CRRS: Custom Regression and Regularisation Solver for Large-scale Linear Systems

Andreea-Ingrid Cross*, Liucheng Guo*, Wayne Luk*, and Mark Salmon†
*Imperial College London, UK
Email: andreea.funie09@imperial.ac.uk,liucheng@tg0.co.uk,w.luk@imperial.ac.uk
†Cambridge University, UK
Email: mhs39@cam.ac.uk

***Abstract*—This paper presents novel regression and regularisation techniques based on Field Programmable Gate Array (FPGA) technology for large-scale datasets for machine learning and other applications. We introduce a customisable design which allows end-users to select their regression and regularisation techniques from a library supporting relevant methods such as Multiple Linear Regression, Ridge Regression, Adaptive/Lasso Regression and Elastic Net Regularisation. We introduce the first Adaptive Elastic Net architecture for FPGAs. Tests on dense and sparse datasets of varying sizes show 158 times speedup and 114 times enhancement in energy efficiency, when comparing an 8-FPGA system with the corresponding software C++ implementation on a 12-core CPU, for an Adaptive Elastic Net regularisation of a matrix with $11.56 * 10^9$ coefficients.**

I. Introduction

Regression analysis is widely used for forecasting, variable selection and regularisation across many scientific and engineering fields, such as machine learning, deep learning [1] and finance [2].These methods, however, are computationally demanding and the performance of CPUs becomes unsatisfactory. Accelerators based on FPGAs and GPUs are being adopted to provide fast, energy-efficient and scalable solutions. While GPUs are good at computing floating-point operations, they often struggle to handle compact data types and irregular parallelism. In contrast, FPGA resources are customisable, enabling irregular parallelism and custom data types. This paper introduces what we believe to be the first pipelined design for the Adaptive Elastic Net algorithm. We generalize this design to a customisable FPGA-based regression and regularisation solver which targets large datasets, obtaining great speedup while preserving accuracy. One issue technologies like GPUs and FPGAs have is their limited on-chip memory. Our study addresses this issue by scaling to multiple FPGAs with an appropriate partitioning of the data among them. The main contributions of our research are as follows.

- The first pipelined Adaptive Elastic Net architecture based on a column-wise Jacobi step operation to enable a fully pipelined solution.
- A novel FPGA-based solver, CRRS, which allows the selection of regression methods such as Adaptive Elastic Net, Elastic Net, Ridge, Lasso, Adaptive Lasso or Multiple Linear Regression, providing horizontal scalability and the ability to solve large datasets ($\approx 94 * 10^9$ matrix coefficient elements).
- Experimental results showing significant speedup when CRRS is compared with its multi-core C++ CPU based software counterpart, as well as with well-known libraries such as *glmnet*.

II. Background

Multiple Linear Regression. In many real-world scenarios the relationship between multiple features (X) and a response (Y) is modelled by fitting a line as in Equation 1 to the observed data, using *multiple linear regression* (MLR).

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \hat{\beta}_2 x_{i2} + ... + \hat{\beta}_p x_{ip} + \hat{\epsilon}_i, \text{ i = 1..n} \quad (1)$$

where $\hat{\beta}$ are regression coefficients, and $\hat{\epsilon}_i$ is the residual error. The *Least Squares* method [3] aims to identify $\hat{\beta}$ with a minimum residual error according to $\hat{\beta}^{OLS} = \arg\min_\beta \sum_{n=1}^{N}(y_n - \beta x_n)^2$. The linear algebraic form of MLR is thus $\hat{Y} = X\hat{\beta} + \hat{\epsilon}$ where $\hat{\beta} = \hat{\beta}_0, \hat{\beta}_1 \cdots \hat{\beta}_p, \hat{\epsilon} = \hat{\epsilon}_0, \hat{\epsilon}_1 \cdots \hat{\epsilon}_p$, and X is the feature matrix. This can be further expressed as $\hat{y} = X\hat{\beta}$ ($\hat{\epsilon} = 0$ in an ideal scenario), leading to solving a linear system through the equation $\hat{\beta}^{OLS} = (X^T X)^{-1} X^T y$, equivalent to solving $Ax = b$ where $A = X^T X$ and $b = X^T y$. We can find a solution for $\hat{\beta}^{OLS}$ through a number of methods such as Jacobi, Gauss-Seidel, Conjugate Gradient, Steepest Descent etc. The Jacobi method is preferred due to its potential for parallelism using the coefficient matrix parallel method [4], without accumulating rounding errors. The Jacobi iteration point form is defined below [5].

