AN FPGA-BASED ACCELERATOR FOR DISTRIBUTED SVM TRAINING

A Thesis

by

YASHWARDHAN NARAWANE

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Rabi Mahapatra
Co-Chair of Committee,	Vivek Sarin
Committee Members,	Peng Li
Head of Department,	Dilma Da Silva

August 2018

Major Subject: Computer Engineering

Copyright 2018 Yashwardhan Narawane

ABSTRACT

Support Vector Machines are a class of machine learning algorithms with applications ranging from classification to regression and categorization. With the exponential increase in edge computing devices, there is a growing demand to adapt SVM-based techniques for edge analytics. However, training SVM is computationally challenging due to a quadratic complexity in the number of training samples. Consequently, SVM training is performed off-line on back-end servers, which possess the computing power to train SVM models. Creating efficient frameworks for SVM-based edge analytics requires a scalable, distributed training algorithm. Alongside, the computational capabilities of edge nodes must be augmented through energy-efficient hardware accelerators. In this research, we present a scalable FPGA-based accelerator for a distributed SVM training algorithm. The accelerator exploits both data and task parallelism to create efficient, pipelined implementations of computing modules in hardware. We evaluate the training performance of our proposed accelerator for five SVM benchmarks, and compare with a high performance CPU cluster and an embedded SoC server deploying equal number of computing units. The proposed FPGA-based accelerator performs SVM training up to 25x and 1.75x faster than the CPU and SoC counterpart respectively. Alongside, the accelerator provides 9x and 6x reduction in energy consumption, relative to the SoC and CPU clusters respectively.

DEDICATION

To my parents, for having my back throughout this journey

ACKNOWLEDGMENTS

I take this opportunity to thank all those who have guided me towards my goals. First and foremost, I'd like to thank my advisor, Dr. Rabi Mahapatra, for giving me the opportunity to work on an interesting, challenging and relevant research topic. His astute guidance and faith in my abilities kept me motivated to pursue this research. I would also like to thank my committee members, Dr. Vivek Sarin and Dr. Peng Li, for their continual support and keen technical insights.

This research wouldn't have been possible without the friendship, guidance and mentoring of Jyotikrishna Dass, PhD Student, Embedded Systems and CoDesign Lab. I owe the foundations of my research to numerous knowledge transfer and brainstorming sessions organized by him. I would also like to thank all members of the Embedded Systems and CoDesign Lab for supporting me throughout this research.

Last but not least, I would like to thank the Computer Science and Engineering Department, Texas A&M University, for giving me the chance to pursue the graduate degree program at this elite institution.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Rabi Mahapatra and Professor Vivek Sarin of the Department of Computer Science and Engineering and Professor Peng Li of the Department of Electrical and Computer Engineering.

An initial working implementation of the algorithm was provided by Professor Rabi Mahapatra. The experimental analysis in Chapter 5 was carried out with help from Jyotikrishna Dass, PhD Student, Department of Computer Science and Engineering.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

This graduate study was supported by the Department of Computer Science and Engineering, Texas A&M University.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION AND LITERATURE REVIEW

Support Vector Machine (SVM) is a type of supervised machine learning technique, which relies on the geometrical properties of the input data to classify it into various disjoint classes. SVM-based models have been successfully deployed for a variety of classification, regression and prediction tasks in numerous applications such as recognizing objects, categorizing activities, understanding semantics, and interpreting content from unstructured data [1][2][3][4][5][6]. SVM consists of two stages; a training stage which creates a classifier/ regression model based on a given input dataset, and a prediction stage, which refers to classifying or predicting on an unseen sample. The focus of this work is accelerating the training stage.

Given the explosion of powerful smart devices and the popularity of data-driven analytics, there has been increasing interest in offloading some or all of the server's tasks onto edge devices. These edge computing units are more efficient in managing the vast amounts of data being generated, help reduce latency for critical applications and lower dependencies on back-end systems. Consequently, there is an imminent need to develop robust frameworks that would enable edge-layer devices to train models without the interplay of back-end systems. Such frameworks should leverage the availability of multiple edge devices connected over a distributed network. Since edge devices lack the compute capabilities of a high performance server, hardware accelerators are an efficient solution to enable capacity augmentation, with a potential to reduce energy costs.

Training an SVM model is computationally expensive as runtime scales quadratically with the number of input samples. Due to the sequential nature of popular SVM solvers, their direct applicability for classifying large data sets is limited, as they scale poorly with growing sample size. Hence, it is imperative to devise efficient algorithms and accelerators to train SVM models in computing devices. For this, the design must meet strict power constraints, while providing efficient implementations in terms of training time. In recent years, there has been a growing trend to utilize Field Programmable Gate Arrays (FPGA) as dedicated co-processors to accelerate computations [7] [8].

There has been limited research towards developing FPGA-based SVM accelerators in literature [9] [10]. The authors in [11] designed an FPGA-based co-processor for the Sequential Minimal Optimization (SMO) algorithm. As with any sequential solver, its scalability to large datasets is limited. Work has also been carried out to devise novel training algorithms in [12] and [13], with the idea that the algorithms themselves be amenable for designing their respective FPGA-based hardware implementations. However, these algorithms do not provide a distributed framework for multi- FPGA implementations. Consequently, they are not suitable for edge analytics frameworks.

In light of the above, authors in [14] propose QRSVM, a distributed framework for training SVM classifiers. The algorithm is based on applying QR decomposition on the approximated kernel matrix. QRSVM trains a single SVM classifier in parallel over multiple nodes, with negligible communication overhead. The proposed work builds upon the QRSVM algorithm to offer a scalable, distributed solution for SVM training.

This research makes the following contributions:

1. We propose an FGPA-based accelerator for distributed SVM training. To the best of our knowledge, this is the first implementation of a distributed SVM training algorithm on FPGAs.

2. We evaluate the training performance of our proposed accelerator for five SVM benchmarks, and compare with implementations on two other platforms, a high performance CPU cluster and an SoC cluster. The proposed accelerator performs up to 1.81x and 24x faster than the CPU and SoC platforms respectively. Additionally, we achieve a 6.4x and 8.4x reduction in energy consumption, compared to the CPU and SoC platforms respectively .

3. We demonstrate the scalability of the proposed accelerator across a varying number of cores. The proposed design scales linearly with increasing number of cores. Our accelerator achieves the lowest time per training iteration amongst the platforms.

The remainder of this thesis is organized as follows. Chapter 2 presents an overview of non-linear SVM formulation and reviews the distributed QRSVM framework proposed in [14]. In Chapter

3, we design an FPGA-based hardware accelerator for distributed QRSVM training. Chapter 4 describes the workflow to convert hardware designs into FPGA logic. We perform experiments and evaluate the performance of our FPGA-based implementation of the algorithm in Chapter 5. Finally we conclude in Chapter 6.

2. PRELIMINARIES

2.1 Non Linear SVM

For most real-world applications, the input data points aren't directly separable in the input space. Instead of learning a linear classifier, we learn non-linear or *kernel* SVM classifier on the dataset. The original, non-separable data can be separated by transforming the data into a higher dimensional space. In this space, the SVM hyperplane can easily learn a classifier boundary. Figure 2.1 shows such a mapping from the input space to higher dimensional space.

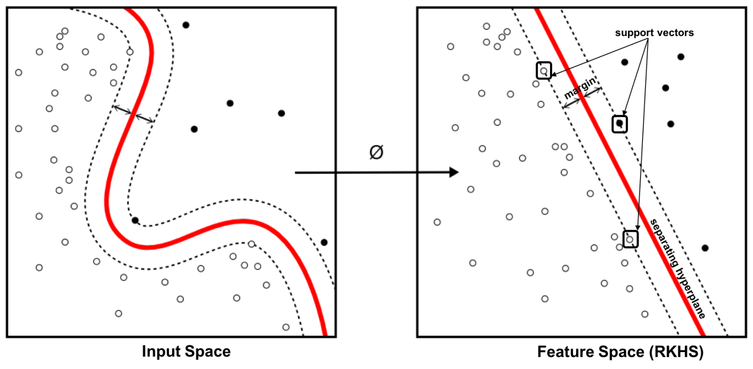


Figure 2.1: Mapping input data to higher dimensional space

Using a function ϕ, the input training sample x is transformed to a higher dimensional space. Often, it isn't possible (or necessary) to know $\phi(x)$ explicitly; instead, we require some measure of "distance" between two samples x and y. The inner product in the transformed space, $\kappa() = \langle\phi(x), \phi(y)\rangle$ provides a measure of distance. For certain spaces, we can compute $\kappa()$ without knowing ϕ. The inner product can be computed quickly, avoiding any explicit coordinate calculation using $\phi()$. Such an approach is referred to as the *kernel* trick. Popular kernels are the sigmoid kernel, polynomial kernel and the Gaussian Radial Basis function (RBF) kernel. For linear SVM, $\kappa()$ is the inner product in the original input space.

For the classification task, we are provided with a set of input training samples, each of whom is associated with a class label.The training dataset is denoted as $D = \{(x_i, y_i), i = 1, ..., n\}$ where x_i is the i^{th} data sample, and $y_i \in \{-1, 1\}$ is the corresponding class label. The training dataset size is n samples, with each sample being a d-dimensional vector.

Equation 2.1 outlines the optimization problem formulated for Kernel SVM, which avoids the explicit formulation of ϕ [14].

$$\min_{\alpha} \frac{1}{2}\alpha^T Z\alpha + e^T\alpha \tag{2.1}$$

$$\text{subject to} \quad L \leq \alpha_i \leq U$$

$Z = (G + D) \in \mathbb{R}^{n\times n}$ is a dense, positive symmetric definite matrix. Here, $G = \{y_i y_j \kappa(x_i, x_j)\}$ is derived from the kernel matrix $K = \{\kappa(x_i, x_j), \forall i, j = 1...n\}$, and D is a diagonal matrix. L and U are the bounds imposed on each Lagrangian multiplier (α_i). $\alpha_i > 0$ indicates that the corresponding point is a support vector. For $\ell_2 - loss$ SVM, $D = (1/2C)I_n$, $L = 0$ and $U = \infty$. The parameter C imposes a penalty for each incorrectly classified sample, and therefore controls the misclassification that can be tolerated by the SVM optimization.

It can be seen that the matrix $Z \in \mathbb{R}^{n\times n}$ grows quadratically with the training samples. For large datasets, working with Z in its native form is computationally expensive. Consequently, efforts are made to obtain a compressed representation of the kernel matrix. Using low-rank optimization methods, one can approximate K to a low rank; i.e. $K \approx AA^T$, with $A \in \mathbb{R}^{n\times k}$ ($k \ll n$). Low-rank approximation has the added advantage of making the kernel matrix separable, similar to the case of linear SVM where $K = XX^T$, $X = \{x_i \in \mathbb{R}^d, i = 1...n\}$.

2.2 QRSVM - A Distributed SVM framework

Authors in [14] have proposed QRSVM, which is a scalable, distributed framework for training SVM by applying the QR decomposition method to the kernel matrix. The framework demonstrates a linear runtime dependency on the input samples, and also shows a high scaling efficiency as the number of cores are increased.

2.2.1 Formulation

The QRSVM framework [14] decomposes the approximated kernel matrix into an orthogonal factor Q, and an upper triangular factor R. In light of the above, Equation 2.1 reformulates to [14]:

$$\min_{\hat{\alpha}} \frac{1}{2}\hat{\alpha}^T\left(RR^T + \frac{1}{2C}I_n\right)\hat{\alpha} + (\hat{e})^T\hat{\alpha} \tag{2.2}$$

$$\text{subject to} \qquad -Q\hat{\alpha} \leq \mathbf{0}_n$$

Here, $\hat{\alpha} = Q^T\alpha$, $\hat{e} = Q^T e$.

