Towards Heterogeneous Solvers for Large-Scale Linear Systems

Stylianos I. Venieris, Grigorios Mingas and Christos-Savvas Bouganis
Department of Electrical and Electronic Engineering
Imperial College London
London, UK
Email: {stylianos.venieris10, g.mingas10, christos-savvas.bouganis}@imperial.ac.uk

***Abstract*—Applying Linear Regression to systems with a massive amount of observations, a scenario which is becoming increasingly common in the era of Big Data, poses major algorithmic and computational challenges. This paper proposes a novel high-performance FPGA-based architecture for large-scale Linear Regression problems as well as a heterogeneous system comprising the custom FPGA architecture, an enhanced GPU module and a multi-core CPU for addressing the aforementioned problem. The system adaptively assigns Linear Regression workloads to the three computing devices to minimise runtime. The device with the highest performance is chosen based on an analytical framework, as well as the workload's size and structure. A quantitative comparison with existing FPGA, GPU and multi-core CPU designs yields speed-ups of up to $\mathbf{18.07}\times$, $\mathbf{32.67}\times$ and $\mathbf{25.84}\times$ respectively.**

I. Introduction

Among the primary mathematical tools that are commonly employed by Big Data analysts, the modelling method of Linear Regression has a prominent role. In various fields of Data Science, from compressive sensing to pattern recognition, linear algebra techniques together with Least Squares methods constitute the most common tool to address Linear Regression problems with massive data. With the scale and scope of scientific computing rapidly increasing, the underlying computing technology that targets such problems is in need for high-performance solutions in order to cope with modern scientific problems that come with large data sets.

This paper focuses on the linear Least Squares method of Linear Regression with emphasis on tackling a wide range of problem sizes. Depending on the application, the size and shape of the matrices that represent the Linear Regression system of equations can vary a lot. As an example, tall-skinny design matrices commonly appear in scientific fields that employ general linear models with many samples and a few tens of parameters, spanning from the Bioinformatics branches of genetics and genomics analysis [1] to the social sciences of Econometrics and Mathematical Psychology [2]. On the other end of the spectrum, in cases where the number of observations is close to the number of parameters, the system matrices approach a square shape. Since the number of samples, and consequently the number of rows, can potentially vary a lot both within and across applications depending on their nature and the availability of data, a system that aims at sustaining a high performance across different matrix sizes has to exploit the heterogeneous set of computing devices that are present in most modern workstations.

Our approach proposes the complementary deployment of heterogeneous computing devices for the solution of linear systems, with sizes spanning from square to tall-skinny, by means of a Least Squares method based on QR decomposition algorithms. The proposed framework comprises three QR algorithms implemented on three computing platforms and is able to automatically select and assign a Least Squares workload to the fastest algorithm-platform combination based on the input matrix dimensions. Our system employs a novel custom reconfigurable architecture mapped onto an FPGA, a highly optimised set of GPU kernels and a multi-core CPU implementation. The QR method is chosen because of its high numerical stability which is a common requirement for several types of applications [3]. Different QR algorithms were selected to be mapped onto the FPGA and the GPU, based on the unique architectural features of each device as discussed in the corresponding state-of-the-art works that target QR in the literature ([4], [5]). The selected algorithms are Tall-Skinny QR (TSQR) and Communication-Avoiding QR (CAQR) respectively [6]. The CPU employs off-the-shelf optimised linear algebra software libraries for the QR factorisations and for the solution of Least Squares systems and also serves as a supervisor for the rest of the computing devices.

The main contributions of this work are the following:

- A novel parametrisable FPGA-based architecture was designed, tailored to the solution of tall-skinny linear systems by means of an enhanced variation of the TSQR algorithm.
- The state-of-the-art GPU work on QR factorisations was enhanced by enabling it to solve Least Squares problems.
- An analytical modelling framework was developed for the performance estimation and optimal configuration of the proposed FPGA-based architecture.
- Finally, a heterogeneous system is proposed for the high-performance solution of linear Least Squares systems across all matrix sizes, from square to tall-skinny, by allocating work to the highest performing platform.

The paper is organised as follows. Section II discusses the algorithmic background on the linear Least Squares problem and the QR factorisation. Section III reviews related work on FPGAs and GPU. Section IV describes the proposed system followed by the analysis of the developed modelling framework in Section V. Finally, Section VI presents a comparison with existing designs and Section VII concludes the paper.