$$x_i^{k+1} \Leftarrow \frac{1}{a_{ii}}[b_i - \sum_{j=1;j\neq i}^{j=n} a_{ij} * x_j^k] \quad (2)$$

To improve the classic Jacobi method convergence rate, we introduce and use as part of our CRRS design the *weighted Jacobi* algorithm [10], which has the following point form:

$$x_i^{k+1} \Leftarrow (1-\omega) * x_i^k + \omega * \frac{1}{a_{ii}}[b_i - \sum_{j=1;j\neq i}^{j=n} a_{ij} * x_j^k] \quad (3)$$

where $\omega \in R$ is the weight parameter.

Ridge Regression. If the MLR corresponding linear system leads to an overfitted/underfitted system of equations [6], to be able to pick a solution we include a further constraint [7], namely the L1 norm penalty.

Lasso Regression. As with Ridge regression, the Lasso is a form of *feature selection* which trades an increase in bias with a decrease in variance by introducing the L2 norm penalty. Lasso can be computed through a modification of the *Least Angle Regression* (LARS) algorithm [3] and it provides sparse models which are easier to interpret.

Elastic Net Regularized Regression. The Elastic Net is a regularisation and variable selection method which linearly combines the Ridge and Lasso methods. It is particularly useful when the number of predictors (p) is much larger than the number of observations (n)[6] and is defined as below.

$$\hat{\beta}^{EN} = \arg\min_{\beta} \frac{1}{2}||y - X\beta||_2^2 + \lambda_1 \sum_{i=1}^{p} |\beta_i| + \lambda_2 \sum_{i=1}^{p} \beta_i^2 \quad (4)$$

The Elastic Net reduces to Ridge regression when $\lambda_1 = 0, \lambda_2 = 1$, to Lasso regression when $\lambda_1 = 1, \lambda_2 = 0$ and to MLR when $\lambda_1 = 0, \lambda_2 = 0$. This estimator inherits drawbacks such as the lack of oracle property. To overcome this, an *Adaptive Elastic Net estimator (AENET)* was introduced in [8]:

$$\hat{\beta}^{AEN} = \arg\min_{\beta} \frac{1}{2}||y - X\beta||_2^2 + \lambda_1 \sum_{i=1}^{p} \hat{w}_i |\beta_i| + \lambda_2 \sum_{i=1}^{p} \beta_i^2 \quad (5)$$

where $\hat{w}_{i=1}^{p}$ are the adaptive data-driven weights. For $\lambda_2 = 0$ we obtain the Adaptive Lasso. The *Generalised Elastic Net* [9] adopts the L_0 norm to recover sparse signals with high probability and is defined as:

$$\hat{\beta}^{GEN} = \arg\min_{\beta} \frac{1}{2}||y - X\beta||_2^2 + \lambda_1 ||\beta||_{0,\delta} + \lambda_2 ||\beta||_2^2, \quad (6)$$

where $||\beta||_{0,\delta} = \sum_{i=1}^{N} \frac{\beta_i^2}{\beta_i^2 + \delta}$ and $\delta > 0$ is a parameter that approaches zero to approximate $||\beta||_0$. This problem can be solved using the iterative IAGENR-L0 algorithm which reduces to solving the following linear system for x^{k+1}:

$$(A^T A + Diag[\frac{2\lambda_1}{\theta + ((x_i^k)^2)^2} + 2\lambda_2, i = 1, N])x^{k+1} = A^T y \quad (7)$$

This algorithm is adopted for the Adaptive Elastic Net regularisation in order to build our hybrid CPU-FPGA design, CRRS.

III. CRRS Design

In this section we exploit the internal parallelism achievable with FPGA technology (in our case, a MAIA dataflow engine (DFE) containing an FPGA and DRAM). We start by describing a reconfigurable design which achieves the throughput rate of one data point per clock cycle. We explain how we enhance our design to take advantage of larger FPGAs, where multiple parallel processing pipelines can be deployed concurrently. We show how we can solve MLR, Lasso, Adaptive Lasso, Ridge, Elastic Net or Adaptive Elastic Net regularisation, as well as how we can scale the solution to the number of available FPGAs. To solve any regression/regularisation problem we need to solve the underlying linear system. For this we use the Jacobi algorithm with a column-wise weighted Jacobi step operation for maximum parallelism on the FPGA.