The Hessian Z is composed of the following two diagonal blocks:

- a $k \times k$ submatrix, $(RR^T)_k + (1/2C)I_k$
- a diagonal term $(1/2C)I_{n-k}$.

These blocks can be solved independently across multiple computing nodes, thereby parallelizing the SVM training.

2.3 Distributed QRSVM Framework

Distributing the SVM training can be interpreted in two ways, based on the underlying application:

1. One interpretation would be to deploy multiple worker nodes to solve a single, large SVM training problem. In this scenario, we assume that the SVM training is initially present at a single node. The parallelization is achieved by partitioning the dataset into smaller subsets, then distributing the subsets among the working nodes.We then formulate independent sub-problems for these subsets, and subsequently solve them *locally* with intermittent synchronization.

2. A second interpretation, which is particularly relevant for edge computing scenarios, pertains to a multi-device environment. Multiple devices, connected with each other over some

network, gather data independently. The objective here is to train an overall SVM classifier, combining the data contained in all edge devices.

Each of the above scenarios present a slightly different challenge for the SVM algorithm. In the first scenario, scalability and load balance is of utmost significance. This stems from the need to engage maximum possible nodes without losing out on efficiency. In the second scenario, each devices would like to operate on its *local* data, minimizing the transmission of data over the network. Distributed QRSVM framework [14] aims to provide fast, scalable, memory-efficient and communication-efficient approach to tackle the above scenarios.

As discussed in Section 2.2, QRSVM framework motivates towards distributing the SVM training by solving the sub-blocks of the Hessian Z in parallel. To create an end to end distributed QRSVM framework, authors in [14] devised the following two stages: *Distributed QR Decomposition* and *Parallel Dual Ascent*. The following sections examine these stages in greater detail.

2.3.1 Distributed QR Decomposition

The Q and R factors of the entire training dataset can be calculated by combining the Q and R factors of the partitioned training dataset. Authors in [14] show that

$$Q = diag(Q_1, Q_2, ..Q_i.., Q_p) \times Q_g \tag{2.3}$$

and

$$R = R_g \tag{2.4}$$

where, $[{R_1}^T, .., {R_p}^T]^T = Q_g R_g$.

where $\hat{A}_i = Q_i R_i$. The authors also ensure that within each computing device, $k \ll \frac{n}{p} \implies p \ll \frac{n}{k}$.

Q, $Q_i's$ and Q_g are orthogonal and R, R_i's and R_g are upper triangular matrices. The Householder matrix are stored as a set of Householder reflectors, denoted as $\{q\}$. Algorithm 1 [14] explains the distributed QR decomposition process in pseudo-code.

Algorithm 1 Distributed QR decomposition of $\hat{A}$

```
1: $\hat{A} \leftarrow [\hat{A}_1; \hat{A}_2; \cdots \hat{A}_p]$        ▷ Partition Dataset among $p$ cores
2: for core $i$ do
3:     $\hat{A}_i \rightarrow \{q_i\}, R_i$        ▷ In parallel across all nodes
4:     $(R_{gather})_{pk \times k} \leftarrow [R_1; R_2; \cdots R_p]$        ▷ Master node
5: end for
6: $R_{gather} \rightarrow \{q_g\}, R_g$        ▷ Master node
```

2.3.2 Parallel Dual Ascent

Authors in [14] formulate the parallel version of the Dual Ascent method for the distributed QRSVM framework as follows.

Step 1: At compute node i,

$$\hat{\alpha}_i{}^{t+1} = F_i^{-1}(-\hat{\beta}_i{}^t + \hat{e}_i) \tag{2.5}$$

where, $\hat{\beta}^t = Q^T \beta^t$ and

$$F_i^{-1} = \begin{cases} F_1{}^{-1} & \text{if} \quad i = 1 \\ -2C & \text{if} \quad i = 2..p \end{cases}$$

Here,

$$F = -\left(R_g R_g{}^T + \frac{1}{2C} \times I_n\right)$$

is block-partitioned as $F = F_1 \oplus F_2 \oplus ... \oplus F_p$. Here, $\oplus$ is an operator that combines the sub-blocks to generate the entire diagonal matrix.

Step 2: At node i,

$$\hat{\beta}_i{}^{t+1} = \hat{\beta}_i{}^t + \eta^\star(-\hat{\alpha}_i{}^{t+1}) \tag{2.6}$$

Here, the optimal step size $\eta^\star$ defined in [14] is used for faster convergence.

These iterations continue until the dual error ($\|\hat{\beta}_i{}^{t+1} - \hat{\beta}_i{}^t\|_1$) reduces below a predefined threshold. Algorithm 2 [14] outlines the Dual Ascent stage through pseudocode.

Figure 2.2 describes the process flow for Distributed QRSVM. The framework can be summa-

Algorithm 2 Parallel Dual Ascent

```
1: for core $i$ do                                   ▷ Repeat until convergence
2:     Update $\hat{\alpha_i}^{t+1}$                  ▷ In parallel
3:     Update $\hat{\beta_i}^{t+1}$                   ▷ In parallel
4:     Compute $\beta_i \leftarrow Q\hat{\beta}$
5:     Compute $\beta_i \leftarrow max\{0, \beta_i\}$  ▷ In parallel
6:     Compute $\hat{\beta_i} \leftarrow Q^T\beta$
7:     Compute error $||\hat{\beta_i}^{t+1} - \hat{\beta_i}^{t}||_1$
8: end for
```

rized into following stages:

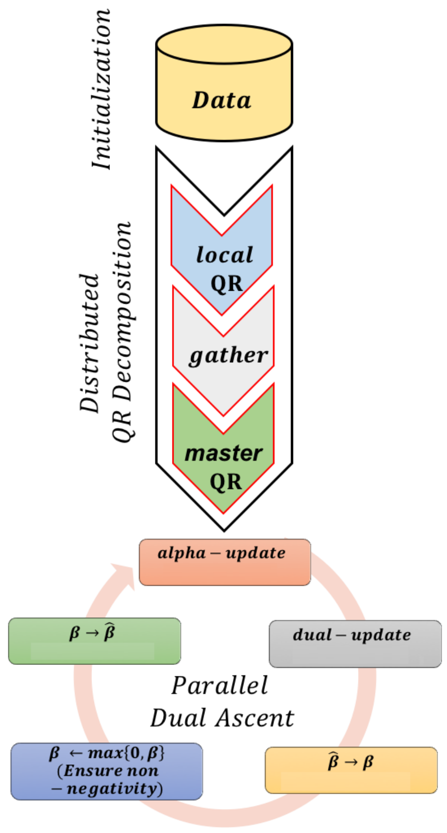

Initialization
Data
Distributed QR Decomposition
local QR
gather
master QR
alpha – update
dual – update
$\beta \rightarrow \hat{\beta}$
Parallel Dual Ascent
$\beta \leftarrow max\{0, \beta\}$ (Ensure non – negativity)
$\hat{\beta} \rightarrow \beta$

Figure 2.2: Process flow for Distributed QRSVM

- *Initialization*: To initiate the computation, we use an approximation technique to compute [15] to compute a low-rank approximation A of the kernel matrix. Subsequently, we distribute A into p equal parts, with each partition associated with a different core. Finally, $\hat{A}_i = diag(y)_i \times A_i$ is used as the training dataset.

- *Distributed QR Decomposition*: The various steps within this stage are given in Algorithm 1. The *local* QR decomposition on partitioned data $\hat{A}_i$ produces a set of Householder reflectors $\{q_i\}$, and an upper triangular matrix R_i. The first k elements of each $R_i's$ are then *gathered* at the master node. The resulting matrix (R_{gather}) at the *master* is further decomposed into q_g and R_g.

- *Parallel Dual Ascent*: As formulated in Algorithm 2, the parallel dual ascent steps (Equation 2.5 and Equation 2.6) are performed at each core i. To impose non-negativity constraint on the dual variable β, $\hat{\beta} \rightarrow \beta$ conversion (Algorithm **??**) is performed. Upon zeroing the negative entries (Step 7, Algorithm 2), $\beta \rightarrow \hat{\beta}$ (Algorithm **??**) is re-transformed before moving to next iteration. The iterations continue until convergence.

3. ACCELERATOR DESIGN

In this chapter, we discuss the design of the proposed FPGA-based accelerator. To motivate a design strategy, we categorize all operations undertaken as part of QRSVM training into two categories: **(1) Computation**, which refers to all operations of high arithmetic intensity, such as matrix multiplication and vector addition among others; and **(2) Communication**, which consist of all operations involving the movement of data among the participating nodes. The authors in [14] argue that QRSVM is a communication-efficient algorithm i.e. the time required for inter-node communication is significantly lesser than per-node computation.

Table 3.1: Communication and Computation Complexity

QRSVM Stage	**Computation**	**Communication**
Distributed QR Decomposition	$O\left(\frac{nk^2}{p}\right)$	$O(k^2)$
Parallel Dual Ascent	$O\left(\frac{nk}{p}\right)$	$O(k)$

Table 3.1 lists the computation and communication complexity for *Distributed QR Decomposition* and *Parallel Dual Ascent* stages of QRSVM. Here, k denotes the rank of the approximated kernel matrix, p denotes the number of processing nodes, and n denotes the total number of training samples. The authors in [14] choose $k \ll \frac{n}{p}$, which makes the communication complexity far lesser than computational complexity.

Hence, we accelerate QRSVM by offloading all *computation* onto the accelerator. By creating efficient hardware architectures for the computations involved, we aim to optimize a large chunk of the algorithm.

For the sake of clarity and simplicity of the design, we assume that the hardware accelerator (FPGA) connects to a CPU. The following discussion shall refer to this CPU as the *host*. The host is responsible for control and coordination of the FPGA device. Figure 3.1 shows the system

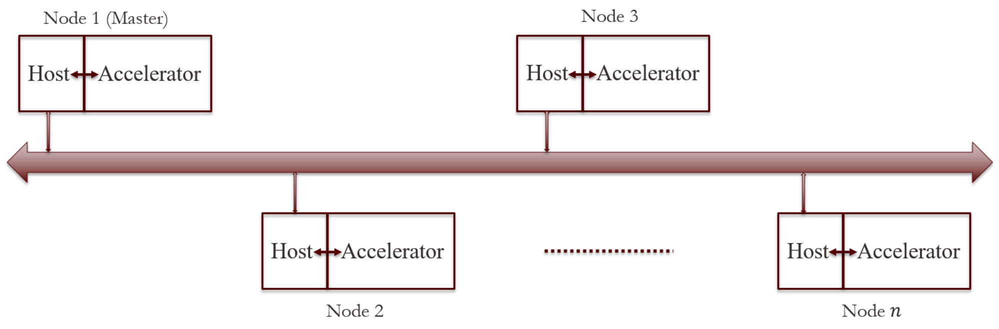


Figure 3.1: System configuration for multiple host-accelerator nodes

configuration for our distributed QRSVM implementation. As seen from the figure, we connect multiple hosts over a network. Each host is paired with an accelerator (FPGA in our case), with the accelerator carrying out *computation* and the host carrying out *communication*.

The following sections illustrate the design of hardware modules that accelerate computations involved in the *Distributed QR Decomposition* and *Parallel Dual Ascent* stages. Subsequent descriptions will refer to each independent computing node (host CPU-FPGA pair) as a *worker* node. One among these nodes is designated as the *master* node.

