VOP: Architecture of a Processor for Vector Operations in On-line Learning of Neural Networks

Gopinath Mahale, Soumitra K. Nandy
CAD Lab, Indian Institute of Science, India
email: {mahaleg,nandy}@cadl.iisc.ernet.in

Eshan Bhatia
BITS Pilani, Pilani, India
email: f2011323@pilani.bits-pilani.ac.in

Ranjani Narayan
Morphing Machines Pvt Ltd., India
email: ranjani@morphing.in

Abstract—In this paper we propose architecture of a processor for vector operations involved in on-line learning of neural networks. We target to implement on-line learning on a Radial Basis Function Neural Network (RBFNN) based Face Recognition (FR) system that has pseudo inverse computation as an essential component during training. Synaptic weights of RBFNN output layer need to be updated whenever the FR system comes across a new face to be learnt. For real-time on-line learning, update of synaptic weights is done using an existing Incremental Pseudo Inverse (IPI) algorithm in the place of compute intensive pseudo inverse algorithm. We design a custom data-path for vector operations appearing in IPI algorithm. The custom data-path along with configuration and memory access mechanisms forms a processing unit, termed Processor for Vector Operations (VOP). We simulate and synthesize VOP to target Virtex-6 FPGA using the Xilinx ISE. Apart from on-line learning, the VOPs can be used in acceleration of several applications involving predominant vector-matrix operations.

I. INTRODUCTION

Biometric systems have become ubiquitous lately with increasing need for access control and security. Face Recognition (FR) [1] is a biometric method that performs recognition using facial image of subjects. FR systems, being a necessary part of safe and inclusive world, are used in applications such as authentication for access, human computer interaction, and recognition of crime suspects in public places etc. In existing FR systems, compute intensive training is performed off-line and trained coefficients are loaded on the FR system for real-time recognitions. However, when the application involves learning unseen faces and tracking them, the FR module needs to learn new faces on-line. Hence for effective functioning of FR system, update of training coefficients in real-time requires the respective algorithm to be accelerated.

We target to implement on-line learning on a Radial Basis Function Neural Network (RBFNN)[2] based FR system [3] which has shown good recognition accuracy on image databases of faces with different pose and illumination conditions. Synaptic weights of RBFNN are updated every time samples of a new or known subject need to be learnt. The new set of synaptic weights are computed by iterative methods like gradient descent or by faster methods such as computation of least square solution, i.e., by computing pseudo inverse. Although gradient descent method is ideal for hardware realizations, the iterative behaviour makes it unsuitable for real-time applications. On the other hand, pseudo inverse involves computation of inverse of a matrix which is of complexity $O(n^3)$. We use an existing incremental method for pseudo inverse computation which has a complexity of $O(n^2)$, suitable for real-time implementations, in the place of matrix inverse computation. We term this algorithm Incremental Pseudo Inverse (IPI) algorithm. We observe that computations in IPI algorithm consist of basic vector operations which include dot products, cross products, addition, subtraction, vector-matrix multiplications, matrix-vector multiplications and vector division by scalar. The parallelism exposed by these vector operations encourage us to target a multi-core environment, where these operations are distributed in parts as micro vector operations among available cores. In addition, these domain specific computations are accelerated by a dedicated processor capable of performing micro vector operations in each core.

In this paper we propose architecture of VOP, a processor for vector operations in on-line learning of neural networks. VOP takes addresses of operands along with information about vector micro operation to be performed and address, where results need to be stored, as inputs. During on-line learning, the vector operations in IPI algorithm are divided into micro vector operations that are distributed among several VOPs in parallel. We target maximal throughput essential for real-time on-line learning of neural networks by IPI algorithm. The required throughput is achieved by exploiting spatial and temporal localities while designing sequence of computations in vector operations. The VOPs can also be used in training other forms of neural networks where, similar to RBFNN, synaptic weights are computed using IPI algorithm incrementally. Although we project VOP as a hardware solution for faster learning of synaptic weights in neural networks, it can also be used in accelerating several applications involving predominant vector operations. We show that VOPs can be effectively used as domain specific Custom Function Units (CFUs) to accelerate vector operations in REDEFINE [4], a multi-core coarse grained reconfigurable architecture. The rest of the paper is organized as follows. In section II we describe on-line learning and incremental learning of RBFNN using IPI algorithm in detail. In section III we explain architecture of VOP in detail with both fixed point and floating point versions of VOP. We show FPGA synthesis results and comment on performance of VOP in section IV. Finally we conclude in section V.