II. Background

A. Linear Least Squares Systems

Given a full-rank, Least Squares system with an input matrix $\boldsymbol{A} \in \mathbb{R}^{(m \times n)}$, a vector $\boldsymbol{b} \in \mathbb{R}^m$ and $m \geq n$, we seek

to find the solution to:

$$\min_{\boldsymbol{x}\in\mathbb{R}^n} \|\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}\|_2^2 \tag{1}$$

A standard method to solve such problems, which is typically preferred due to its numerical stability, is based on the QR factorisation [7]. Using the QR decomposition, matrix $\boldsymbol{A}$ can be decomposed as a product of an $(m\times n)$ orthogonal matrix $\boldsymbol{Q}$ and an $(n\times n)$ upper triangular matrix $\boldsymbol{R}$ where the following equation holds:

$$\boldsymbol{A}=\boldsymbol{Q}\boldsymbol{R} \tag{2}$$

An alternative form of the QR factorisation is the full-QR decomposition where $\boldsymbol{A}=\widetilde{\boldsymbol{Q}}\widetilde{\boldsymbol{R}}$ with $\widetilde{\boldsymbol{Q}}\in\mathbb{R}^{(m\times m)}$ and $\widetilde{\boldsymbol{R}}\in\mathbb{R}^{(m\times n)}$. In this case, the orthogonal, $(m\times m)$ square matrix $\widetilde{\boldsymbol{Q}}$ can be partitioned as $\widetilde{\boldsymbol{Q}}=[\boldsymbol{Q_1}\widetilde{\boldsymbol{Q_2}}]$, where $\boldsymbol{Q_1}$ is $(m\times n)$, and $\widetilde{\boldsymbol{R}}=[\boldsymbol{R}^\top\boldsymbol{0}^\top]^\top$, where $\boldsymbol{R}$ is $(n\times n)$ and upper triangular. Since multiplication by an orthogonal matrix does not change the Euclidean norm of a vector, we multiply (1) by $\widetilde{\boldsymbol{Q}}$:

$$\begin{aligned}\|\widetilde{\boldsymbol{Q}}^\top\boldsymbol{A}\boldsymbol{x}-\widetilde{\boldsymbol{Q}}^\top\boldsymbol{b}\|_2^2 &= \left\|\begin{bmatrix}\boldsymbol{R}\boldsymbol{x}\\ \boldsymbol{0}\end{bmatrix}-\begin{bmatrix}\boldsymbol{Q_1}^\top\boldsymbol{b}\\ \widetilde{\boldsymbol{Q_2}}^\top\boldsymbol{b}\end{bmatrix}\right\|_2^2\\ &= \left\|\boldsymbol{R}\boldsymbol{x}-\boldsymbol{Q_1}^\top\boldsymbol{b}\right\|_2^2+\left\|\widetilde{\boldsymbol{Q_2}}^\top\boldsymbol{b}\right\|_2^2\end{aligned} \tag{3}$$

The second term in this equation is independent of $\boldsymbol{x}$ and therefore is irrelevant to the minimisation. By focusing on the first term, since $\boldsymbol{R}$ is full rank by assumption, the solution of the linear system $\boldsymbol{R}\boldsymbol{x}=\boldsymbol{Q_1}^\top\boldsymbol{b}$ is the solution of the Least Squares problem, i.e. $\boldsymbol{x}=\boldsymbol{R^{-1}}\boldsymbol{Q_1}^\top\boldsymbol{b}$, which can be efficiently solved using back-substitution. Moreover, the value of the objective function, i.e. the norm of the residuals, at the minimum is:

$$\|\widetilde{\boldsymbol{Q_2}}^\top\boldsymbol{b}\|_2=\sqrt{\|\boldsymbol{Q_1}^\top\boldsymbol{b}\|_2^2-\|\boldsymbol{b}\|_2^2}. \tag{4}$$

Finally, multiple Least Squares systems can be solved simultaneously by using a matrix $\boldsymbol{B}$ in place of $\boldsymbol{b}$.

B. QR Factorisation

Several algorithms have been proposed for the computation of QR factorisations of small-scale matrices, with the most well-known being the Gram-Schmidt orthogonalisation, the Cholesky QR, the Givens rotations and the Householder transformations [3]. In this work, the Householder transformations method is used because of its high numerical stability, its use in the TSQR and CAQR algorithms and its common deployment in software libraries [6].

In small-scale matrices with an m-to-n ratio of up to a few tens, Householder QR can be effectively employed. In large-scale problems, a matrix class of primary interest is the tall-skinny class, where $m\gg n$, with an m-to-n ratio spanning from some tens to thousands. Such matrices commonly appear in various applications, such as the computation of likelihoods that arise in the analysis of massive data in the field of Bioinformatics, and require high-performance QR factorisations. Problems of this type can be effectively addressed by the Tall-Skinny QR (TSQR) and Communication-Avoiding QR (CAQR) algorithms. A key element of both methods is the Householder QR algorithm.