We make a number of design and implementation decisions. 1) We assume the matrix correctness before sending it to the FPGA; 2) We perform a column-wise summation and transpose the matrix on the CPU before sending it; 3) For the weighted Jacobi, we use the popular value of $\omega = \frac{2}{3}$ for the weight parameter [10]; 4) The initial solution x_0 is the null vector; 5) The w weight vector contains only the 1 value initially, unless stated otherwise; 6) All elements used for computation are *single precision floating point (SFP)* values; 7) The default maximum number of iterations is set to $10,000$; 8) The default values for other scalar inputs are: $\lambda_1 = 1 * 10^{-3}, \lambda_2 = 1 * 10^{-5}, \theta = 1 * 10^{-6}$ as per [9], *isAdaptive* = 0 , γ = 2, eps = 1.0e-4.

CRRS allows the following parameters to be customised: number of FPGAs, parallelisation degree on each FPGA, number of iterations, weight, convergence rate threshold and penalty specific parameters. CRRS is based on the generalized Elastic Net following the IAGENR-L0 method, but optimized to be able to solve an AENET regularisation. We call this novel pipeline-friendly algorithm GAENET.

Algorithm 1 GAENET

```
1:  Pick an initial guess value $x_0$
2:  $iteration \leftarrow 0$
3:  Check for convergence
4:  while convergence not reached do:
5:  for p := 1 until nPipes do:
6:  for i := 1 until R/nPipes do:
7:  $residual \leftarrow 0$
8:  for j := 1 until C do:
9:  $residual \leftarrow residual + A[j] * x[j]$
10: end for
11: $interim \leftarrow \theta + x[i] * x[i]$
12: $L1 \leftarrow 2 * \lambda_1 * \theta$
13: if $isAdaptive == 1$ then:
14:     $L_1^{'} \leftarrow L1 * w[i]$
15: else:
16:     $L_1^{'} \leftarrow L1$
17: $newD \leftarrow Diag[i] + L1^{'}/(interim * interim) + 2 * \lambda_2$
18: $xNew \leftarrow x[i] + \omega * [(B[i] - residual)/newD - x[i]]$
19: $DidConverge(xNew, x[i])$
20: Write $xNew$ to memory, by overwriting $x_i$
21: end for
22: end for
23: Check for convergence
24: $iteration \leftarrow iteration + 1$
25: end while
```

In Algorithm 1, $nPipes$ represents the number of pipes/threads. R is the number of rows and C is the number of columns of the coefficient matrix. x is the solution vector. b is the constant-term column-matrix element, and $Diag$ stores the main diagonal elements of the coefficient matrix which is represented by the array A. ω is the weighted Jacobi parameter, $newD$ represents the new diagonal elements after computation as in Algorithm 1, line 17. λ_1, λ_2, and θ are the penalty specific parameters. The $isAdaptive$ parameter allows the user to choose between the classic and adaptive Lasso/Elastic Net.