II. ON-LINE LEARNING OF RBFNN

RBFNN [2] is widely used in pattern recognition applications and has been popular due to its fast learning [5] capabilities and good generalization performance [6] for large datasets. We use Gaussian functions as radial basis functions of RBFNN. During training, samples in sample space are clustered. Every hidden node of RBFNN is associated with a single cluster. Let $N_{\text{clust}}$ be the number of hidden nodes. Output of $j^{th}$ hidden node for input vector $x$ is given by

$$
Y_j = \exp \left( \frac{\|x - \mu_j\|^2}{2\sigma_j^2} \right), \quad j = 1, 2, ..., N_{\text{clust}}
$$

(1)
where, \( \mu_j \) and \( \sigma_j \) are mean and standard deviations of samples in \( j^{th} \) cluster respectively. Nodes in the output layer, termed output nodes, perform weighted addition of all the hidden node outputs and a bias input equal to 1. Number of output nodes is equal to the number of classes, \( N_{class} \). Output of \( k^{th} \) output node is given by

\[
y_k = w_{k,N_{class}+1} + \sum_{j=1}^{N_{class}} Y_j \times w_{k,j}, \quad k = 1, 2...N_{class} \quad (2)
\]

Here the weights are equal to the synaptic weights computed during training. The input sample is classified corresponding to the index of output node with highest output value. To make the RBFNN based FR system behave in this manner, it needs to be trained beforehand which involves two steps: cluster formation in sample space and synaptic weight computation. There are various approaches for clustering the data in sample space. Simplest among them is to perform supervised clustering of data, where each cluster comprises samples from corresponding classes in it. This approach is extended to methods [7][8] that assign multiple clusters per class to avoid overlap of clusters, a major reason for misclassification. Another widely used method is k-means clustering [9] which is an unsupervised iterative clustering algorithm. There exist incremental algorithms for forming clusters [10].

We focus on finding synaptic weights, which is the compute intensive part of RBFNN training. We consider the method of finding pseudo inverse of a matrix, that is suitable for real-time implementations. Let \( H \) be a matrix made of output of hidden nodes for every input sample along with a column of ones as bias input. Each column in matrix \( H \) holds output of corresponding hidden node for different input samples as shown in Fig. 1(a) and Fig. 1(b). Let \( w \) be a matrix made of synaptic weights to be computed and \( y \) be the corresponding set of outputs of output layer. Using equation 2 these matrices are related as \( H \times w = y \). Least square solution to this overdetermined system of equations is given as \( w = H^* \times y \), where, \( H^* \) is the pseudo inverse of matrix \( H \), which is computed as

\[
H^* = (H^T H)^{-1} H^T \quad (3)
\]

Finding pseudo inverse involves matrix inverse operation of complexity \( O(n^3) \). For on-line learning it is possible to avoid inverse operations by computing new set of synaptic weights using existing ones whenever a new sample or class needs to be learnt. There are various works reported in literature on IPI [11][12]. A variance matrix update based incremental method [13][14] works well for our requirement which is based on update of variance matrix, \( A^{-1} \), given by \( A^{-1} = (H_p^T H)^{-1} \). This method also uses projection matrix \( P \), given by

\[
P = I_p - HA^{-1} H^T \quad (4)
\]

where, \( I_p \) is a unit matrix. In this method, \( A^{-1} \) and \( P \) are updated with addition of a new sample to existing clusters or addition of a new cluster. For adding samples and clusters, updates are performed using different sets of computations. Synaptic weights are computed from updated variance matrix by[13]

\[
w = A^{-1}(H^T y) \quad (5)
\]

We run IPI algorithm to train the RBFNN on a FR system that uses Principle Component Analysis (PCA) [15] for feature extraction and RBFNN for classification of these features. A standard face database, AT&T face database [16], is used to analyse correctness of IPI algorithm. Alternative samples from each class are used as training and testing samples. Single cluster per class is used for simplicity. The computations and observed results in MATLAB platform for the face database are described in the following subsections. There are counterpart equations [13] for removal of a sample or a cluster from the training database. We do not consider them at this stage, as these operations do not come in the scope of FR system under consideration. We target to include new samples and classes on-line, and removal can be done off-line. However, these operations too can be performed in real-time using VOP described in Section III.