C. Householder QR

Using the Householder transformations method, matrix $\boldsymbol{A}$ can be QR-factored with $\boldsymbol{R}=\boldsymbol{Q_n}\boldsymbol{Q_{n-1}}...\boldsymbol{Q_1}\boldsymbol{A}$ and $\boldsymbol{Q}=\boldsymbol{Q_1^{-1}}\boldsymbol{Q_2^{-1}}...\boldsymbol{Q_n^{-1}}$, where $\boldsymbol{Q_k}=\begin{pmatrix}\boldsymbol{I_{k-1}} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{H_k}\end{pmatrix}$. $\boldsymbol{H_k}$ denotes the Householder matrix which is formed as $\boldsymbol{H_k}=\boldsymbol{I_k}-2\frac{\boldsymbol{v_k}\boldsymbol{v_k}^\top}{\|\boldsymbol{v_k}\|^2}$, $\boldsymbol{v}_k\in\mathbb{R}^{(m-k+1)}$ is the Householder reflector vector for the k_{th} column and $\boldsymbol{I_k}$ is the identity matrix. Algorithm 1 outlines the Householder QR steps. At the outer loop's k_{th} iteration, the elements below the diagonal in column k are zeroed and the rest of the columns, that constitute the trailing submatrix of the k_{th} iteration, are updated. In this way, after the completion of the final iteration, the upper triangular $(n\times n)$ matrix $\boldsymbol{R}$ has been produced, together with a matrix $\boldsymbol{V}\in\mathbb{R}^{(m\times n)}$ that stores all the $\boldsymbol{v}_k$ vectors. In Algorithm 1, the factor $-2\frac{1}{\|\boldsymbol{v_k}\|^2}$ is stored in the scalar variable τ_k which is also returned as an element of vector $\boldsymbol{t}\in\mathbb{R}^{(n\times 1)}$ at the end of Householder QR. Finally, matrix $\boldsymbol{V}$ and vector $\boldsymbol{t}$ can be used to explicitly reconstruct $\boldsymbol{Q}$.

Algorithm 1 Householder QR [8]

- Input: Matrix $\boldsymbol{A}\in\mathbb{R}^{(m\times n)}$ -
- $\boldsymbol{x}_k$ stands for the k_{th} column vector -
- $\boldsymbol{x}_k(i)$ represents the i_{th} element in vector $\boldsymbol{x}_k$ -
- $\boldsymbol{dotProduct}(\boldsymbol{x},\boldsymbol{y},n)$ represents $\boldsymbol{x}^\top\boldsymbol{y}$ for vectors of length n -
- **for** $k=1$ to n **do**
  - Generate Householder reflector $\boldsymbol{v}_k$ -
  - $\boldsymbol{x}_k\leftarrow\boldsymbol{A}(k:m,k)$
  - $d1\leftarrow\boldsymbol{dotProduct}(\boldsymbol{x}_k,\boldsymbol{x}_k,m-k+1)$
  - $d2\leftarrow\sqrt{d_1}=\|\boldsymbol{x}_k\|_2$
  - $\boldsymbol{v}_k\leftarrow\boldsymbol{x}_k$
  - $\boldsymbol{v}_k(1)\leftarrow\boldsymbol{x}_k(1)+sign(\boldsymbol{x}_k(1))d_2$
  - $d_3\leftarrow\boldsymbol{dotProduct}(\boldsymbol{v}_k,\boldsymbol{v}_k,m-k+1)$
  - $\tau_k\leftarrow-\frac{2}{d_3}$
  - Update columns of $\boldsymbol{A}$ -
  - **for** $j=k$ to n **do**
    - $\boldsymbol{y}_j\leftarrow\boldsymbol{A}(k:m,j)$
    - $d_4\leftarrow\boldsymbol{dotProduct}(\boldsymbol{y}_j,\boldsymbol{v}_k,m-k+1)$
    - $d_5\leftarrow\tau_k d_4$
    - $\boldsymbol{y}'_j\leftarrow d_5\boldsymbol{v}_k+\boldsymbol{y}_j$
    - $\boldsymbol{A}(j:m,j)\leftarrow\boldsymbol{y}'_j$
  - **end for**
- **end for**
- **return** The upper triangular part of $\boldsymbol{A}$ containing the matrix $\boldsymbol{R}\in\mathbb{R}^{n\times n}$, matrix $\boldsymbol{V}\in\mathbb{R}^{(m\times n)}$ where individual columns are indexed $\boldsymbol{v}_k$ and a vector $\boldsymbol{t}\in\mathbb{R}^{(n\times 1)}$ containing all the τ_k values.

D. Tall-Skinny QR (TSQR)

TSQR employs a divide-and-conquer approach to factorise a matrix, while retaining optimality with respect to the computation-to-communication ratio [6]. Fig. 1 illustrates how the algorithm consists of a local stage followed by a merge stage. In the local QR stage, the $(m\times n)$ input matrix $\boldsymbol{A}$ is divided vertically into panels of size $(b_R\times n)$, where $b_R=2n$ to form a binary tree, and the total number of panels is $L=\lceil\frac{m}{b_R}\rceil$. Each panel is then factorised using Householder QR to obtain intermediate upper triangular $\boldsymbol{R}$ matrices. Next, in the merge stage, the intermediate $(n\times n)$ $\boldsymbol{R}$ factors of the local stage are merged by stacking in pairs, so that $\frac{L}{2}$ matrices of size $(b_R\times n)$ (i.e. $2n\times n$) are formed and factorised orderly as dictated by the tree structure. After the final decomposition of the merge stage, we end up with a single $\boldsymbol{R}$ matrix and a sequence of $\boldsymbol{V}$ and $\boldsymbol{t}$ factors which can be used to explicitly form $\boldsymbol{Q}$ [9].