Single Iteration CRRS Design. To aid the FPGA implementation of GAENET, we introduce a new vector, $Diag$ which consists of each element from the main diagonal of the coefficient matrix. We then replace each of the main diagonal elements from the original matrix with 0, such that they do not add any value when included into the partial adder computation performed on the FPGA. The single iteration CRRS design can be summarized as follows: 1) Load coefficient matrix A as an array, together with the constant-terms column-matrix b, the initial solution x_0, the initial weight vector w and the main-diagonal elements vector $Diag$, to accelerator DRAM; 2) Load all scalar values ($\omega, \epsilon, \lambda_1, \lambda_2, \theta, isAdaptive$); 3) Compute GAENET and write the new x_1 vector solution to accelerator DRAM by overlapping the old x_0; 4) Read the new X_1 vector solution from accelerator DRAM and output results to CPU; 5) If the classic Elastic Net version is chosen, stop, otherwise, compute the new weights on the CPU according to Equation 9; 6) Repeat Steps 3 - 4. Our design consists of a binary reduction tree, several subtraction units, dividers, adders and multipliers and some control units, organized in processing elements. We also use a multiplexer ($adenet$) to choose between computing the adaptive or the classic regularisation/regression method, according to the value of the scalar input $isAdaptive$ – 1 = adaptive, 0 = classic. We ingest from DRAM one column from the coefficient matrix A, together with the vector value x_i. The elements we add at iteration $k+1$ are of the form $a_{ij}x_j^k$ and the binary tree reduces the inputs to produce the sum $\sum_{j=1;j\neq i}^{j=n} a_{ij} * x_j^k$, named $residual$. We create a first processing element (PE) called $interim$ by feeding the value X_i together with its duplicate value, into a multiplier whose result is then fed into an adder to produce Algorithm 1, line 11. We then have a series of elements. L1 is a *scalar* which has the value $2*\lambda_1*\omega$ precomputed on the CPU - as displayed in Algorithm 1, line 12. According to the multiplexer value $isAdaptive$ we then feed the result of L1 into another multiplier together with the weight value W_i to get L1', or we pick the result of L1, and this represents the second PE named L1'. We feed $interim$ and its duplicate into a multiplier which represents the third PE, named $duplicateInterim$ which then gets fed together with L1' into a division unit, to form the fourth PE, called $newInterim$. The value of $2*\lambda_2$, as displayed in Algorithm 1, line 17 is precomputed on the CPU and brought on the FPGA as a scalar named L2. A fifth PE is then formed by feeding L2 into an adder together with the $newInterim$ PE. This then gets fed into another adder together with the diagonal value $Diag_i$, to form the sixth PE, named $newD$. We feed the accumulated sum $residual$ and the B_i value into a subtraction unit to produce a seventh PE value equal to $b_i - \sum_{j=1;j\neq i}^{j=n} a_{ij} * x_j^k$, named $inter$. This value and the $newD$ value will enter a final division unit to produce the eight PE named $interResult$, who will then enter a final subtraction unit together with the x_i value, to produce the ninth PE named $newElement$. Further we have the tenth PE named $weightedElement$ computed from a multiplier that uses $newElement$ and the weight constant ω. The last PE, $xNew$, is produced using an adder with the $newElement$ and the x_i value, at a rate of one value per clock cycle.

We further improve our design by implementing several parallel processing pipelines (*pipes*), allowing for multiple blocks of rows — up to p pipes to be evaluated in parallel.

Multiple Iterations CRRS Design. We are able to compute multiple iterations of CRRS using a hybrid CPU-FPGA approach. To allow convergence checking we implement an extra PE, *DidConverge*, which, at the end of the iteration, outputs its result back to the CPU as an integer value, 0 – convergence not reached or 1 – convergence reached. The *DidConverge* PE feeds in the new x_j^{k+1} value and the old x_j^k value to a subtraction unit whose result is then compared to a preset threshold ϵ and if smaller, convergence is reached. When using multiple iterations, our single iteration design is modified as follows. In the 3rd step of the single iteration solver, we also check for convergence on the FPGA and output its result back to the CPU. For the 4th step, we only read the current x_{k+1} vector solution from accelerator DRAM and output its value to the CPU if convergence is reached. We enter the 5th step if not converged and an adaptive method version is chosen and executed by repeating steps 3-4.

CRRS Design on Multiple FPGAs. Each accelerator has a fixed amount of DRAM available. Our solver can scale with the number of FPGAs available. On the CPU we check if we can split the data equally between FPGAs, i.e. for a matrix of $n \times n$, with m available FPGAs, the matrix is split in m matrices with $\frac{n}{m}$ rows and n columns each. If an equal split is not possible, we attempt to fill the DRAM of as many FPGAs as possible with blocks of rows, with the remaining data being allocated to a final FPGA. If more than 8 FPGAs are needed, communication between 1U units would need to be managed to avoid bottlenecks, but we leave this as further work.

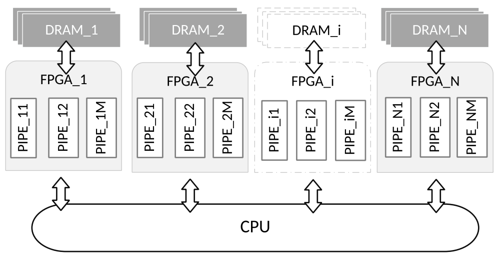

Fig. 1: CRRS on multiple FPGAs

IV. CRRS Implementation

FPGA Implementation. We write the values of the x solution vector on DRAM as they are computed. If we are not following an adaptive method, we read the x value from DRAM only when converged, otherwise we read it at the end of each iteration, compute new weights w according to Equation 9 and rewrite to FPGA DRAM. Each of our FPGAs has 48GB of DRAM which is equivalent to $\approx 12 * 10^9$ single precision floating-point (4 bytes) values. We also store the new x solution vector values, the b constant-term column-matrix elements, the main diagonal elements $Diag$, the corresponding array of coefficients for matrix A and the w adaptive method weights values. With a total of 8 FPGAs, we are able to increase matrix size further, by splitting the input matrix between multiple FPGAs.