### A. Addition of new samples to existing clusters

To add a new sample to existing clusters, \( H \) matrix is updated with a single row corresponding to the added sample as shown in Fig. 1(a). Let \( h_{p+1} \) be the new added row and \( d \) be the desired output at output layer for the new sample. \( A_{p+1}^{-1} \) and \( F_{p+1} \) are updated as follows[13].

\[
P_{p+1} = \begin{pmatrix} P_p & 0 \\ 0 & 0 \end{pmatrix} + \frac{1}{1 + h_{p+1}^T A_{p+1}^{-1} h_{p+1}} \times \\
\begin{pmatrix} H_p A_{p+1}^{-1} h_{p+1} \\ -1 \end{pmatrix} \begin{pmatrix} H_p A_{p+1}^{-1} h_{p+1} \\ -1 \end{pmatrix}^T \quad (6)
\]

### Table I. Recognition rate for ORL database in the case of samples added to existing clusters

<table>
<thead>
<tr>
<th>Number of initial samples</th>
<th>Number of samples added</th>
<th>Recognition Accuracy with IPI</th>
<th>Recognition Accuracy with pseudo inverse</th>
<th>Root Mean Square error in synaptic weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>160</td>
<td>40</td>
<td>93.5</td>
<td>93.5</td>
<td>( 8.11 \times 10^{-7} )</td>
</tr>
<tr>
<td>120</td>
<td>80</td>
<td>94</td>
<td>94</td>
<td>( 4.34 \times 10^{-7} )</td>
</tr>
<tr>
<td>80</td>
<td>120</td>
<td>77.5</td>
<td>77.5</td>
<td>( 3.86 \times 10^{-7} )</td>
</tr>
</tbody>
</table>

### Table II. Recognition rate for ORL database in the case of samples from new class added to the database

<table>
<thead>
<tr>
<th>Number of initial classes</th>
<th>Number of added classes</th>
<th>Recognition Accuracy with IPI</th>
<th>Recognition Accuracy with pseudo inverse</th>
<th>Root Mean Square error in synaptic weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>39</td>
<td>95.5</td>
<td>95.5</td>
<td>( 1.31 \times 10^{-7} )</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>95.5</td>
<td>95.5</td>
<td>( 1.08 \times 10^{-7} )</td>
</tr>
<tr>
<td>35</td>
<td>5</td>
<td>95.5</td>
<td>95.5</td>
<td>( 0.45 \times 10^{-7} )</td>
</tr>
</tbody>
</table>
\[ A_{p+1}^{-1} = A_p^{-1} - \frac{A_p^{-1} h_{p+1} h_{p+1}^T A_p^{-1}}{1 + h_{p+1}^T A_p^{-1} h_{p+1}} \]  
\[ w_{p+1} = w_p + A_{p+1}^{-1} h_{p+1} (d - h_{p+1} w_p) \]  

Synaptic weights are computed either incrementally with each added sample using equation 8 or at once by equation 5 using updated \( A^{-1} \). Update of \( P \) is not required if we just add samples to the existing clusters. But, if a new cluster is added in future, the computation needs the updated \( P \) as described in section II-B later. Table I shows recognition accuracy with change in initial number of samples. We do not find difference in recognition accuracies for pseudo inverse and IPI algorithm irrespective of initial number of samples. Adding more samples from each class without updating mean and standard deviations of existing clusters lead to misclassification that gets reflected as the dip in recognition accuracy. We address this problem by introducing additional clusters under every class for new samples.