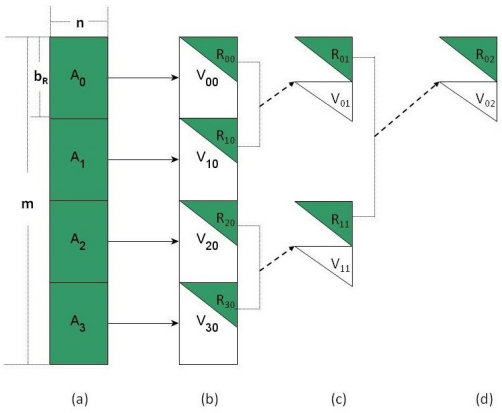

Fig. 1. Tall-Skinny QR, (a) local stage, (b, c & d) merge stage.

E. Communication-Avoiding QR (CAQR)

The CAQR algorithm was introduced by Demmel et al. [6] as a generalisation of TSQR for general rectangular matrices. Instead of a vertical division into panels, the matrix is divided into a grid of panels, where each panel is factorised using TSQR. In our system, an enhanced GPU implementation of CAQR is employed, with more details given in Section IV. A detailed discussion of the advantages of mapping CAQR onto a GPU can be found in [5].

F. Reducing QR-based Least Squares Computational Cost

As shown in (3), using the QR factorisation to solve a linear Least Squares system requires the computation of $\boldsymbol{Q_1^\top b}$. The typical steps to find $\boldsymbol{x}$ include the QR factorisation of $\boldsymbol{A}$ $(O(2mn^2 - \frac{2}{3}n^3))$, a matrix-vector multiplication for the computation of $\boldsymbol{Q_1^\top b}$ $(O(2mn))$ and finally a back-substitution to solve $\boldsymbol{Rx} = \boldsymbol{Q_1^\top b}$ $(O(n^2))$. Although a brute-force computation would reconstruct $\boldsymbol{Q_1}$ in an explicit form, an alternative approach would be to implicitly compute $\boldsymbol{Q_1^\top b}$ by appending $\boldsymbol{b}$ along the columns of $\boldsymbol{A}$ before performing the QR factorisation [7]. This result is particularly useful for the case of Householder QR, where $\boldsymbol{Q_1}$ is not explicitly formed and hence additional computations would be required to do so. After $\boldsymbol{Q_1^\top b}$ has been computed, the Least Squares system can be solved by performing back-substitution on $\boldsymbol{Rx} = \boldsymbol{Q_1^\top b}$. Moreover, in the case where the system's residuals or their norm is required, the same approach can be followed, as implied by the right-hand side of (4). In particular, after the computation of $\boldsymbol{Q_1^\top b}$, the norm of the residuals requires two additional dot products followed by a subtraction and a square root operation. The above technique is used in combination with the QR algorithms mentioned above, as it will be described in Section IV.

III. Related Work

The emerging need for high-performance systems that handle Linear Regression tasks has recently led to novel FPGA and GPU designs. In [10], Abba et al. mapped the Levenber-Marquardt algorithm onto an FPGA-based architecture, optimised to solve non-linear Least Squares problems, instead of linear ones as in our work. In [11], an FPGA-based Cholesky decomposition is employed instead of QR. The focus is on the solution of triangular linear equations, without addressing non-square tall-skinny matrices, claiming a speed-up of $4.8\times$ in single-precision and $4.3\times$ in double-precision against a dual core CPU implementation. Despite being less computationally complex than QR, the Cholesky decomposition is a numerically unstable method [12], an issue which is not addressed in [11]. Since several applications cannot tolerate numerical instability, our proposed system can tackle a wider range of problems. In the context of large-scale linear Least Squares problems, Anderson et al. [5] presented a GPU implementation of the CAQR algorithm which achieves the state-of-the-art GPU performance for very tall-skinny matrices. In our system, we use a set of GPU kernels based on this work with extensions so that $\boldsymbol{Q_1^\top b}$ is computed implicitly during the QR decomposition. Finally, Rafique et al. [4] proposed an FPGA-based TSQR accelerator targeting tall-skinny matrices, which was used as the base of our architecture.

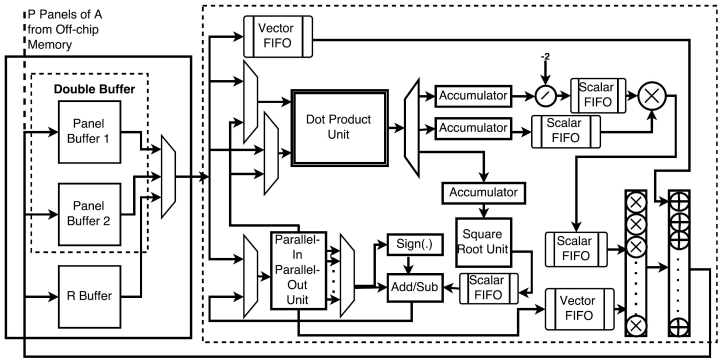

Fig. 2. Parallel architecture for tall-skinny Linear Systems

IV. Heterogeneous Computing System

A. FPGA Custom Architecture

Fig. 2 shows our proposed custom architecture. In our implementation, we use double-precision floating-point representation in order to account for the typical precision requirements of the data analysis community. The dataflow of the architecture is partially based upon the TSQR core proposed by Rafique et al. [4], with substantial enhancements that extend its functionality from performing QR decompositions to solving linear Least Squares systems, enable runtime flexibility with respect to the input matrix size along both dimensions and increase the maximum number of columns from 51 to 275 for the same FPGA device. The main strategy of the architecture can be identified in the exploitation of both the coarse-grained parallelism of TSQR and the fine-grained parallelism of Householder QR. Its main computation is the pipelined execution of Householder QR on the current panel, with the Double Buffered I/O subsystem scheduling the appropriate panel based on the TSQR tree structure.