3.1 Distributed QR Decomposition

Figure 3.2 illustrates the Distributed QR Decomposition process for an 8×3 matrix A across 2 nodes. A brief explanation of each sub-task is given below:

- Initially, the matrix A is partitioned as $A = [A_1; A_2]$. The partitioned dataset is distributed to the two processing nodes. Here, we designate Node 1 as the *master*, and Node 2 as the *worker* node.

- QR Decomposition of the partitioned dataset is carried out *in parallel* across all nodes (**local QR** step in Figure 2.2)

- The local upper triangular R$_i$'s are gathered at the *master* node to form R$_{gather}$.

- Finally, a QR decomposition of R$_{gather}$ is carried out at the *master*.

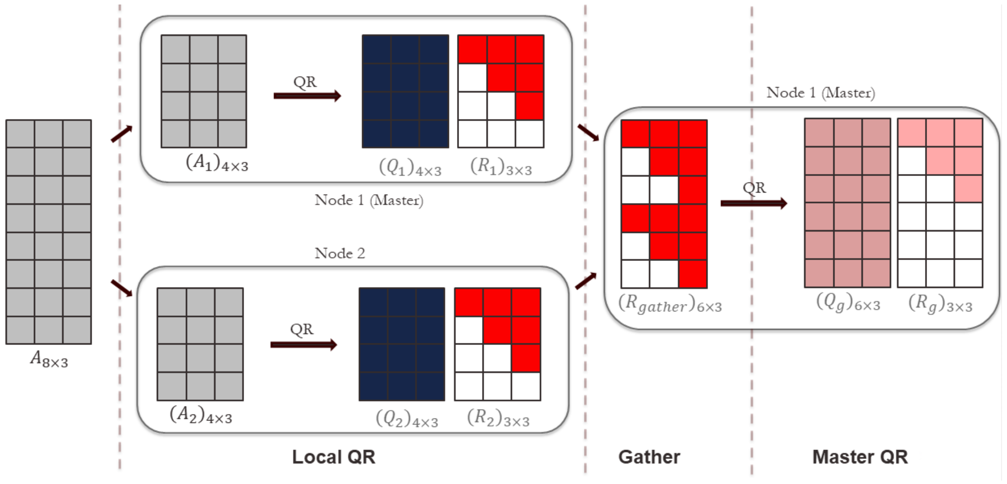


Figure 3.2: Distributed QR Decomposition of $A_{8\times3}$ over 2 nodes

From Figure 3.2, it can be noted that each node (*worker* or *master*) performs an identical operation. The additional task at the master is another QR decomposition. This eliminates the need to create separate designs for the *master* and *worker* nodes.

3.1.1 Kernel Architecture

Algorithm 3 outlines QR decomposition via householder reflectors, which is a modified version from [16]. Here, $< x, y >$ denotes the inner product/dot product of x and y.

Algorithm 3 `QR Decomposition`

```
1: $Q_{\hat{n}\times k}, (\hat{A}_i)_{\hat{n}\times k}$                    ▷ $\hat{n}$ : samples per node
2: for $t \leftarrow 1$ to $k$ do
3:     $q \leftarrow \hat{A}(t:\hat{n}, t)$
4:     $q(t) \leftarrow q(t) + sign(q(t)) \times < q, q >$
5:     $q \leftarrow \frac{q}{<q,q>}$
6:     $Q(t:\hat{n}, t) \leftarrow q$
7:     $\hat{A}_i(t:\hat{n}, t:k) \leftarrow \hat{A}_i(t:\hat{n}, t:k) - 2q < q, \hat{A}_i(t:\hat{n}, t:k) >$
8: end for
```

In Algorithm 3, candidates for hardware acceleration are:

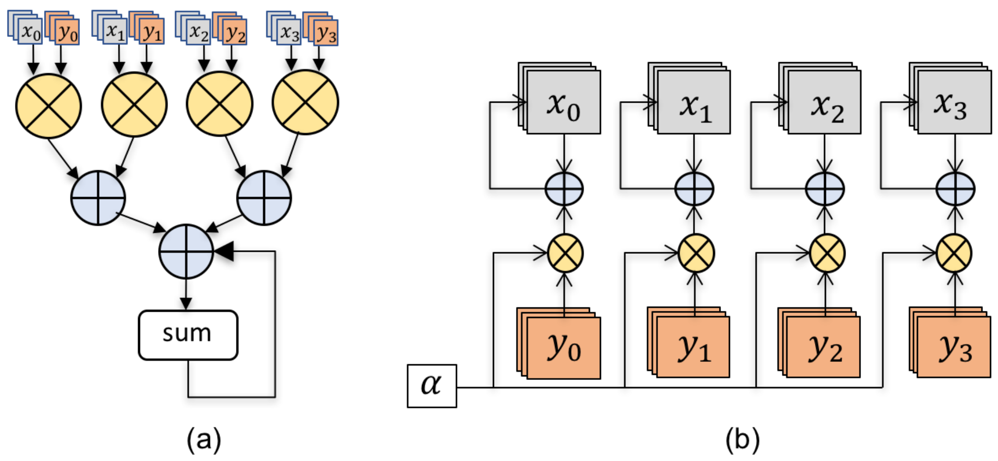


Figure 3.3: Modules for computing (a) Dot product, $sum =< \vec{x}, \vec{y} >$ (b) saxpy, $\vec{x} = \vec{x} + \alpha\vec{y}$. The modules process 4 samples in each pass

- Computing the inner product $< q, q >$.
- Update of $\hat{A}_i$ (Step 7). This can be modeled as a vector-matrix product $< q, \hat{A}_i >$, followed by a rank-1 update: $\hat{A}_i \leftarrow \hat{A}_i - 2q < q, \hat{A}_i >$.

It can be observed that all arithmetic operations can be composed from two BLAS Level - 1 functions:

1. Computing the dot product of vectors $\vec{x}$ and $\vec{y}$, $sum =< \vec{x}, \vec{y} >$
2. Scaled addition of two vectors $\vec{x} = \vec{x} + \alpha\vec{y}$. This operation is referred to as *saxpy* in the subsequent text.

The high degree of data parallelism inherent to these operations can be exploited to develop vectorized hardware implementations for the same. FPGA's are amenable to Single Instruction Multiple Data (SIMD) style implementations, given their reconfigurable nature and high internal memory bandwidths. Figure 3.3 shows the architectures for computing dot product and saxpy.

The module to compute dot product is a binary reduction tree as shown in Figure 3.3(a). At the tree's leaf nodes, respective entries from x and y are multiplied. Products are pairwise summed along the tree branches, and the result is stored in a register (denoted by sum).

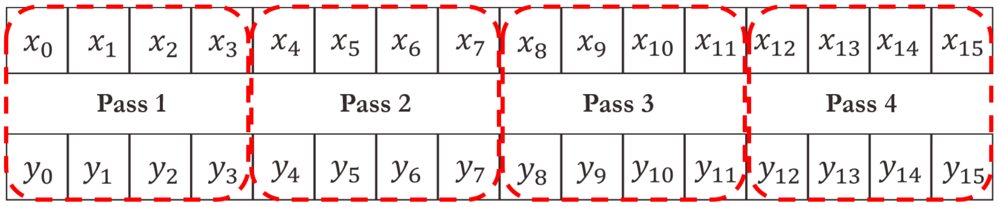

x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15}
Pass 1
Pass 2
Pass 3
Pass 4
y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8 y_9 y_{10} y_{11} y_{12} y_{13} y_{14} y_{15}

Figure 3.4: Illustrating batching for vectors of length 16

For longer vectors, we group x and y into batches. Figure 3.4 illustrates the batching operation when x and y have length 16. In this case, sum serves as an accumulator for the partial product.

The module for computing the saxpy operation is illustrated in Figure 3.3(b). The architecture consists of a multiplier that computes αy, followed by an adder that overwrites x with $x + \alpha y$. Similar to the dot product, longer vectors can be processed through the batching technique shown in Figure 3.4.

3.1.2 Pipelined Kernel Design

The kernel modules proposed above can be pipelined in order to increase throughput. To pipeline the design, we combine all arithmetic units at the same depth into one pipeline stage. This allows a given stage to process new samples without waiting for completion of all succeeding stages. Figure 3.5 shows the pipelined architectures for dot product and saxpy.

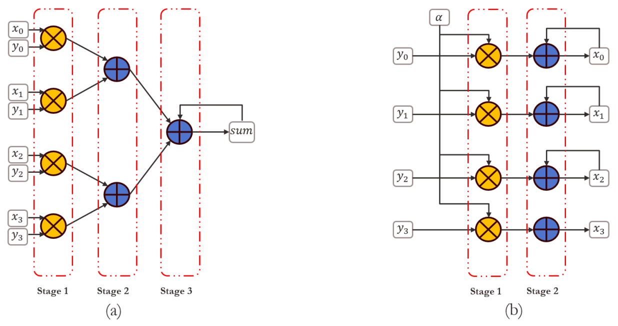


Figure 3.5: Pipelining architectures for (a) Dot Product and (b) saxpy

For the dot product module, each level of the reduction tree can be treated as an independent pipeline stage. Similarly, for the saxpy operation, the adders are grouped in one stage, with the multipliers in the second stage.

To determine the maximum number of pipeline stages that can be deployed, we examine the interface between the accelerator and external memory. Figure **ref** shows an N-bit wide data bus connecting the FPGA and DDR memory. Let B denote the bit width of a single entry (32 for single precision floating point, 64 for double precision). For the dot product module, the maximum number of leaf nodes, $W = \lfloor \frac{N}{B} \rfloor$. Consequently, the number of pipeline stages $D = \log_2 W$. For the saxpy operation, we can deploy a maximum of W adders and multipliers in parallel. The pipeline depth remains constant at 2.

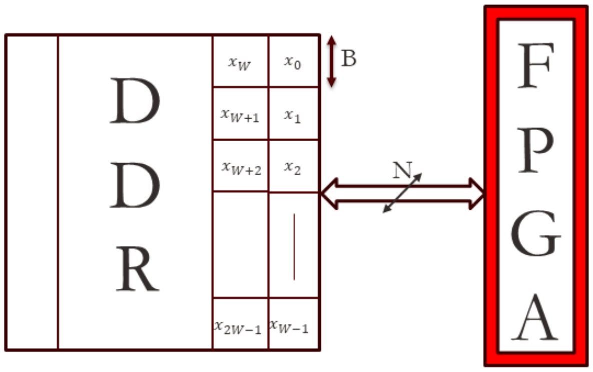


Figure 3.6: Illustrating the memory layout

Since the BLAS operations in Algorithm 3 operate on columns of the partitioned dataset $\hat{A}_i$, the dataset is stored in column major form i.e. $\hat{A}_i(:,1)$ is stored first, followed by $\hat{A}_i(:,2)$, and so on. Storing data in column major format makes the memory access pattern contiguous, reducing the memory access time. Since the hardware modules executed in a pipelined manner, they access W column elements in each clock cycle. Consequently, while storing data in memory, the column length must be an integral multiple of W. Otherwise, data points from two different columns may end up getting processed in the same batch. To ensure this, we pad columns with requisite number of zeros.

Pipelining the functional modules changes the frequency and nature of memory accesses, which may lead to subtle performance bugs. For example, in the saxpy operation ($x = x + \alpha y$), consider the operation of the adder that performs the addition $x + \alpha y$. In each pass, the adder performs the following three operations:

1. Reading x from memory;

2. Adding x with αy;

3. Writing updated x to memory

Since the kernel execution is pipelined, each of these operations occur in a single clock cycle. Figure 3.7 illustrates the timing diagram for pipelined operation of one such adder (highlighted in blue).