### B. Addition of samples of a new class

Generally in IPI algorithm, the dimensions of input samples do not change which keeps the number of columns of matrix \( H \) fixed. However, in RBFNN the number of columns of matrix \( H \) is equal to \( N_{clust} \) and it changes every time a new class is added. To add samples of a new class, initially they are input to existing hidden nodes to get hidden node outputs for these samples as shown in Fig. 1(b). With addition of every new sample, \( A^{-1} \) and \( P \) matrices are updated using equations 6 and 7 respectively. Adding a new class corresponds to adding one or multiple new clusters in sample space. Mean and standard deviation for the new added cluster is found. Output of corresponding new hidden node for all the input samples are entered in the \( H \) matrix as shown in Fig. 1(b). Let \( h_{m+1} \) be the new column in \( H \) matrix corresponding to the new added cluster. With every new cluster added, \( A_{m+1}^{-1} \) and \( P_{m+1} \) are updated using the following equations[13].

\[ A_{m+1}^{-1} = \begin{pmatrix} A_1^{-1} & 0 \\ 0 & 0 \end{pmatrix} + \frac{1}{\lambda_{m+1} + h_{m+1}^T P_{m} h_{m+1} + 1} \begin{pmatrix} A_1^{-1} h_{m+1} h_{m+1}^T A_1^{-1} & A_1^{-1} h_{m+1} h_{m+1}^T \\ A_1^{-1} h_{m+1} h_{m+1}^T & A_1^{-1} h_{m+1} h_{m+1}^T \end{pmatrix} \begin{pmatrix} A_1^{-1} h_{m+1} h_{m+1}^T A_1^{-1} & A_1^{-1} h_{m+1} h_{m+1}^T \\ A_1^{-1} h_{m+1} h_{m+1}^T & A_1^{-1} h_{m+1} h_{m+1}^T \end{pmatrix}^T \]

\[ P_{m+1} = P_{m} - \frac{P_{m} h_{m+1} h_{m+1}^T P_{m}}{\lambda_{m+1} + h_{m+1}^T P_{m} h_{m+1}} \]

Here \( \lambda \) is the regularization parameter. Table II shows variation in recognition accuracy with change in initial number of classes. In these experiments we ignore the regularization parameter and make \( \lambda \) equal to zero. Here we observe that irrespective of initial number of classes, addition of new classes lead to same recognition accuracy as that of by finding pseudo inverse using equation 3.

Our target FR system requires addition of new classes and samples on-line in real-time. From Table I and II it is clear that, on the considered dataset, adding clusters and samples to the existing database in incremental manner using IPI algorithm does not degrade the recognition accuracy; at the same time offers advantages in terms of computation complexity. However, the rounding error, which is reflected as the root mean square error in synaptic weights increases with problem size. We design light weight VOPs to perform the basic vector operations involved in update of \( A^{-1} \) and \( P \). Multiple VOPs performing such micro vector operations together can perform large vector operations. A detailed description of architecture of VOP for real-time performance is given in section III.

### III. PROCESSOR FOR VECTOR OPERATIONS

Having confirmed efficiency of IPI algorithm, we focus on exploiting inherent parallelism in the vector operations in order to meet real-time requirements. We find limited work [17] in literature for acceleration of incremental learning methods. In this section we first provide the design details of VOP, which is used to perform vector operations. We use multiple VOPs in a multi-core environment in order to exploit parallelism that exists within and across multiple vector operations. We target REDEFINE [4] as our target multi-core architecture.

We design VOPs to perform micro vector operations and use them as domain specific co-processors under each core of the multi-core architecture. Fig. 2(a) shows pictorial representation of VOP. The orchestrator is an entity to configure VOPs for the operation to be performed and it supplies addresses of operands and result. The orchestrator can be either an associated processor core as described before, or a dedicated unit as we see in section III-D. Once the operations are performed, the VOP raises the ready signal indicating that it is ready to receive configuration information for the next vector operation.