1) Exploiting TSQR Parallelism: In TSQR, the factorisations of independent panels in the local stage and throughout the merge stage can be executed in parallel as exemplified by the tree structure in Fig. 1. The I/O subsystem includes three buffers as shown in Fig. 2. Initially, the panels of the input matrix are stored in the off-chip memory. The top two buffers constitute the Double Buffering mechanism while the third buffer serves as a way to execute all the TSQR merge stages on-chip and keep the off-chip memory transfers to a minimum. This is achieved by storing the intermediate $\boldsymbol{R}$ results after every panel factorisation in the third buffer, which, after being stacked in pairs as dictated by TSQR, are further factorised.

A detailed treatment of the inherent parallelism of Householder QR can be found in [6]. Since a single panel would keep the pipelined arithmetic units underutilised, a set of

independent panels is fed into the architecture. The number of panels in the set is denoted by P and is one of the runtime configurable parameters of the core, in contrast to [4] where it is fixed after compilation. Pipelining is done by executing the first Householder QR outer loop iteration for P panels one after the other, followed by the second outer loop iteration for the same P panels, and so on until the QR factorisations are finished for this set of panels. Then, the next set of P panels is processed in the same manner.

2) Introducing Flexibility: With reference to Fig. 2, the vector arithmetic units (i.e. the Dot Product reduction-tree unit and the Multiplier and Adder arrays) must have a fixed number of inputs, denoted by M, which meets the resource constraints of the target FPGA and in particular the available DSP blocks. As an example, M determines the number of multipliers in the first stage of the Dot Product unit. At the k_{th} iteration of Householder QR, P vectors of size $(b_R - k + 1)$ are fetched from one of the three buffers and fed into the Dot Product unit in a pipelined manner. Lines 7 and 8 of Algorithm 1 illustrate this process for a single panel where vector $\boldsymbol{x}_k$ is first fetched and then fed into the Dot Product unit. The alteration between the three buffers takes place as follows. One of the buffers from the Double Buffering mechanism is selected alternatively followed by the third buffer, and this process is repeated until the completion of the TSQR algorithm. In [4], when a column vector has to be fed into the Dot Product unit, if b_R is greater than M, the design needs to be recompiled and the FPGA reconfigured with M set equal to b_R. Moreover, if b_R exceeds the maximum value of M as determined by the available DSP blocks, then the input matrix cannot be factorised and hence the architecture is bounded by the DSP blocks. By employing a vector-partitioning strategy, our approach decouples M from the vector size, b_R, and consequently from the size of the input matrix, since b_R is equal to double the total number of columns, n. In this way, our architecture can tackle problems with varying b_R and n without reconfiguration and independently of the number of available DSP blocks, in contrast to [4].

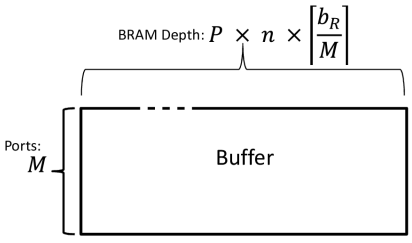

Fig. 3. On-chip Buffer Organisation for Vector Partitioning

Our mechanism implements the partitioning by means of appropriate memory organisation and buffer addressing. Each panel consists of n column vectors of length equal to b_R. Each column vector is split into $\lceil \frac{b_R}{M} \rceil$ slices of length equal to M. As depicted in Fig. 3, the RAM blocks are organised so that they provide a bandwidth of M words per cycle while having the required depth to accommodate P panels. In this way, if $b_R > M$, $\lceil \frac{b_R}{M} \rceil$ addresses that correspond to slices of the same column vector are read and fed into the Dot Product unit in consecutive cycles. Finally, at the output of the Dot Product unit, an accumulator is employed to sum together the partial results for each column vector. As a result, this technique offers a twofold gain. Firstly, it provides runtime flexibility with respect to the matrix number of columns and decouples it from the available DSPs. In contrast to [4], this allows the architecture to process matrices with more columns, bounded by on-chip memory capacity rather than DSPs. Secondly, in terms of hardware cost, it removes the need for massive partitioning multiplexers which would lead to an excessive LUTs utilisation, by means of a more efficient on-chip memory utilisation.