Resource Usage. Table I shows the kernels, managers and IO resource usage, as a percentage of the total available resources.

TABLE I: FPGA total resource usage (expressed in percent) for single precision floating-point arithmetic implementation

Pipes	**1**	**24**
Total Logic	20.92%	59.45%
LUTs / FFs	12.16% / 8.72%	39.13% / 28.41%
BRAM / DSP	29.45% / 0.61%	74.09% / 25.67%

CPU Implementation of GAENET. The CPU implementation is built using C++11, parallelised using OpenMP and compiled using g++ 4.9.2 with flags *-O3 -march=native -fopenmp* to enable general performance, architectural and multi-threaded optimization for Intel XEON, being parallelised in a similar manner to the hardware implementation, by dividing the original matrix into sub-matrices. Table II shows the scalability of our CPU implementation for a dense matrix of $107,520 \times 107,520$ elements by presenting the average time taken for the CPU to perform one AENET iteration for GAENET, after 10 independent runs. We disable Hyper-Threading and only use 6 threads per CPU (in total 12 threads) to prevent the CPU implementation from scaling sub-linearly.

TABLE II: CPU scalability for up to 12 threads and 1 AENET iteration - $107,520 \times 107,520$ matrix

# Threads	1	2	4	8	10	11	12
CPU Time (s)	641.16	337.45	164.4	82.2	66.79	61.06	56.74
Speedup	1	1.9	3.9	7.8	9.6	10.5	11.3

V. Evaluation

The accelerator we use is a Maxeler MPCX node containing a Maia DFE with a Stratix V FPGA and 48 GB of onboard DRAM. We evaluate software results on the CPU node attached to the MPCX node through a Mellanox FDR Infiniband switch, namely a Dual Intel Xeon E5-2640 with 6 cores (testing on 12 threads) and 64 GB of onboard DRAM. Our FPGA implementation runs at a clock frequency of $200MHz$ for the 1 pipe solution, at $190Mhz$ for the 8 and 16 pipe solutions and at $180MHz$ for the 24 pipe solution. All run times are measured using the C++11 standard library time function: *chrono::high_resolution_clock::now()*. The Stratix V FPGA node adopts 28nm technology, while the Dual Intel Xeon E5-2640 adopts 22nm.

We provide results with the initial CPU-DRAM transfer time plus execution time, as well as just with the execution times. FPGA performance is evaluated against a 12-thread C++ counterpart as well as the R libraries *glmnet* and *msaenet*. The *glmnet* R library can use multi-core architectures only for cross-validation, hence we only show single threaded performance. Despite its efficiency, *glmnet* does not support Adaptive Elastic Net/Lasso regularisation. To overcome this, we use *msaenet* - an efficient R library built on top of *glmnet*. We show performance results for the Adaptive Elastic Net Regression because to the best of our knowledge, this is the first implementation of its kind on an FPGA. We test both dense and sparse randomly generated matrices with 10% of their total predictors associated as real predictors. All sparse matrices generated have 40% non-zero entries. We measure the execution and total computation times after 10 iterations.

Single FPGA - CRRS Performance Results. Table III shows peak speedups of 20.91 times when measuring only the execution time and 12.07 times when measuring total computation time, for a dense matrix of $107,520 \times 107,520$ elements. We provide timings for one FPGA across a different number of pipes.

TABLE III: Single FPGA performance results

# of pipes	1	8	16	24
# of iterations		1		
CPU Time (s)	56.74	56.74	56.74	56.74
FPGA Exec. Time (s)	61.1	7.68	3.88	2.72
FPGA Speedup	**0.92**	**7.39**	**14.62**	**20.86**
FPGA Total Time (s)	83.1	29.68	25.88	24.72
FPGA Speedup	**0.68**	**1.91**	**2.19**	**2.30**
# of iterations		10		
CPU Time (s)	627.51	627.51	627.51	627.51
FPGA Exec. Time (s)	618.94	80.45	41.31	30.01
FPGA Speedup	**1.01**	**7.80**	**15.19**	**20.91**
FPGA Total Time (s)	640.94	102.45	63.31	52.01
FPGA Speedup	**0.98**	**6.13**	**9.91**	**12.07**