#### Algorithm 1: Cache friendly vector-matrix multiplication

**Input:** Matrices \( A_{M \times N} \) and \( B_{M \times N} \)  
**Output:** Matrix \( C_{N \times N} \)  
**for** \( i = 1 \) to \( M \)  
**do**  
\[ // \text{ Accumulate } L \text{ number of intermediate results in the circular FIFO} \]  
\[ \text{for } j = 1 \text{ to } L \text{ do} \]  
\[ \text{Temp}(j) = \text{Temp}(j) + A(i) \times B(i, ((s - 1) \times L) + j) \]  
end  
\[ // \text{ Store the contents of circular FIFO in vector } C \]  
\[ \text{for } j = 1 \text{ to } L \text{ do} \]  
\[ C((s - 1) \times L) + j) = \text{Temp}(j) \]  
end  
end
VOP consists of Configuration Registers (CRs), a Configuration Control (CC), an Address Generation and Control (AGC) unit and a data-path. CC receives the configuration information which includes dimensions and characters of operand and result vectors and writes them to the corresponding registers in CRs. CRs consists of a set of registers that are accessed by AGC for address and control signal generation. AGC generates read and write addresses of operand and result vectors respectively and accordingly generates control signals for the data-path. A Load Store unit (LSU) takes care of read and writes of data to scratch pad and external memories. In our initial realization, we use 32 bit fixed point units for computation. We extend this work to VOPs with 64 bit double precision floating point units suitable for data sets with larger data ranges. In the following sub-sections we describe architectures of data-path and AGC unit. This is followed by description of modifications introduced for floating point operation support in VOPs and its integration on REDEFINE architecture.

A. Data-path

Data-path of VOP is shown in Fig. 2(b). It consists of a multiplier, an adder, a subtracter and a divider that together perform target vector operations. The data-path consists of two input FIFOs to store elements of operand vectors. Few registers are made available in the data-path to store intermediate results. Using a series of multiplexers and demultiplexers the operands are routed to implement different vector operations. The control signals from AGC are input to the data-path through an input control FIFO to achieve synchronization with input data. An accumulator FIFO stores the intermediate results of vector operations leading to better spatial and temporal data locality which is explained in section III-A1. The majority of vector operations performed in the IPI algorithm are vector-matrix operations. In these operations we find least data re-use, and the parallelism exploited under each VOP in multi-processor scenario is limited by the data bandwidth of Network on Chip (NoC). This makes us implement single stream of computations in each VOP which can be increased based on the NoC data bandwidth.

1) Sequence of operations: The sequence of operations are configured in order to achieve good spatial and temporal localities and to achieve maximal throughput. We consider an L1 cache of cache line length L through which the VOP accesses the data from external memory. The external memory is a Double Data Rate Synchronous Dynamic Random Access Memory (DDR SDRAM). Due to column-wise access of matrices stored in row-major order, there will be frequent row-changes in DDR SDRAM which lead to considerable memory latencies. We target high throughput by avoiding these latencies. In addition, we intelligently configure sequence of computations to maintain data in cache for a longer duration to achieve temporal and spatial data localities. In the dot product, cross product, matrix addition, subtraction, division by a scalar and matrix-vector multiplication operations matrices are accessed row-wise. But, in the case of vector-matrix multiplication, the vector is multiplied with each column of operand matrix which requires the data to be accessed column-wise. We device a technique described in Algorithm 1 in which every time a single element of vector is multiplied with L elements of matrix and accumulated on a circular FIFO of depth L. Once all the elements of vector are multiplied with corresponding elements of the matrix, the circular FIFO will have L elements of product vector. This form of access is cache friendly and results in good spatial and temporal locality, although matrix-matrix multiplications are rare in our IPI algorithm, it also follows similar format of computations. As division is an expensive operation, for division by scalar, we compute inverse of the denominator by supplying one of the operands as 1 to the divider. The result is saved in an intermediate register and it is repeatedly used for multiplying with each element of the vector.