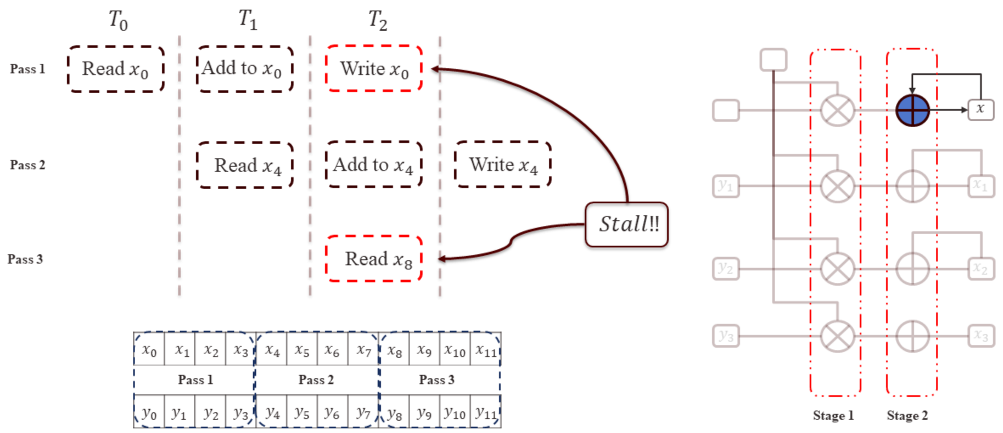


Figure 3.7: Pipelining the saxpy operation

In the first two passes, the adder performs the read/add/write operations without any stalls. In the third pass, however, the adder attempts to write to x_0 while simultaneously reading from x_8.

x is stored in off-chip DDR memory, which is configured as half-duplex, hence the concurrent reads/writes are serialized. This leads to frequent pipeline stalls, reducing kernel throughput.

To overcome this limitation, we turn our attention to Block Random Access Memory(BRAM), which refers to reconfigurable memory available in the FPGA fabric. On most modern FPGA's, Block RAM ranges from a few Kilobytes (KB) to a few Megabytes (MB) in size. Compared to off-chip DDR memory, Block RAM offers lower memory access times. Moreover, BRAM can be configured in full-duplex mode, supporting concurrent reads and writes to the memory. Figure 3.8 shows a hardware module (referred here as IP module) connected to two different kinds of memory: half-duplex DDR and full-duplex Block RAM.

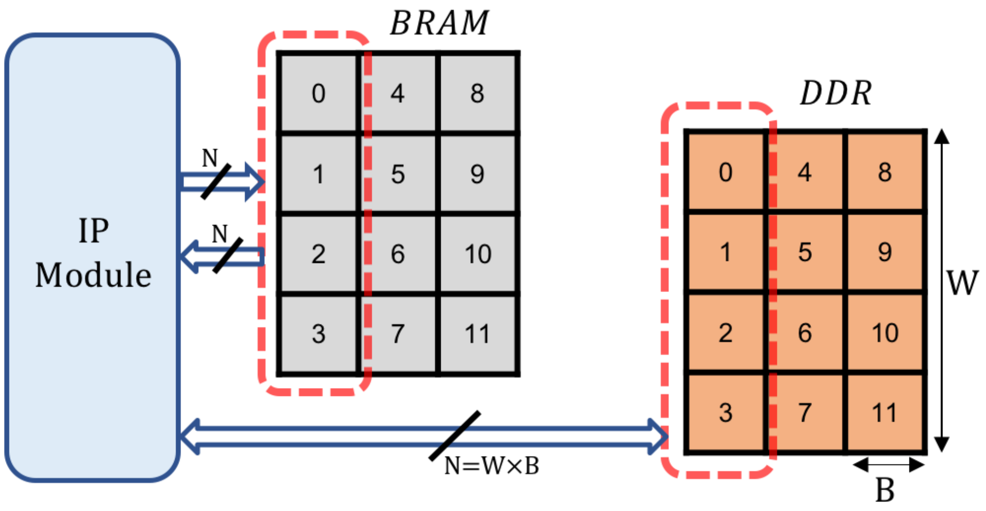


Figure 3.8: Memory interface for Block RAM (Full-duplex) and DDR (Half-duplex)

In the saxpy operation, $\vec{x}$ is read and written simultaneously, while $\vec{y}$ is only read. Hence, we store $\vec{x}$ in full-duplex BRAM while $\vec{y}$ is allocated to half-duplex DDR. Since Block RAM is of limited size, we must use BRAM to cache data structures. Algorithm 4 illustrates the caching strategy for the rank-1 update of $\hat{A}_i$ (Step 7 in Algorithm 3). The data subscript represents memory allocation.

Algorithm 4 `Rank-1 update of` $\hat{A}_i$ `(at step` t`)`

```
1: $Q_{\hat{n}\times k}, (\hat{A}_i)_{\hat{n}\times k}$
2: q $\leftarrow Q(t:\hat{n}, t)$
3: for $i \leftarrow t$ to $k$ do
4:     $a_{BRAM} \leftarrow A(t:\hat{n}, i)$                    ▷ Load into BRAM
5:     Compute $sum = q^T a_{BRAM}$
6:     $a_{BRAM} \leftarrow a_{BRAM} - 2 \times sum \times q$
7:     $A(t:\hat{n}, i) \leftarrow a_{BRAM}$                    ▷ Write to DDR
8: end for
```

3.2 Parallel Dual Ascent

As shown in Figure 2.2, each iteration of the parallel dual ascent (DA) stage comprises of the following steps:

1. Updating variables $\hat{\alpha}$ and $\hat{\beta}$
2. Transforming $\hat{\beta}$ to β
3. Ensuring non-negativity on β
4. Converting β to $\hat{\beta}$
5. Estimating the dual error for convergence check

The rule for updating $\hat{\alpha}$ is defined in Equation 2.5. It can be seen that the update requires a vector subtraction, followed by pre-multiplication by F^{-1}.

Figure 3.9 shows the structure of F^{-1}. Based on the figure, the following can be observed about the structure of F^{-1} at each node:

- At the *master* node, F_1^{-1} consists of a dense $k \times k$ block, followed by a diagonal sub-matrix
- At all other nodes, F_i^{-1} is a diagonal matrix

Thus, at the *master* node, the top k elements of $\hat{\alpha}$ can be obtained by solving

$$LL^T\hat{\alpha}(1:k) = (\hat{e} - \hat{\beta})(1:k)$$

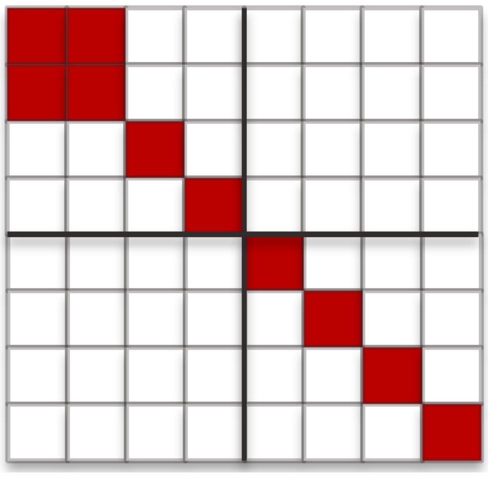


Figure 3.9: Structure of F_1^{-1}. The red boxes represent non-zero entries

where L is the Cholesky factor of F_1 $(F_1 = LL^T)$ [16]. Elsewhere, the matrix multiply can be computed by scaling $(\hat{e} - \hat{\beta})$ by $(-2C)$.

The update rule of $\hat{\beta}$, described in Equation 2.6, indicates it being a saxpy operation. As discussed before, $\hat{\beta}$ must be stored in full-duplex Block RAMs to prevent pipeline stalls.

Each Dual Ascent iteration requires converting between $\hat{\beta}$ and β. From Equation 2.2, $\beta = Q\hat{\beta}$ and $\hat{\beta} = Q^T\beta$. We store Q as a set of Householder reflectors $\{q_i\}$. Therefore, multiplying Q with $\hat{\beta}$ is detailed in Algorithm 5. By reversing the start and end indices, we can compute $\hat{\beta} = Q^T\beta$ [16]

Algorithm 5 Computing $\beta = Q\hat{\beta}$

```
1: $Q_{n\times k}, (\hat{\beta})_{n\times 1}$
2: for $t \leftarrow k$ to 1 do
3:     $q \leftarrow Q(:,t)$
4:     $\hat{\beta} \leftarrow \hat{\beta} - 2q < q, \hat{\beta} >$
5: end for
6: $\beta \leftarrow \hat{\beta}$
```

It can be seen that converting between $\hat{\beta}$ and β is similar to the update of $\hat{A}_i$ in QR Decomposition (Algorithm 3, Step 7). Each iteration of the ***for*** loop comprises of an inner product, followed by a saxpy operation.

We observe that the dual ascent computations comprise calls to the same BLAS Level-1 kernels designed previously. This provides the opportunity to reuse the synthesized modules for dual ascent operations. For this to correctly work, we must adhere to the memory layout proposed earlier. This can easily achieved through column major storage of $\hat{\alpha}$, $\hat{\beta}$ and $\hat{e}$.

At the end of each iteration, we compute the iteration error $\left\|(\hat{\beta}^{k+1} - \hat{\beta}^{k})\right\|_1 = \sum_{i=0}^{n} |(\hat{\beta}^{k+1}(i) - \hat{\beta}^{k}(i)|$. The architecture to compute iteration error is shown in Figure 3.10. The design is similar to the module for computing dot product, with the leaf nodes configured to compute the difference of absolute values.

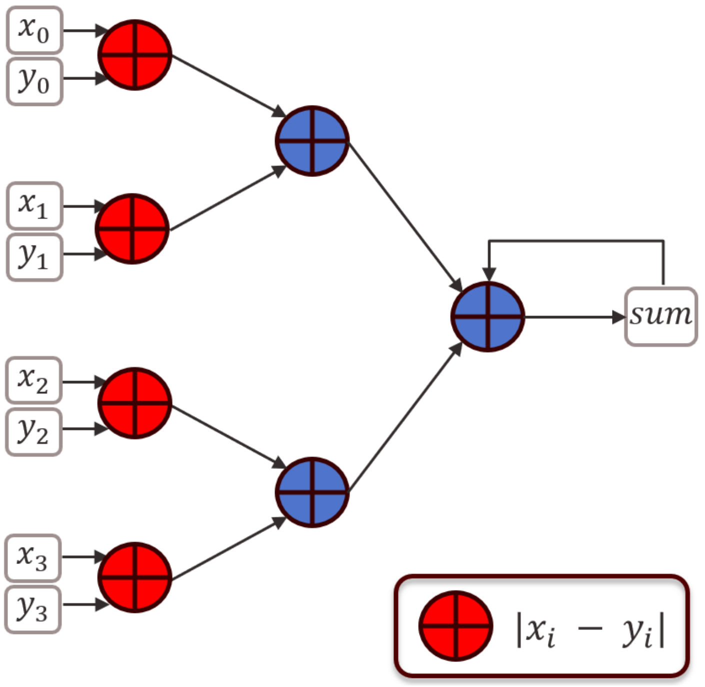


Figure 3.10: Computing the iteration error $\left\|(\hat{\beta}^{k+1} - \hat{\beta}^{k})\right\|_1$

4. IMPLEMENTATION

In this chapter, we detail the work flow to implement a desired hardware module onto an FPGA. The process of implementing a desired design on to the FPGA can be broken up into three different stages.

- First, the hardware modules must be described through through special purpose hardware-description code called Register Transfer Level (RTL) code.
- After creation of RTL, the hardware modules must be encapsulated into an IP core. The IP core should include appropriate interfaces, in order to connect with the host CPU and various memory elements.
- Finally, software routines must be written to control the IP core's functions.

