reduce the memory usage. Lower-precision floating point arithmetic, such as 18-bit floating point, will be considered. In addition, the BFGS-QN implementation targeting neural network training will be further investigated to find space for approximate computation which generally exists in neural network.

ACKNOWLEDGMENT

The authors would like to thank the support of the National Natural Science Foundation of China under Grant 61574099 and the Tianjin Municipal Transportation Science and Technology Development Plan Project under Grant 2017B-40.

REFERENCES

[1] S. Razavi and B. Tolson, "A new formulation for feedforward neural networks," *Neural Networks, IEEE Transactions on*, vol. 22, no. 10, pp. 1588–1598, Oct 2011.

[2] I. T. Christou, *A Review of Optimization Methods*, 2012.

[3] A. W. Savich, M. Moussa, and S. Areibi, "The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study," *IEEE Transactions on Neural Networks*, vol. 18, no. 1, pp. 240–252, Jan 2007.

[4] X. Wang and J. Zambreno, "An FPGA implementation of the Hestenes-Jacobi algorithm for singular value decomposition," in *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, May 2014, pp. 220–227.

[5] H. Li, J. Davis, J. Wickerson, and G. A. Constantinides, "Architect: Arbitrary-precision constant-hardware iterative compute," in *IEEE International Conference on Field-Programmable Technology*, 2017.

[6] X. Dong, H. Li, and Y. Wang, "High-speed FPGA implementation of an improved LMS algorithm," in *Computational Problem-solving (ICCP), 2013 International Conference on*, Oct 2013, pp. 342–345.

[7] A. Roldao and G. A. Constantinides, "A high throughput FPGA-based floating point conjugate gradient implementation for dense matrices," *Acm Transactions on Reconfigurable Technology & Systems*, vol. 3, no. 1, pp. 156–165, 2010.

[8] A. Gomperts, A. Ukil, and F. Zurfluh, "Development and implementation of parameterized FPGA-based general purpose neural networks for online applications," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 1, pp. 78–89, Feb 2011.

[9] N. Izeboudjen, A. Farah, H. Bessalah, A. Bouridene, and N. Chikhi, "Towards a platform for FPGA implementation of the MLP based back propagation algorithm," in *Computational and Ambient Intelligence, International Work-Conference on Artificial Neural Networks*, 2007, pp. 497–505.

[10] Q. Liu, R. Sang, and Q. Zhang, "FPGA-based acceleration of Davidon-Fletcher-Powell quasi-Newton optimization," *Transactions on Tianjin University*, vol. 22, no. 5, pp. 381–387, Oct. 2016.

[11] A. Antoniou and W. S. Lu, *Practical Optimization: Algorithms and Engineering Applications*. Springer Publishing Company, Incorporated, 2010.

[12] "Performance and resource utilization for divider generator v5.1," https://www.xilinx.com/support/documentation/ip_documentation/ru/div-gen.html#virtex7, Jan 2018.

[13] "Performance and resource utilization for floating-point v7.1," https://www.xilinx.com/support/documentation/ip_documentation/ru/floating-point.html, Jan 2018.

[14] Q.-J. Zhang, K. Gupta, and V. Devabhaktuni, "Artificial neural networks for RF and microwave design - from theory to practice," *Microwave Theory and Techniques, IEEE Transactions on*, vol. 51, no. 4, pp. 1339–1350, Apr. 2003.

[15] F. Xu, H. Chen, X. Gong, and Q. Mei, "Fast nonlinear model predictive control on FPGA using particle swarm optimization," *IEEE Transactions on Industrial Electronics*, vol. 63, no. 1, pp. 310–321, Jan 2016.

[16] K. C. S. Li, "FPGA-based pipelining floating multiply accumulator," *Electronic Technology & Software engineering*, vol. 2, pp. 140–142, 2016.