B. Address generation and control unit

For each new vector operation, CC initializes the CRs with configuration information received from the orchestrator and sends a start signal to AGC. AGC basically performs the following two operations: Generating addresses to be sent to the LSU, for loading the operands and storing the results and generating control signals for multiplexers and demultiplexers present in the data-path, depending on the operation chosen. A Finite State Machine (FSM) running on AGC generates addresses of input vector elements along with address of resultant vector. In addition, it reads out control signals for different operations which are stored in a small memory named control signal memory. This form implementation leads to simpler FSM and results in reduced resource utilization. Different vector operations are implemented using states that read corresponding set of locations in control signal memory.

Algorithm 2: MAC operation with pipe-lined floating point multiplier and adder

<table>
<thead>
<tr>
<th>Input:</th>
<th>Matrices $A_{1 \times M}$ and $B_{1 \times M}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>Scalar C</td>
</tr>
<tr>
<td>for i = 1 to $\lceil \frac{M}{add} \rceil$ do</td>
<td></td>
</tr>
<tr>
<td>for j = 1 to $P_{add}$ do</td>
<td></td>
</tr>
<tr>
<td>// multiply and accumulate $P_{add}$ accumulated results in pipe-line registers</td>
<td></td>
</tr>
<tr>
<td>$Temp(j) = Temp(j) + A(((i-1) \times P_{add}) + j) \times B(((i-1) \times P_{add}) + j)$</td>
<td></td>
</tr>
<tr>
<td>// Summation of accumulator FIFO contents</td>
<td></td>
</tr>
<tr>
<td>for j = 1 to $P_{add}$ do</td>
<td></td>
</tr>
<tr>
<td>$C = C + Temp(j)$;</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>
We synthesize both fixed point and double precision floating point VOPs on XILINX ISE to target Virtex-6 CX130T FPGA. From the synthesis report the maximum operating frequency of these modules is found to be 180 MHz and 140 MHz respectively. The device utilization on the FPGA is listed in Table III. Operations in the floating point core could not be mapped to DSP slices, hence we see a large percentage of slice LUT usage.

The IPI algorithm is executed on REDEFINE environment described in section III-D. For initial number of classes and samples, \( A^{-1} \) and \( P \) matrices along with synaptic weights are computed off-line using equation 3 and equation 4 respectively and stored in memory. The tasks of adding new samples of existing classes and samples of new classes on-line are performed by dividing the computations in the form of micro vector operations and distributing the computations among the available processor cores. We store elements of matrices in row-major order in memory. Storing the elements of matrix \( H \) with empty locations between the rows in memory makes the job of appending new columns to \( H \) matrix easier. The number of mathematical operations involved in both adding new samples and classes using the IPI algorithm are listed in Table IV. Initially we experiment with a single fixed point VOP with a single stream of computations connected to CE of REDEFINE. We run the IPI algorithm on the CE with and without VOP to analyse the speed up. The maximum operating frequency of CE is 180 MHz on Virtex 6 CX130T FPGA. We use a cache memory of 128 kB with cache line size of 8 words. Fig. 5(a) shows the time required to add a sample of an existing cluster, as described in section II-B, with varying number of classes. A more practical analysis is seen in Fig. 5(b), where FR system learns a new class with five samples as described in section II-B. The plot shows the time required to add five samples of a new class and to compute synaptic weights. Similarly we experiment with another set-up, where data to be processed by VOP is pushed to the scratch pad memories of respective tiles. With unavailability of cache memory for this model, the order of data reads from external memory plays an important role in getting the performance as described in section III-A1. Plots in Fig. 5(c) and Fig. 5(d) show the speed-up achieved by VOP for this push model. In these plots the latencies due to row changes in the DDR SDRAM are not considered, though it is considerably minimized for VOP. From these plots the latencies due to row changes in the DDR SDRAM are not considered, though it is considerably minimized for VOP. From the plots in Fig. 5 we observe that irrespective of the number of classes, i.e., the problem size, we get consistent speed-up using VOPs.