For FPGA synthesis and implementation, we use Amazon Web Services' EC2 F1 instances [8] for. F1 instances are cloud-based, compute optimized instances containing Field Programmable Gate Arrays (FPGAs). The instances come equipped with all requisite development tools to create custom hardware accelerators. Each F1 instance features [8]:

- High frequency Intel Xeon E5-2686 v4 (Broadwell) processors
- 16 nm Xilinx UltraScale Plus XVU9P FPGAs
- Local 64 GiB DDR4 ECC protected memory per FPGA
- Dedicated PCI-Express x16 interface between FPGA and host CPU

The instances are available in two categories:

1. *f1.2xlarge* : This instance provides the user with 8 Intel Xeon CPUs and 1 Xilinx FPGA.
2. *f1.16xlarge* : This instance provides 64 CPUs and 8 FPGAs

The remainder of this chapter presents a brief overview of the workflow described above.

4.1 Creating Register Transfer Logic (RTL)

To synthesize Register Transfer Level (RTL) code for our design, we use the Xilinx Vivado High Level Synthesis (HLS) [17] tool. HLS automatically converts high level language (C, C++ and System C) specifications into RTL code. This eliminates the need to manually write RTL, thereby reducing the time for IP creation.

We illustrate the creation of RTL using High Level Synthesis by taking an example C function *mult*. The *mult* function takes its input as two arrays A and B, each of size N. The function computes the element-wise product of A and B, and stores it back in A; i.e.

$$A[i] \leftarrow A[i] \times B[i] \qquad i \in \{0, \ldots, N-1\}$$

The C code for *mult* is provided below:

```
void mult(float A[N], float B[N])
{
    int i;
    for(i=0; i < N; i++)
        A[i] = A[i] * B[i];
}
```

The function for which the RTL code needs to be generated is called the *top* function. The Vivado HLS compiler accepts a single *top* function to generate RTL code. Here, we set the *top* function as *mult*. For more complex functions, the *top-level* function can make calls to other functions. In such cases, HLS generates RTL code for all functions called from *top*.

Every input argument of the *top* function is synthesized as an independent input port. In our case, the arguments A and B are synthesized as array interfaces. In addition to all input arguments, HLS auto-generates a *return* port for the hardware module. This port connects the IP to the *host* CPU. The host can monitor IP status and control information, as well as exchange parameters with the IP over this port.

4.1.1 Design Optimizations

The RTL, synthesized through high level (C/C++/System C) code, can be optimized to reduce operation latency, increase kernel throughput or augment memory bandwidth. To achieve these optimizations, HLS specifies in-built *directives* and *pragmas* [18], which are compiler options applied to specific portions of the input code. These are applied to data structures, loops or function arguments to optimize the generated RTL code. A few such directives and their corresponding optimizations are listed below [19]:

- ***HLS PIPELINE***: This directive helps pipeline the execution of loop-based iterative code. For the example *mult* function, this directive can be applied to the *for* loop. Applying this directive pipelines the computation $A[i] \times B[i]$. As a result, the hardware module begins computing $A[i+1] \times B[i+1]$, as soon as $A[i] \times B[i]$ is done computing.

- ***HLS UNROLL***: This directive unrolls an iterative loop by a specified factor. Here, loop unrolling refers to the process of executing multiple loop iterations concurrently. As a result of unrolling, the number of loop iterations are reduced by a factor equal to the degree of unrolling. For example, unrolling the *for* loop in *mult* by a factor of two, would synthesize RTL for two floating point multipliers. These multipliers would simultaneously operate on elements from A and B. Doing so cuts the *for* loop iterations by half.

- ***ARRAY RESHAPE***: This directive groups together multiple array elements as a single element. As the IP now operates on an element of a larger bit width each clock cycle, the IP core's memory bandwidth is increased. Vectorized implementations of loop operations can be obtainedby applying this directive in conjunction with **HLS UNROLL**.

4.1.2 Vivado HLS Workflow

While describing the modules through high level code, a robust test suite must also be created to ensure the correctness of the synthesized hardware. In light of this, the workflow for creation of RTL code from C/C++ code involves the following four stages:

1. *C Simulation :-* At this stage, the design is specified through C/C++ code. Alongside, a test program is created to validate the code that specifies the design. This test program compares the desired output (output of the design code) against a golden reference. If the design code generates the correct output, the test passes and the test program returns a 0. To indicate test failure, the test program returns any other integer value.

2. *C synthesis :-* In this stage, the RTL code describing the hardware modules is created from the C/C++ design code. As described before, we apply directives to optimize the generated RTL. The RTL creation is done for a target FPGA, which is specified at synthesis time. The outcome of this stage is the RTL code, along with an estimate of the chip area utilization. The area estimate is specified by the number of chip elements (LUT, FF, BRAM and DSP) utilized to synthesize the design.

3. *Cosimulation :-* In the cosimulation stage, HLS auto-generates an RTL test bench from the C/C++ test program. This test bench is used to verify the functionality of the RTL code created during the previous stage. Cosimulation eliminates the need to manually write an RTL test bench, generating it from high level code instead.

4. *Export IP :-* Once the RTL is verified in through cosimulation, it is packaged into an IP core. This IP core can be directly included as a pre-built block in third party applications.

4.1.3 Creating QRSVM IP

We described the hardware kernels for QRSVM computations in Chapter 3. We use the workflow detailed in the previous section to synthesize RTL code for the hardware modules. These modules are packaged into a single IP core, denoted as QRSVM IP. Figure 4.1 shows the QRSVM IP block.

The QRSVM IP must connect with the host CPU, on-chip Block RAM and off-chip memory (i.e. DDR). For the same, we use the Advanced eXtensible Interface (AXI) [20] protocol to create IP interfaces. Vivado HLS natively supports AXI-based interfaces for interfacing IP cores. Interfaces for the QRSVM IP are described below:

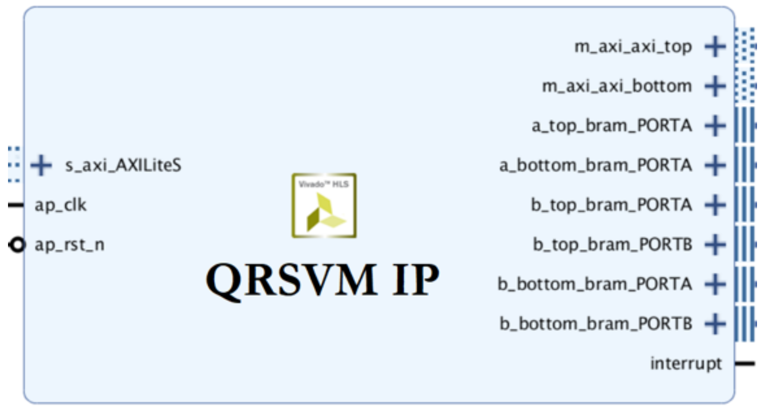


Figure 4.1: QRSVM IP Block

- The IP interfaces with the host CPU through an *AXIlite* port. In this configuration, the host is Master while the IP runs in slave mode. The host and IP core exchange control and status information over this interface port. Additionally, the port allows the host to read/write parameters (such as pointers to input data, return values etc.) to/from the IP respectively.

- The on-chip BRAM caches are synthesized with a *bram* interface. The maximum bus width supported by this interface is 1024 bits. As we use double precision floating point numbers, maximum parallel compute units $W = \lfloor \frac{N}{B} \rfloor = 16$.

- All references to off-chip memory (DDR) are made over an *AXI Master* port. With this interface, the IP assumes the role of the bus controller and can directly issue memory references without host mediation. For uniformity, we set the data bus width to 1024 bits.

4.1.4 Increasing Memory Bandwidth

As illustrated in Figure 3.6, the throughput of the kernel modules is largely governed by the data bus width, N. The bus width directly determines the number of parallel functional units W ($W = \lfloor \frac{N}{B} \rfloor$). Doubling N would double W, which in turn would increase throughput.

However, there are limitations to the maximum bus width for a given interface. For example, the *bram* interface supports a maximum data bus width of 1024 bits, while the *AXI* interface supports up to 2048 bits.

To overcome these limitations, we create multiple independent interfaces to increase memory bandwidth. Subsequently, the computations can be split into smaller, independent sub-problems to utilize these interfaces. For instance, computing the dot product of vectors x and y can be regrouped into smaller sub-problems, as shown below:

$$< x, y >= x^T y = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = x_1^T y_1 + x_2^T y_2$$

Thus, we can parallelize the computation of dot product by splitting x and y into two equal halves, and computing each partial product in parallel.

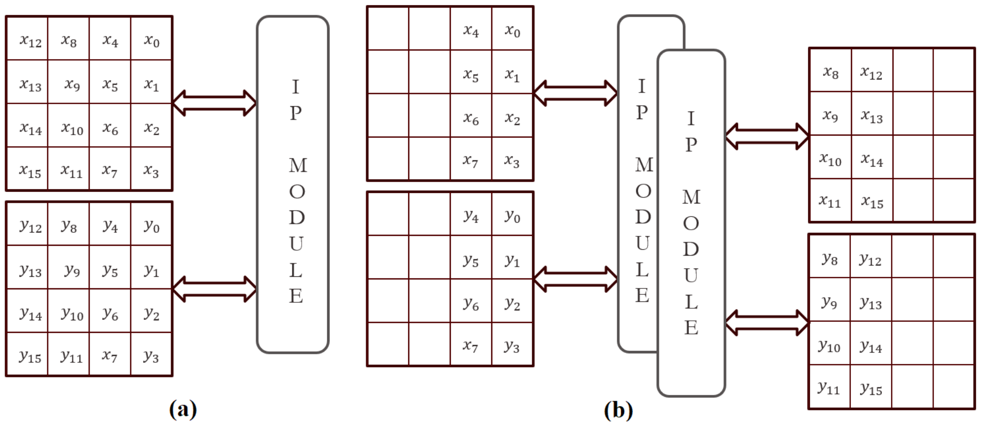


Figure 4.2: (a) Computing $< x, y >$ with 2 memory interfaces (b) Computing $< x, y >$ with 4 memory interfaces

In Figure 4.2(a), we illustrate computing $< x, y >$ with 2 memory interfaces. Since $N = 16$ and $W = 4$, we would require 4 passes to arrive at the final result. In Figure 4.2(b), we utilize additional memory interfaces to parallelize the problem. We begin by splitting x and y between the available memories, with the first half stored in one location and the second half in the other location. Alongside, we synthesize an additional IP module, and connect it with the appropriate memories. With this arrangement, each IP module works on its respective sub-problem. Therefore, both modules can operate in parallel, and only require 2 passes to arrive at the solution.

Thus, as long as the read/write requests are satisfied simultaneously, i.e. no two memory references direct to the same interface, the memory bandwidth or throughput can be effectively doubled. In our design, we create two independent *AXI Master* and *bram* interfaces for accessing memory. As all QRSVM arithmetic occurs on matrix columns, we split each column vector into two halves, namely *top* and *bottom*. As seen in Figure 4.1, each memory interface is also split into *top* and *bottom* halves. The interfaces access their respective halves in parallel, thereby doubling the effective memory bandwidth.

4.2 FPGA Synthesis and Implementation

Upon creation of the QRSVM IP core, it must be synthesized and implemented onto a target FPGA chip. The IP contains appropriate interfaces for the host CPU, on-chip Block RAM and off-chip DDR. Consequently, appropriate connections must be made to connect the IP core and other components. This section illustrates the process of synthesizing and implementing the QRSVM IP onto the Xilinx Virtex FPGA, available as part of the EC2 F1 instance.