As expected, when CPU-DRAM transfer time is included, we notice a significant decrease in the single iteration CRRS version's speedup. We reduce the impact across multiple iterations for classic methods by only transferring the data to DRAM once at the beginning, rewriting the new solutions to DRAM as they get computed, and reading the final result only once when reaching convergence/maximum number of iterations. For the adaptive methods, a transfer overhead is introduced, as we need to read the current X vector solution, compute the new weights vector W on CPU, and overwrite old W values with new ones on DRAM. We show results for AENET, so the maximum overhead time is included.

Multiple FPGAs - CRRS Performance Results. Table IV shows results for 1 and 10 iterations of the AENET method applied on a $107,520 \times 107,520$ dense matrix. We use multiple FPGAs with 24 pipes each for maximum parallelism. The CPU controls synchronization, so all multi-FPGA performance results include synchronization time. We notice a linear increase in speedup with the number of FPGAs, and peak speedups of 158.31 times when measuring just execution time, or 86.47 times including CPU-DRAM transfer time.

Figure 2 shows how CRRS scales across 8 FPGAs when performing one iteration of the GAENET algorithm. Due to insufficient CPU RAM for a matrix with more than $\approx 12 * 10^9$ single precision floating point elements, we cannot measure real speedup for any larger matrix. We estimate speedup for $215,040 \times 215,040$ and $307,200 \times 307,200$ matrices by extrapolating CPU execution time from other tests performed over varying sizes matrices. These tests measure the CPU implementation of GAENET, *msaenet*, and the 8-FPGA-based CRRS execution times. We display the extrapolation function for predicted speedups for dense and sparse matrices. We also show the R^2 measurement which measures the fit of the

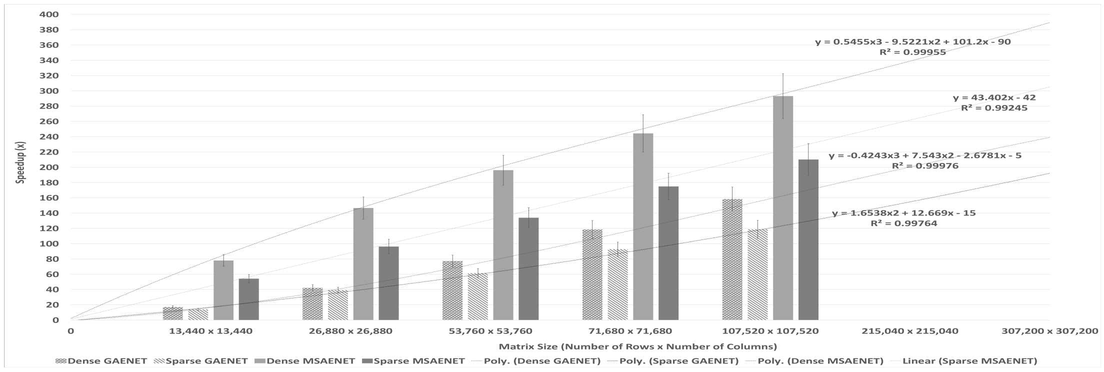

Fig. 2: 12-core GAENET vs 1-core MSAENET vs 8-FPGA CRRS Performance Comparison for Execution Times

TABLE IV: Multiple FPGAs performance results

# of FPGAs	1	2	4	8
# of iterations		1		
CPU Time (s)	56.74	56.74	56.74	56.74
FPGA Exec. Time (s)	2.72	1.38	0.72	0.36
FPGA Speedup	**20.86**	**41.12**	**79.14**	**158.29**
FPGA Total Time (s)	25.91	12.95	7.04	3.53
FPGA Speedup	**2.19**	**4.38**	**8.06**	**16.07**
# of iterations		10		
CPU Time (s)	627.51	627.51	627.51	627.51
FPGA Exec. Time (s)	30.02	15.24	7.93	3.96
FPGA Speedup	**20.91**	**41.18**	**79.18**	**158.31**
FPGA Total Time (s)	54.14	27.04	14.48	7.26
FPGA Speedup	**11.59**	**23.21**	**43.33**	**86.47**

extrapolation function and we notice all functions are over 0.99. We display 10% bars of error from the values to the extrapolation line and show that our results are always within bound, hence drawing the following conclusions within 10% error: 1) For one iteration of AENET targeting the 12-core C++ CPU GAENET for a matrix with $\approx 94.37 * 10^9$ values, an 8-FPGA based CRRS is up to 240 times faster for a dense dataset, and 190 times faster for a sparse one. 2) For one iteration of AENET targeting the highly-efficient *msaenet* for the same matrix, an 8-FPGA based CRRS could be up to 390 times faster for a dense dataset, and 310 times for a sparse one.