3) Solving Linear Least Squares Systems: As discussed in Section II, in the context of linear Least Squares systems, TSQR can be employed to compute the product $\boldsymbol{Q_1^\top b}$, or $\boldsymbol{Q_1^\top B}$ in the general case of multiple systems. This is achieved by appending matrix $\boldsymbol{B}$ along the columns of the $(m \times n)$ input matrix $\boldsymbol{A}$ and executing the outer loop of the Householder QR n times while the inner loop is executed $n + n_B$ times, where n_B is the number of columns of $\boldsymbol{B}$. In this way, matrix $\boldsymbol{B}$ is updated in every outer loop iteration and after the final iteration, it contains the product $\boldsymbol{Q_1^\top B}$. The novel hardware design proposed here follows the aforementioned mathematical approach by partitioning and storing matrix $\boldsymbol{B}$ along the columns of the panels of $\boldsymbol{A}$. In terms of hardware cost, the resource implications are minimal since in most real-life problems $n \gg n_B$ and hence there is no substantial increase on the memory requirements.

B. GPU

The GPU component of our system consists of an enhanced version of the design proposed by Anderson et al. [5]. The original design employs the CAQR algorithm for the QR factorisation of tall-skinny matrices and manages to outperform competing implementations by up to $13\times$. Our enhancement employs the method analysed in Section II for the calculation of $\boldsymbol{Q_1^\top B}$ by means of a redesigned kernel, which follows the modified Householder QR and appends matrix $\boldsymbol{B}$ along the columns of the input matrix. In this way, in the context of linear Least Squares problems, the computation-to-communication ratio is increased and our GPU module achieves a competitive performance which at points outperforms our custom architecture.

C. CPU

The CPU serves primarily as a supervisor for the other devices, but also as a complementary computing unit. Software running in the CPU is responsible for assigning the Least Squares workload to the computing device with the highest performance based on the matrix shape and size. At a system level, we exploit the fact that in specific matrix-size ranges either the FPGA-based custom architecture or the GPU module dominates in terms of performance and enable the CPU to select the highest performer for the target matrix. The FPGA performance is estimated using the developed modelling framework while the GPU module is subject to profiling as a pre-processing step. In the case of very small matrices where the overhead of data transferring is not amortised by the performance of either device, the Least Squares problem is solved by the CPU using parallel linear algebra libraries, such as OpenBLAS. Finally, for the solution of the Least Squares problem, the CPU executes the back-substitution as soon as $\boldsymbol{Q_1^\top B}$ becomes available from the other platforms. If the norm of the residuals is required, the CPU executes two additional dot products followed by a subtraction and a square root operation as dictated by (4).

V. Modelling Framework

To quantitatively determine the performance, the FPGA resource utilisation and the optimal configuration of the custom architecture, an analytical modelling framework was developed. The configurable architectural parameters comprise the size of the vector arithmetic units, M, and the number of panels, P, that will be active in parallel in the pipeline. All models have been verified empirically.

A. Performance Model

The primary performance metrics of interest are the execution time and throughput of our core. The execution time measurement begins with the input matrix already transferred to the FPGA's off-chip memory and reaches the point where $\boldsymbol{Q_1^\top B}$ is available on the CPU side. As shown in (5), the latency of Householder QR for a set of P panels, T_{hqr}, is a function of parameters M and P and the number of columns of $\boldsymbol{A}$ and $\boldsymbol{B}$, i.e. n and n_B respectively. T_{hqr} is broken down into two components: the critical path and the additional delays.

$$\mathrm{T}_{hqr} = n \times \mathrm{T}_{critical\ path}(P, M) + \mathrm{T}_{delays}(P, M, n, n_B) \quad (5)$$

where n is also the number of Householder QR outer loop iterations. The critical path models the aggregate latency of the floating-point units along the critical path for all P panels in each outer loop iteration as given by (6).

$$\begin{aligned} \mathrm{T}_{critical\ path} \quad &= \mathrm{T}_{dot} + \mathrm{T}_{accum} + \mathrm{T}_{sqrt} + \mathrm{T}_{add} \\ &+ \mathrm{T}_{dot} + \mathrm{T}_{accum} + \mathrm{T}_{div} + \mathrm{T}_{mult} \\ &+ \mathrm{T}_{mult} + \mathrm{T}_{add} + \mathrm{T}_{FIFOs} \end{aligned} \quad (6)$$

with

$$\begin{aligned} \mathrm{T}_{dot} \quad &= \mathrm{T}_{mult} + \mathrm{T}_{add} \times \lceil log_2 M \rceil \\ \mathrm{T}_{accum} \quad &= \left(9 + \left\lceil \frac{b_R}{M} \right\rceil\right) 1(b_R > M) \text{ (Flopoco operator)} \\ \mathrm{T}_{FIFOs} &= (P-1) \times \left\lceil \frac{b_R}{M} \right\rceil + 9 \text{ (implementation dependent)} \end{aligned}$$

where $1(b_R > M)$ is 1 if $b_R > M$ and 0 otherwise. The exact values depend on the latencies of double-precision floating-point operators and in our case yield $\mathrm{T}_{critical\ path} = 227 + (P-1) \times \lceil \frac{b_R}{M} \rceil + 24 \times \lceil log_2 M \rceil$, based on the Xilinx Coregen library and the Flopoco accumulator latency [13]. The additional delays capture the on-chip buffer write-back of the results for all outer loop iterations and are measured by profiling. Finally, the total number of local Householder QRs in all the TSQR stages is calculated based on the tree structure of Fig. 1 as shown in (7), with $L = \lceil \frac{m}{b_R} \rceil$, while the overall TSQR execution time, which is also the execution time for the computation of $\boldsymbol{Q_1^\top B}$, is given by (8).