4.2.1 FPGA Block Design

The F1 environment supports the Xilinx Vivado suite for synthesis and implementation of FPGA designs. The IP is integrated with other components to form the FPGA *block design*. The block design can be assembled through a GUI interface or Tcl-based command arguments. All block designs for F1 instances are divided into two parts, namely:

- **Custom Logic (CL)**, which denotes the user-defined hardware blocks. In our case, CL refers to the QRSVM IP.
- **AWS Shell (SH)**, which is a pre-built, parameterizable IP block that connects the user logic, i.e. **CL**, with the host CPU and off-chip DDR. The type and number of these interfaces can be configured through the SH. Additionally, the SH provides the CL with a configurable input clock.

Figure 4.3 shows the FPGA block design with CL and SH. As seen in the figure, the SH is connects with the CL's *AXIlite* port. Thus, all communications with the CL are handled by the SH. Additionally, the SH provides two AXI-based DDR interfaces, namely **S_AXI_DDRA** and **S_AXI_DDRB**.

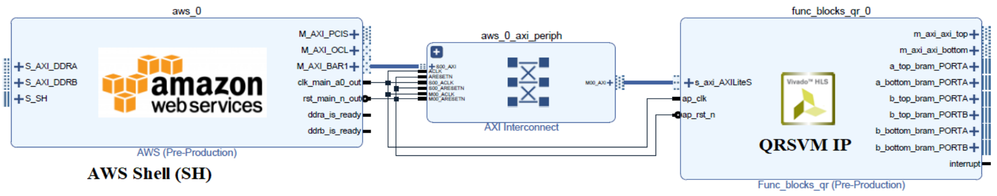


Figure 4.3: AWS Shell with QRSVM IP (CL)

Upon configuring the CL and SH, we must add the Block RAM modules. As discussed in Chapter 3, the Block RAM is configured in full-duplex mode to achieve high throughput with pipelined kernels. Block RAM can also be added (and configured) through the GUI interface. Figure 4.4 shows the QRSVM IP, connected with two Block RAMs. The Block RAMs are connected with the QRSVM IP in full-duplex mode. This is indicated by the two ports on the BRAM, **PORTA** and **PORTB**, connecting with the IP. These two ports can be independently accessed, thereby enabling simultaneous reads/writes to the BRAM.

Upon adding the Block RAMs, the QRSVM IP must be interfaced with the off-chip DDR. As discussed above, the Shell (SH) provides access to the DDR via an AXI interface. The QRSVM IP is synthesized with AXI Master ports, which enable direct memory access from the IP. The completed block design is shown in Figure 4.5.

Upon validation of input parameters, the block design is synthesized and implemented onto a target FPGA chip. For the F1 instances, the target FPGA is the Xilinx Virtex Ultrascale FPGA ***xcvu9p-flgb2104-2-i***. The output from a synthesis is a bitstream, which can be downloaded onto the FPGA to program the block design.

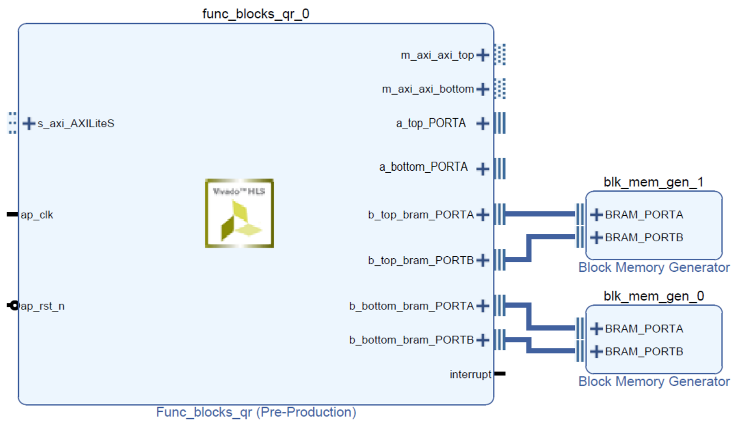


Figure 4.4: Interfacing QRSVM IP with Block RAM

4.3 Controlling the QRSVM IP

Upon synthesizing and implementing the FPGA block design, we must create a host program which exploits the FPGA for training an SVM classifier using QRSVM algorithm. The FPGA is attached with the host CPU through a PCIe slot. Therefore, the host program must direct all status/control information over the PCIe bus. In addition, the FPGA off-chip memory (DDR) is also accessible to the host through the PCIe slot.

The F1 instance provides a Software Development Kit (SDK) [21], which provides the requisite helper functions for all PCIe communications. For the same, the SDK provides header files encapsulating PCI library functions. These functions can be invoked in the host routines to communicate with the FPGA.

The salient steps for distributed QRSVM with multiple FPGA-CPU nodes, can be summarized as follows:

1. The hosts recognize and register their corresponding FPGA as a PCI device, using the library functions provided in the SDK. At this stage, the hosts load their respective partition of the training dataset into the FPGA DDR memory. It is to be noted that column major storage must be adhered to while loading the dataset.

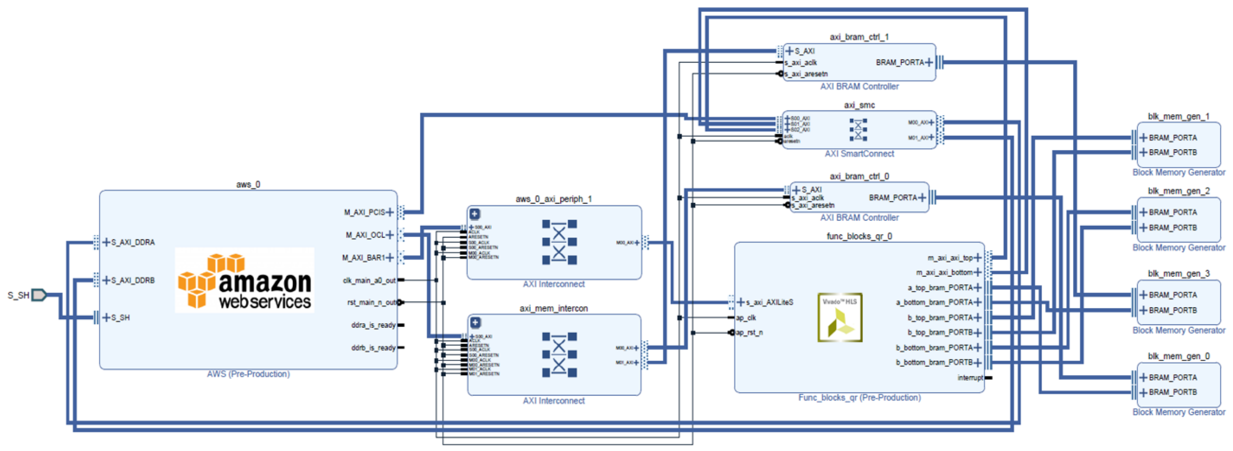


Figure 4.5: FPGA Block Design for QRSVM IP

2. The hosts initialize their respective FPGAs by sending control parameters such as pointers to dataset locations, learning rate etc. These communications occur over the QRSVM IP's *AXIlite* port.

3. Through appropriate control commands, the hosts direct the FPGAs to perform required computations (dot product, saxpy etc.). The operation status is monitored by polling the AXIlite port's status register.

4. Whenever a communication requirement arises, the hosts read the appropriate data from their respective FPGAs. Subsequently, the hosts transfer data through MPI *Gather* and *Scatter* calls.

5. EXPERIMENTS

To illustrate the benefits of the proposed FPGA accelerator, we compare the accelerator against two other experimental setups: **(1)**A high performance CPU cluster, and **(2)** An embedded SoC cluster, typically marketed as an edge computing solution. For all three platforms, we evaluate and compare the time taken to train an SVM classifier using QRSVM method. Following is a brief description of the platforms and their salient features:

- **FPGA Accelerator**: Amazon EC2 F1 [8] instance was used to synthesize the FPGA accelerator. As outlined in Chapter 4, the *f1.16xlarge* instance features eight Xilinx Virtex UltraScale+ VU9P FPGAs . The FPGAs connect to the host CPUs through a PCIe slot. To elicit the benefits of FPGA acceleration, we offload all computation to the FPGA. The host processors are responsible for program flow control, and handling communication through a message passing protocol such as MPI.

- **CPU Cluster**: The CPU cluster is comprised of Intel Xeon E5-2686 v4 high performance processors. These processors are also available as part of the F1 instances. It is to be noted that these processors act as the host for the FPGA-based accelerator described above. In subsequent experiments, we use CPUs available in a single *f1.16xlarge* instance.

- **SoC Cluster**: The SoC platform is a commercially available HPE ProLiant m800 [1] server cartridge. Each cartridge contains four TI KeyStone II 66AK2H SoCs operating at 1.0 GHz. Each SoC contains four ARM A15 processors alongside eight TI C66x DSP cores. However, we only utilize the ARM processors, in order to maintain a homogeneous computing environment across all hardware platforms.

In our experiments, the notion of a node is unique to each platform. For the FPGA accelerator, a node refers to a CPU-FPGA pair; i.e. one core of the Intel Xeon processors, attached to one

[1]`https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c04500667`

Xilinx FPGA. As stated previously, we offload all computation to the FPGA. Ideally, we wouldn't require a high performance, Intel Xeon-like CPU as the FPGA host, since the CPU performs no arithmetic. For the CPU cluster, we define a single core of the Intel Xeon processors as a node. In the case of the SoC cluster, a single core of the ARM A15 is referred to as a computing core.

For the CPU and SoC clusters, we run an MPI-based C++ implementation of QRSVM, using the Armadillo library [22] integrated with LAPACK/BLAS for linear algebra calculations.

Table 5.1 lists the binary classification datasets chosen to evaluate QRSVM. These datasets were taken from the LIBSVM repository [2]. We choose the first 4 datasets in their entirety. SUSY is a treated as a special case, wherein we choose $2M$ random samples from the original $5M$ for weak scalability tests. We use Memory Efficient Kernel Approximation (MEKA) [15] to obtain the *k*-rank approximation of the kernel matrix.

Table 5.1: Dataset Description

Benchmark	**Application**	**#samples (n)**	**#features (d)**	***k*-rank**
MNIST	Image	60,000	780	128
Skin	Health	200,000	3	64
Webspam	Email	350,000	254	128
Covtype	Geography	464,810	54	64
SUSY	Physics	2,000,000	18	128

5.1 FPGA Synthesis

We use Vivado High Level Synthesis [17] to synthesize our FPGA design. The HLS synthesis estimates indicate that the QRSVM IP can operate up to clock speeds of ~ 200 MHz. However, we adopt a conservative clock of 125 MHz for the sake of obtaining timing closure. Table 5.2 lists the post-implementation area utilization.

The post-implementation results indicate the disproportionate utilization of Block RAM, as opposed to other resources on chip. Block RAM's are vital for the proposed design, as they help

[2] `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html`

Table 5.2: Utilization for FPGA *Xilinx Virtex xcvu9p-flgb2104-2-i*

Resource	**BRAM**	**DSP**	**FF**	**LUT**
Total	1405	1221	545248	449113
Available	2160	6840	2363536	1181768
Utilization(%)	65	18	23	38

create full-duplex, high bandwidth memory to serve vectorized, pipelined kernels. Consequently, the availability of Block RAM constraints the maximum number of data points that can be processed at each node. The *xcvu9p-flgb2104-2-i* FPGA (FPGA in the EC2 F1 instance) contains ~7 MB of Block RAM. Out of the available 7 MB, we use 4 MB to synthesize two BRAM blocks of 2 MB each. These blocks cache the dual variables $\hat{\alpha}$ and $\hat{\beta}$ at each node. Owing to this size limitation, each FPGA node can process a maximum of 256K data points.