When testing our design on the *U.S. Census dataset* ($10,000 \times 45,000$ SFP elements) for an Elastic Net algorithm, the 8-FPGA-based CRRS performs up to 2 times faster than a 8 Tesla P100 using the open-source GPU library $H2O$.

Energy and Power Consumption Results. Table V shows that running our design for one iteration of the AENET method on 8 Stratix V FPGAs, each with 24 pipes, can be up to 114 times more energy efficient than its 12-core CPU counterpart.

TABLE V: Total Energy/Power Consumption for CPU vs FPGA for 1 AENET iteration with a $107,520 \times 107,520$ matrix

Technology	**12-core CPU**	**1 FPGA**	**8 FPGA**
Power (Watt)	90.00	16.40	124.80
Energy (Joules)	5,106.60	44.61	44.93

VI. Conclusion and Future Work

Our study shows the effectiveness of FPGAs for accelerating a number of regression and regularisation methods through our custom regularisation and regression solver, CRRS. We also introduce the first AENET pipelined architecture, implemented on FPGAs. We further show how CRRS is able to provide an efficient scalable solution which allows us to solve large-scale datasets that cannot fit the on-board DRAM of a single FPGA.

We perform tests on dense and sparse datasets and show that our floating-point precision multi-FPGA regression and regularisation solver is able to compute an iteration of the AENET in under a second, with its total computation time including CPU-DRAM transfer time across 10 iterations being less than 10 seconds, when evaluating CRRS on a dataset with $\approx 11.56 * 10^9$ values. When comparing our approach with the well-known *glmnet* R library on both dense and sparse datasets we show that CRRS can achieve a speedup of up to 390 times.

We plan to further develop our design to support mixed-precision, thus enabling users to obtain their desired trade-off between accuracy and speed, and to compare performance against a CPU cluster and cloud-based FPGA acceleration.

VII. Acknowledgments

The support of UK EPSRC (EP/I012036/1, EP/L00058X/1, EP/L016796/1 and EP/N031768/1), the European Union Horizon 2020 Research and Innovation Programme under grant agreement number 671653, the Maxeler University Programme, Altera, Intel and Xilinx is gratefully acknowledged.

References

[1] G. Kang, et al., *Shakeout: A New Regularized Deep Neural Network Training Scheme*, AAAI-16, 2016.

[2] J. Li et al., *Predicting Exchange Rates Out of Sample: Can Economic Fundamentals Beat the Random Walk?*, Journal of Financial Econometrics, 13(2), pp. 293-341, 2015.

[3] X. Yan et. al, *Linear Regression Analysis: Theory and Computing*, World Scientific Publishing Co., 2009.

[4] D.P. O'Leary, R.E. White, *Multi-splittings of matrices and parallel solution of linear systems*, SIAM J. Algebr. Discrete Methods, 6(4), pp. 630-640, 1985.

[5] M. Oieshanskii, et al., *Iterative Methods for Linear Systems: Theory and Applications*, Society for Industrial and Applied Mathematics, 2014.

[6] Hui Zou et al., *Regularisation and variable selection via the Elastic Net*, Journal of the Royal Society: Series B, 67 (2), pp. 301-320, Blackwell Publishing, 2005.

[7] R. A. Maronna, *Robust Ridge Regression for High-Dimensional Data*, Technometrics, 53(1), pp. 44-53, 2011.

[8] H. Zou et al., *On the adaptive elastic-net with a diverging number of parameters*, Ann. Statist., 37(4), pp. 1733-1751, 2009.

[9] S. Li et al., *A generalised Elastic Net regularisation with smoothed l_0 penalty*, Advances in Pure Mathematics, 7, pp. 66-74, 2017.

[10] A. Imakura, et al. *A parameter optimization technique for a weighted Jacobi-type preconditioner*, JSIAM, 4, pp. 41-44, 2012. https://goo.gl/HFhVXu, 2017.