$$\text{No. of Local QRs} = \sum_{i=0}^{\lceil log_2 L \rceil} \left\lceil \frac{L}{2^i P} \right\rceil \quad (7)$$

$$\mathrm{T}_{TSQR} = \mathrm{T}_{hqr} \times \text{No. of Local QRs} \quad (8)$$

B. Resource Model

From a resource perspective, the main design constraints are the number of DSPs and on-chip RAM blocks of the target FPGA. Assuming that all floating-point multipliers are implemented with DSP slices, the value of M is limited by the number of DSPs in the target FPGA. The total number of DSPs used by the core are given by constraint (9).

$$D_{mult} \times (2M + 1) + D_{ctrl} \leq D_{fpga} \quad (9)$$

where D_{fpga} is the number of available DSPs, D_{mult} are the DSPs per multiplier and D_{ctrl} is the implementation-dependent number of DSPs used by the control logic (in our case, double-precision is used, where $D_{mult} = 9$ based on the Xilinx Coregen library and $D_{ctrl} = 16$).

In terms of on-chip RAM, as shown in Fig. 2, the core has three buffers and two vector FIFOs of equal size with a capacity requirement as given by (10).

$$5 \times P \times (b_R \times n') \times WL \leq B_{fpga} \quad (10)$$

where WL is the wordlength, B_{fpga} is the available on-chip Block RAM and $n' = n + n_B$ is the number of columns in a panel after appending matrix $\boldsymbol{B}$.

C. Configuration Optimisation Framework

The developed optimisation framework aims at determining the values of the configurable parameters, M and P, that achieve the lowest execution time for a given matrix size and available hardware resources. Starting with M, we pose the following optimisation problem:

$$\min_{M \in \mathbb{Z}^+} |M - b_R|, \text{ s.t. } M \leq \frac{D_{fpga} - D_{mult} - D_{ctrl}}{2 \times D_{mult}} \quad (11)$$

Based on this formulation, the optimal value of M is the one that minimises partitioning and is given by the positive integer that lies closest to b_R while it satisfies the DSP constraint (9). To comprehend the reasoning behind minimising partitioning, it is essential to consider the effect of the selection of M on the performance for a given FPGA device. If the size of each column vector, b_R, is less than M, the vector is zero-padded to match M prior to entering a vector arithmetic unit and the tree-reduction Dot Product unit will contain redundant addition stages, and hence latency cycles. On the other hand, in the case where b_R is much greater than M, the partitioning of each vector will add substantial overhead to the overall execution time because of the required accumulation of the partial results at the output of the Dot Product unit.

Similarly to the parameter M and after it has been set, the number of panels P is determined through analysis. The critical factors for the selection of P are the architecture's pipeline depth, the available amount of on-chip RAM and the total number of Householder QRs for the TSQR algorithm of the input matrix. The main pipeline of the core consists of the Dot Product unit and the accumulator followed by the Square Root unit. At this point, the results of the Square Root unit are fed back to the Dot Product unit for the completion of the current outer loop iteration of the Householder QR algorithm and, therefore, no more new vectors need to be fetched from the buffers in order to keep the pipeline utilised. The pipeline depth is given by (12), where T_{accum} is included only when $b_R > M$. Finally, the maximum number of panels that would yield full utilisation of the pipeline is given by (13).

$$\begin{aligned} Partition\ Level \quad &= \left\lceil \frac{b_R}{M} \right\rceil \\ Pipeline\ Depth \quad &= \mathrm{T}_{dot} + \mathrm{T}_{accum} 1(b_R > M) \\ &+ \mathrm{T}_{sqrt} \end{aligned} \quad (12)$$

$$P_{pipeline\ max} = \left\lfloor \frac{Pipeline\ Depth}{Partition\ Level} \right\rfloor \quad (13)$$

where $1(b_R > M)$ is 1 if $b_R > M$ and 0 otherwise.

Apart from the pipeline depth, P is also constrained by the size of the available on-chip memory. Depending on the level of partitioning, the number of addresses per panel is:

$$Addresses\ per\ Panel = Partition\ Level \times n' \quad (14)$$

where $n' = n + n_B$ is the number of columns in a panel after appending matrix $\boldsymbol{B}$. The maximum feasible value for P is given by (16), where the maximum addresses per buffer can be found by (15) with each address accommodating M words.

$$Addresses\ per\ buffer = \frac{B_{fpga}}{5 \times M \times WL} \quad (15)$$

$$P_{bram\ max} = \left\lfloor \frac{Addresses\ per\ buffer}{Addresses\ per\ Panel} \right\rfloor \quad (16)$$

where the factor of 5 in the denominator of (15) comes from the core's three buffers and two vector FIFOs of equal size, for a total of five memory structures, as shown in Fig. 2 and captured by constraint (10).