5.2 Performance Analysis

In this section, we compare the performance of the three platforms with regards to training time, scalability and energy consumption while executing the QRSVM algorithm. We evaluate the platforms for a varying number of cores. The maximum cores are limited by the availability of FPGA units in a single Amazon *f1.16xlarge* instance. Therefore, we restrict the evaluation of our platform upto 8 FPGA nodes. It is to be noted that permitting availability, the framework can be easily evaluated for a higher number of cores.

5.2.1 Training Time Analysis

Table 5.3 shows the distributed QRSVM training times for the given benchmarks on all three hardware platforms. The training time for QRSVM can be calculated as the sum of the times taken for *Distributed QR Decomposition* and *Parallel Dual Ascent*. Since the parallel dual ascent is iterative in nature, it accounts for a larger share of training time among the two stages [14]. Additionally, the algorithm design ensures that for a given step size η^*, the number of parallel dual ascent iterations (t) remains constant for a given number of cores p across all platforms. Therefore, training time is largely governed by the time taken for each dual ascent iteration, i.e.

the *per-iteration time* achieved on a given platform.

We observe that for almost all the datasets, the proposed FPGA design trains faster than both the CPU-based and SoC-based clusters. By creating efficient hardware designs for the computationally dominant steps of QRSVM, the FPGA accelerator achieves the lowest *per-iteration time* among the three platforms. Consequently, the training time for the FPGA accelerator is the lowest.

It is worth noting that for the MNIST dataset, the training time for the FPGA accelerator at $p = 8$ is higher than that of the CPU-based cluster. MNIST is a small dataset with $n = 60K$ samples. As we go beyond $p = 2$ cores, the per-node computation reduces to an extent where inter-node communication begins affecting the overall runtime. Consequently, it can be argued that deploying cores beyond $p = 2$ for a small dataset like MNIST is overkill.

Our FPGA design possesses the limitation that each node can handle a maximum of $256K$ samples. Hence, for the benchmarks Webspam $(n = 350K)$ and Covtype $(n = 464K)$, we leave the FPGA entry for $p = 1$ blank, and report training times for $p = 2$ onwards. A similar issue arises while running $n = 2M$ SUSY samples for $p = 8$ on the SoC server. Spawning two MPI processes in a single SoC, with $n = 250K$ samples per process leads to a memory overrun. However, going by the FPGA speedup trend relative to SoC, one can safely argue that the training time for $p = 8$ would be ~24x longer than that of FPGA.

5.2.2 FPGA speedup relative to SoC and CPU

Let us denote training time for a given benchmark on p cores of CPU cluster, SoC server and FPGA cluster as T_p^{cpu}, T_p^{soc} and T_p^{fpga} respectively. For a given number of cores p, we compute the accelerator speedup with respect to the SoC cluster as

$$S_p^{soc} = \frac{T_p^{soc}}{T_p^{fpga}}$$

Similarly, $S_p^{cpu} = \frac{T_p^{cpu}}{T_p^{fpga}}$ denotes the speedup relative to the CPU cluster. Table 5.3 tabulates these speedup values for all benchmarks. We observe that the FPGA-based accelerator outperforms both SoC server and CPU cluster, for both sequential ($p = 1$) and distributed ($p = \{2, 4, 8\}$) implementation. The proposed design achieves a maximum sequential speedup of 25x over the SoC cluster, with a maximum of 1.72x relative to the CPU cluster. For the distributed implementation, the proposed accelerator achieves a maximum speedup of 24x and 1.81x relative to the SoC and CPU clusters respectively. These results establish the FPGA-based accelerator to be the most efficient among the competing platforms.

5.2.3 Parallel Scalability

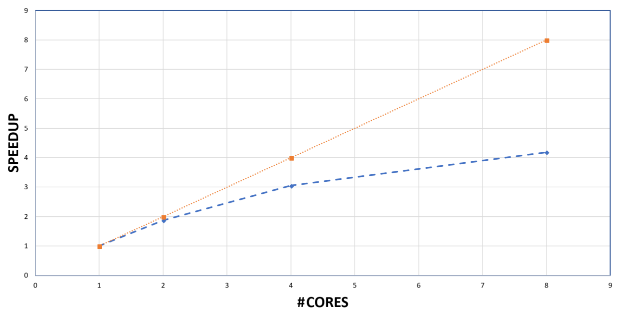
To determine the scalability of the proposed design, we measure the algorithm's performance upon doubling the number of cores. We demonstrate two flavors of scalability: *strong* scaling and *weak* scaling.

- To evaluate the **strong** scaling property, we measure the training times as we double the number of participating nodes to solve the same problem. For a perfectly parallelizable algorithm, the training time should halve as we double the number of nodes.

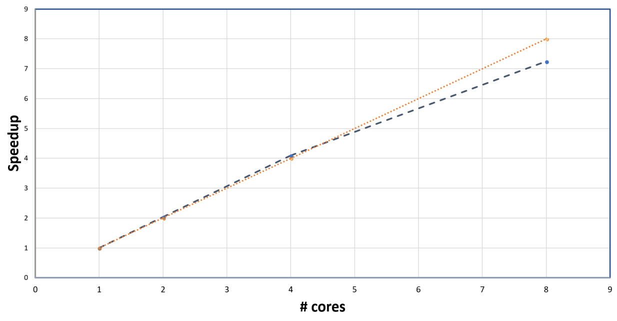
- The **weak** scaling property is evaluated by doubling the participating nodes, while keeping the workload per node constant. The SUSY dataset is used exclusively to evaluate this property. We fix the per-node samples to be $n = 250K$, and evaluate the *per-iteration* time as we double the number of nodes.

It can be observed in Figure 5.1 that for relatively larger datasets, namely *Skin, Webspam* and *covType*, the speedup for the proposed accelerator (blue line) is close to the ideal speedup (orange line). In other words, as we double the number of cores, the training becomes faster by a factor of nearly two. This trend can be attributed to the communication-efficient design of QRSVM [14]. As the communication overhead is significantly lower than the computational load, the training time decreases by a factor equal to the number of nodes.

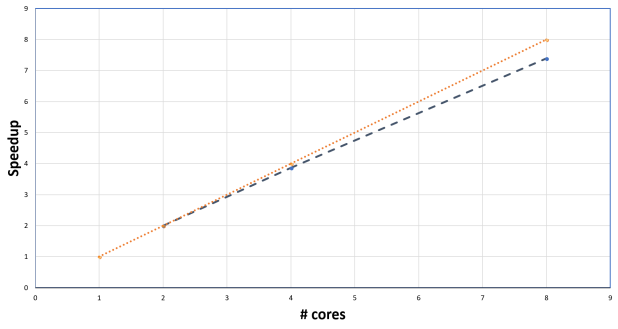
The trend line for *MNIST* quickly falls below ideal efficiency. As discussed before, this can be attributed to reduced computation per node, as we go beyond $p = 2$ nodes. It should also be noted that baseline implementation for Webspam and covType benchmarks on FPGA accelerator



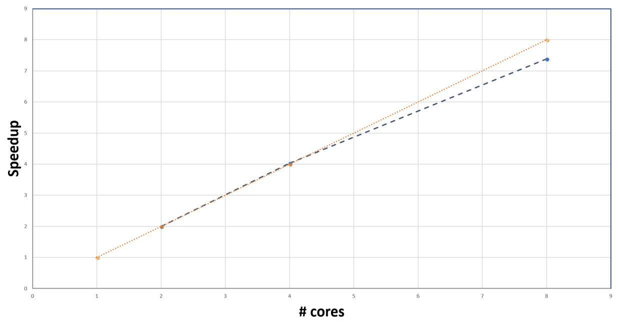
(a) MNIST



(b) Skin



(c) Webspam



(d) covType

Figure 5.1: Strong Scaling Analysis: FPGA accelerator scales linearly with increasing nodes

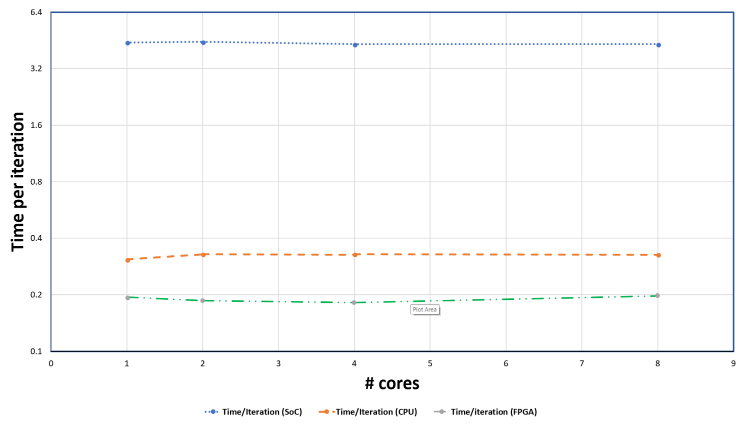


Figure 5.2: Weak scaling analysis: FPGA accelerator achieves least iteration time

is taken as $p = 2$. In the absence of a sequential ($p = 1$) implementation, it is a fair baseline, given that we see a speedup of 2 at $p = 2$ for the other datasets.

To evaluate the weak scaling efficiency of the algorithm, we turn to Figure 5.2, which presents the *per-iteration* time for all three platforms for an increasing number of nodes. It can be observed that for any given p, the FPGA-based accelerator has the lowest per-iteration time amongst all three platforms. It was discussed earlier that parallel dual ascent accounts for a lion's share of the training time. Therefore, by achieving the lowest per-iteration time, the FPGA accelerator proves to be the platform of choice for training QRSVM.

5.2.4 Energy Efficiency

To determine the viability of the proposed FPGA accelerator as an edge-computing solution, we evaluate the energy consumption of the three platforms for a given training task.

- We calculate the approximate energy consumption by multiplying the QRSVM training time with the power rating of the device.
- Additionally, we measure how energy consumption of the FPGA accelerator changes, as we increase the number of computing nodes.