In Fig. 5(b), the time taken to add a new class to database with 400 classes is 1.06 seconds, and it further increases as the number of classes increases. To target a real-time FR system with 450 recognitions per second similar to the one proposed by [3], for worst case scenario, the acceleration needed is approximately 90 times. Therefore we further look into exploiting parallelism in the architecture. We exploit parallelism within VOP by implementing more than one streams of computations as described in section III-A. Here, the performance scales approximately linearly with number of streams, provided availability of network data bandwidth. We exploit parallelism across VOPs which are housed in different nodes of REDEFINE. A large vector operation broken into micro block vector operations distributed among the VOPs is
executed in parallel. Here too the performance scales linearly with number of VOPs, provided equal availability of data bandwidth at all the tiles. Although VOP is designed for acceleration of on-line training algorithm, it can also be used in several applications that have predominant vector-matrix operations. We take an example of eigenspace projection of PCA [15] that extracts 32 discriminant features in a vector of length \( l \). Fig. 6 shows speed-up achieved for different value of \( l \). We find similar speed-up results for other applications.

V. Conclusion

We develop a processor for vector operations involved in on-line learning of Radial Basis Function Neural Network Based Face Recognition (FR) system. We use Incremental Pseudo Inverse (IPI) algorithm in the place of traditional pseudo inverse method to reduce computation complexity during update of synaptic weights. To achieve on-line learning, we accelerate the vector operations in IPI using Processor for Vector Operations (VOP). We experimentally show that, irrespective of problem size, VOP with a single stream can bring consistent speed-up, which can be scaled further using parallelism within VOP and across VOPs. In addition, VOPs can be used in acceleration of large number of applications with predominant vector operations.

Acknowledgment

The authors would like to thank Sachin Bhat, Madhava Krishna C., Saurabh Misra and Niveditha Hariharan for their interest in this work.

REFERENCES


Fig. 5. Time taken for operations on a single CE with and without VOP

Fig. 6. Time taken for eigenspace projection on CE with and without VOP

TABLE III. DEVICE UTILIZATION ON VIRTEx-6 CX130T FPGA

<table>
<thead>
<tr>
<th></th>
<th>Fixed point VOP</th>
<th>Floating Point VOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of slice registers</td>
<td>8790(9%)</td>
<td>6275 (3%)</td>
</tr>
<tr>
<td>Number of slice LUTs</td>
<td>2529 (0%)</td>
<td>2575 (32%)</td>
</tr>
<tr>
<td>Number of Block RAM</td>
<td>1 (0%)</td>
<td>3 (3%)</td>
</tr>
<tr>
<td>Number of DSP48Es</td>
<td>470 (0%)</td>
<td>470 (0%)</td>
</tr>
</tbody>
</table>

TABLE IV. NUMBER OF COMPUTATIONS IN VOP; \( m \) NUMBER OF CLASSES AND \( p \) NUMBER OF TOTAL SAMPLES

<table>
<thead>
<tr>
<th></th>
<th>Addition/Subtraction</th>
<th>Multiplication/Division</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding a sample of ex-</td>
<td>(4(m+1)^2+(m+1)(p+</td>
<td>(4(m+1)^2+(m+1)(p+1))</td>
</tr>
<tr>
<td>isting class</td>
<td>1)+4p^2+1)</td>
<td>(1)(p+2)+p^2+3p+2)</td>
</tr>
<tr>
<td>Adding samples of a</td>
<td>(6(m+1)^2+(2m+2)(p+2))</td>
<td>(6(m+1)^2+(2p+1)(m+1)+4p^2+p+1)</td>
</tr>
<tr>
<td>new class</td>
<td>(+4p^2+5p+3)</td>
<td>(+1))</td>
</tr>
<tr>
<td>Synaptic weight compu-</td>
<td>((m+1)(3m+1)+m)</td>
<td>((m+1)(3m+1))</td>
</tr>
<tr>
<td>tation by equation 8</td>
<td>(m(m+1)(p+m+1))</td>
<td>(m(m+1)(p+m+1))</td>
</tr>
<tr>
<td>Synaptic weight compu-</td>
<td>((m+1)(3m+1)+m)</td>
<td>((m+1)(3m+1))</td>
</tr>
<tr>
<td>tation by equation 5</td>
<td>(m(m+1)(p+m+1))</td>
<td>(m(m+1)(p+m+1))</td>
</tr>
</tbody>
</table>