Table 5.4 shows the power rating for each platform. For the FPGA platform, we ignore the host

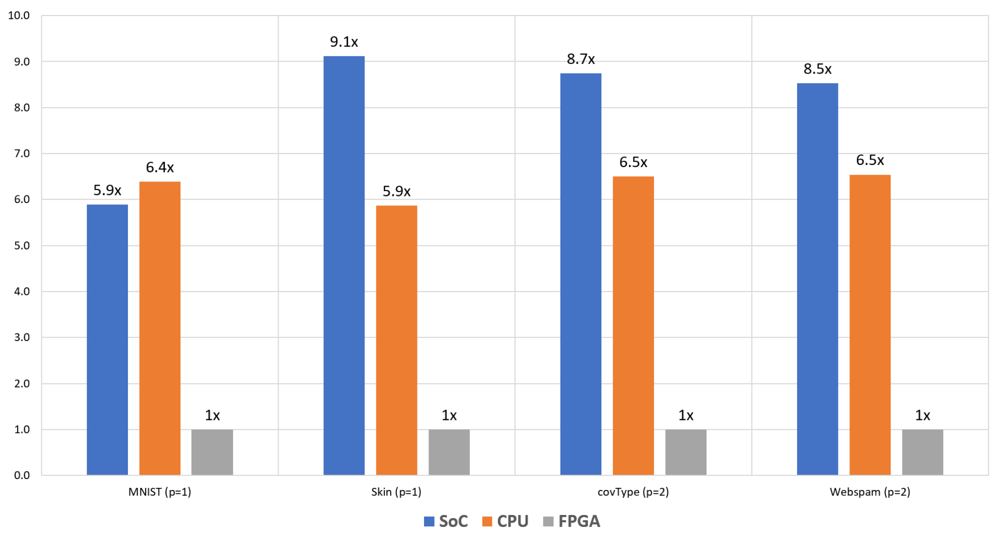
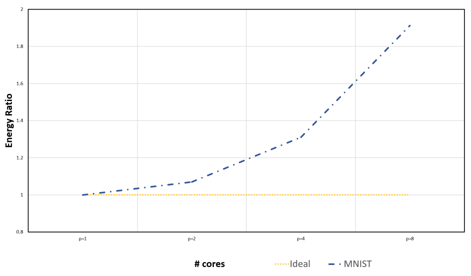


Figure 5.3: Energy comparison between FPGA, CPU and SoC platform

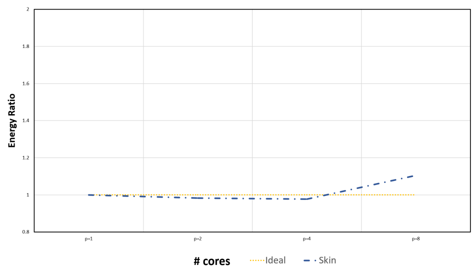
power consumption and only consider the power consumed by the FPGA, since the algorithm's computations are carried out on the FPGA. For the CPU cluster, we take the Thermal Design Power (TDP) [23] of the Intel Xeon CPU as the power consumed by a single core. By all angles, this value is a pessimistic estimate of the actual power consumed. However, studies based on Intel architectures [24] have shown that "uncore" power, i.e. power consumed by units peripheral to the cores, accounts for $\sim 75\%$ of the total chip power. Therefore, the energy estimations are accurate to a fair degree. In a similar vein, The SoC rating is the power consumed by a single KeyStone II SoC [25].

Figure 5.3 shows the normalized energy consumption of the CPU and SoC platforms, relative to the FPGA accelerator. For datasets *MNIST* and *Skin*, we compute the energy consumption for the $p = 1$ case. For *Webspam* and *covType*, we plot the consumption for $p = 2$. From the figure, we can gauge that the FPGA accelerator achieves a $6x - 8x$ reduction in energy consumption over the CPU and SoC platforms. This reduction is achieved through a combination of decreased training times and lower power consumption of the FPGA architecture.

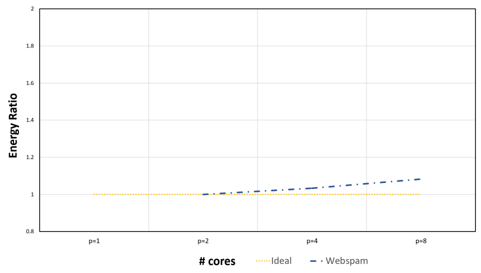
Figure 5.4 illustrates the increase in FPGA energy consumption with an increasing number of



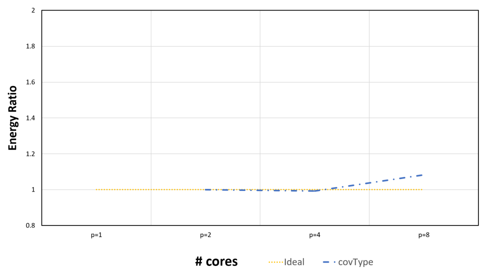
(a) MNIST



(b) Skin



(c) Webspam



(d) covType

Figure 5.4: Energy Consumption for the FPGA accelerator

cores. We observe that for *Skin, Webspam* and *covType*, the energy consumption increases by less than 10% as we add more processing nodes. For the *MNIST* dataset, we see an increasing trend for the energy consumption. This can be attributed to the poor scaling efficiency for MNIST (ref. Figure 5.1a).

In conclusion, the FPGA based accelerator is the most energy efficient platform for QRSVM training, consuming $6x - 8x$ lesser energy than its CPU and SoC counterparts. Moreover, the FPGA energy consumption is more or less constant for a given task, and is independent of the number of computing nodes.

Table 5.3: QRSVM Training time (s). Speedup of FPGA with respect to SoC and CPU are denoted as S_p^{soc} and S_p^{cpu}

#cores	**#iterations**	**MNIST** ($C = 1$, $\gamma = 2^{-6}$, $\eta^* = 0.9$)				
p	t	*CPU*	*SoC*	*FPGA*	S_p^{soc}	S_p^{cpu}
1	181	18.78	179.12	10.92	16x	1.72x
2	181	8.25	61.27	5.84	10x	1.41x
4	182	4.35	31.06	3.58	9x	1.27x
8	182	2.55	20.12	2.61	8x	0.97x

#cores	**#iterations**	**Skin** ($C = 1$, $\gamma = 2^{-8}$, $\eta^* = 0.9$)				
p	t	*CPU*	*SoC*	*FPGA*	S_p^{soc}	S_p^{cpu}
1	67441	7167	115189	4536	25x	1.58x
2	64424	3093	48335	2228	22x	1.39x
4	59761	1607	21773	1108	20x	1.45x
8	54744	759	6121	626	10x	1.21x

#cores	**#iterations**	**Webspam** ($C = 1$, $\gamma = 1$, $\eta^* = 0.9$)				
p	t	*CPU*	*SoC*	*FPGA*	S_p^{soc}	S_p^{cpu}
1	559	236.54	2848	-	-	-
2	566	133.99	1809	76.14	24x	1.76x
4	564	65.92	895	39.36	23x	1.67x
8	569	34.58	477	20.59	23x	1.68x

#cores	**#iterations**	**Covtype** ($C = 1$, $\gamma = 2^{3}$, $\eta^* = 0.9$)				
p	t	*CPU*	*SoC*	*FPGA*	S_p^{soc}	S_p^{cpu}
1	1132	292.63	3192	-	-	-
2	1125	160.08	2229	91.45	24x	1.75x
4	1080	77.19	1079	45.36	24x	1.70x
8	1068	37.86	520	24.75	21x	1.53x

#cores	**#samples**	**SUSY** ($C = 1$, $\gamma = 2^{-3}$, $\eta^* = 0.9$)				
p	n	*CPU*	*SoC*	*FPGA*	S_p^{soc}	S_p^{cpu}
1	250K	171.29	2452	108.08	23x	1.58x
2	500K	232.01	3131	131.02	24x	1.77x
4	1M	319.03	4189	176.18	24x	1.81x
8	2M	497.45	-	299.63	-	1.66x

Table 5.4: Power rating for different platforms

Platform	**Rating (W)**	**Operating Frequency**	**Source**
FPGA	39	125 MHz	Synthesis
CPU	145	3 GHz	TDP
SoC	14	1 GHz	Datasheet

6. CONCLUSION

In this research, we propose an FPGA-based accelerator for a distributed SVM algorithm. The accelerator was assembled by designing vectorized, pipelined hardware modules for the underlying SVM computations. The accelerator was implemented on a cloud-based, multi-FPGA platform provided by Amazon Web Services. We evaluate the accelerator by comparing against two other computing platforms, an Intel Xeon-based high performance computing cluster and a commercial ARM A15-based SoC server. On a per-node basis, the accelerator delivers up to 1.81x and 24x faster training than the CPU and SoC platform respectively. In addition, the design demonstrates a high degree of scalability, adapting to both growing data sizes and increasing compute nodes in a distributed framework. Alongside, the FPGA-based accelerator consumes up to 6.4x and 8.4x less energy, in comparison to the CPU cluster and the SoC server. In light of the above result, the FPGA accelerator is a high performance, energy efficient alternative for applications that involve training and analytics at the edge.

REFERENCES

[1] M. Papadonikolakis and C. S. Bouganis, “Novel cascade fpga accelerator for support vector machines classification,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, pp. 1040–1052, July 2012.

[2] M. Qasaimeh, A. Sagahyroon, and T. Shanableh, “Fpga-based parallel hardware architecture for real-time image classification,” *IEEE Transactions on Computational Imaging*, vol. 1, pp. 56–70, March 2015.

[3] X. Song, H. Wang, and L. Wang, “Fpga implementation of a support vector machine based classification system and its potential application in smart grid,” in *2014 11th International Conference on Information Technology: New Generations*, pp. 397–402, April 2014.

[4] S. Kim, S. Lee, K. Min, and K. Cho, “Design of support vector machine circuit for real-time classification,” in *2011 International Symposium on Integrated Circuits*, pp. 384–387, Dec 2011.

[5] M. Ayinala and K. K. Parhi, “Low-energy architectures for support vector machine computation,” in *2013 Asilomar Conference on Signals, Systems and Computers*, pp. 2167–2171, Nov 2013.

[6] M. Ruiz-Llata, G. Guarnizo, and M. YÃľbenes-Calvino, “Fpga implementation of a support vector machine for classification and regression,” in *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–5, July 2010.

[7] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, “A reconfigurable fabric for accelerating large-scale datacenter services,” in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA ’14, (Piscataway, NJ, USA), pp. 13–24, IEEE Press, 2014.

[8] "Amazon EC2 F1 Instances." `https://aws.amazon.com/ec2/instance-types/f1/`.

[9] M. Papadonikolakis and C. S. Bouganis, "A scalable fpga architecture for non-linear svm training," in *2008 International Conference on Field-Programmable Technology*, pp. 337–340, Dec 2008.

[10] M. B. Rabieah and C. S. Bouganis, "Fpga based nonlinear support vector machine training using an ensemble learning," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, Sept 2015.

[11] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. P. Graf, "A massively parallel fpga-based coprocessor for support vector machines," in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pp. 115–122, April 2009.

[12] P. B. A. Phear, R. K. Rajkumar, and D. Isa, "Efficient non-iterative fixed-period svm training architecture for fpgas," in *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*, pp. 2408–2413, Nov 2013.

[13] D. Anguita, A. Boni, and S. Ridella, "A digital architecture for support vector machines: theory, algorithm, and fpga implementation," *IEEE Transactions on Neural Networks*, vol. 14, pp. 993–1009, Sept 2003.

[14] J. Dass, V. N. S. P. Sakuru, V. Sarin, and R. N. Mahapatra, "Distributed QR Decomposition Framework for Training Support Vector Machines," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 753–763, June 2017.

[15] S. Si, C.-J. Hsieh, and I. Dhillon, "Memory efficient kernel approximation," in *Proceedings of the 31st International Conference on Machine Learning* (E. P. Xing and T. Jebara, eds.), vol. 32 of *Proceedings of Machine Learning Research*, (Bejing, China), pp. 701–709, PMLR, 22–24 Jun 2014.

[16] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd Ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[17] "Vivado High-Level Synthesis." `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

[18] "Vivado Design Suite User Guide: High Level Synthesis." `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf`.

[19] "Improving Performance Objectives." `http://users.ece.utexas.edu/~gerstl/ee382v_f14/soc/vivado_hls/VivadoHLS_Improving_Performance.pdf`.

[20] "AXI Reference Guide." `https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf`.

[21] "AWS EC2 FPGA Software Development Kit." `https://github.com/aws/aws-fpga/tree/master/sdk`.

[22] C. Sanderson, "Armadillo c++ linear algebra library," June 2016.

[23] "TDP of Intel Xeon E5-2686 v4." `https://en.wikichip.org/wiki/intel/xeon_e5/e5-2686_v4`.

[24] B. Goel and S. A. McKee, "A methodology for modeling dynamic and static power consumption for multicore processors," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 273–282, May 2016.

[25] "66AK2Hx KeyStone SoC." `http://www.multivu.com/assets/54044/documents/54044-66AK2H-Product-Bulletin-original.pdf`.