As a last step, the optimal number of panels, P_{opt}, that minimises the overall TSQR latency is found as in (17).

$$P_{opt} = \arg\min_{P} \mathrm{T}_{TSQR}(P),\ \text{s.t. } P \in [1, P_{pipeline\ max}] \quad (17)$$

where T_{TSQR} is a locally convex function of P, as given by (8). The proof of $\mathrm{T}_{TSQR}(P)$'s local convexity has been omitted due to space constraints. Finally, if P_{opt} requires more than the FPGA's on-chip Block RAM, the final value is saturated to $P_{bram\ max}$ as in (18).

$$P = \min(P_{opt}, P_{bram\ max}) \quad (18)$$

D. I/O Requirements

In a typical operation of our system, the FPGA core would factorise a set of P panels from the Double Buffer followed by P panels from the R buffer, before a new set of P panels has to be fetched from the off-chip memory. Therefore, the required I/O bandwidth can be estimated as in (19).

$$\text{I/O Bandwidth} = \frac{P \times (b_R \times n') \times WL}{2 \times T_{hqr}} \text{bits/cycle} \quad (19)$$

where the numerator is the product between the total number of words for P panels of size $(b_R \times n')$ and the wordlength in bits, denoted by WL. The denominator is the latency of factorising two sets of P panels, which corresponds to the available time for loading a new set of panels. Meeting the I/O bandwidth requirements allows the off-chip memory latency to be hidden and the FPGA core to be kept busy at all times without stalling.

VI. Experimental Evaluation

A. Experimental Setup

The custom architecture was implemented using VHDL and the Xilinx ISE Design Suite (v14.5) was used. The target platform was the Xilinx Virtex-6 SX475T FPGA with an achieved operating frequency of 200 MHz and a PCIe interface to the CPU. In all the experiments, an NVIDIA Tesla K20 GPU and an Intel CPU i7-4770 (@3.40 GHz) were used, with the latter having four cores with two hardware threads per core and with 16 GB RAM and 8 MB cache.

To compare the proposed heterogeneous system with existing works, we evaluated its performance in typical linear Least Squares scenarios. The workload of our benchmarks includes an input system of the form $\|\boldsymbol{Ax} - \boldsymbol{b}\|_2^2$, where $\boldsymbol{A} \in \mathbb{R}^{(m \times n)}$, $\boldsymbol{b} \in \mathbb{R}^m$ and $m \geq n$, and requires as output the value of $\boldsymbol{x}$ that minimises it. Since the aim of our design is to sustain high performance for any matrix size, the performance was measured as a function of m and n, and includes the data transfer times between the CPU and the computing devices. For each matrix size, our heterogeneous system selects the computing device with the highest performance based on the target CPU and GPU profiling information and our FPGA performance models. The existing works include OpenBLAS and CULA routines for the solution of linear systems targeting multi-core CPUs and GPUs respectively as well as the FPGA TSQR core with the highest reported performance [4] mapped onto the same device as the one used here together with OpenBLAS routines for the remainder of the computations that are required for the solution of the Least Squares system. The performance results were obtained as averages over several runs of the experimental workloads.

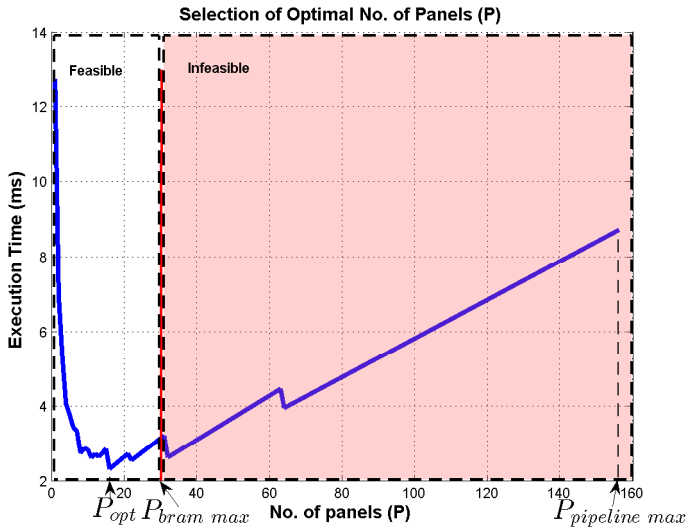

Fig. 4. Execution Time (T_{TSQR}) vs. P ($m = 6400$, $n' = 51$, $M = 100$)

B. FPGA Solver Optimal Configuration Paradigm

In all cases, parameters M and P in the FPGA design are set as determined by the presented optimisation framework. An instance of the optimisation procedure is shown in Fig. 4. For an input matrix $\boldsymbol{A} \in \mathbb{R}^{(6400 \times 50)}$ and $\boldsymbol{b} \in \mathbb{R}^{6400}$, each panel has a size of $(b_R \times n')$ with $n' = n + n_B = 50 + 1$ and $b_R = 2n = 100$. The target Virtex-6 SX475T FPGA includes 2,016 DSP blocks which leads to the constraint $M \leq 110$. In this scenario, given the input matrix size and the target FPGA DSPs, we are able to avoid vector partitioning altogether, because b_R is lower than the maximum permitted value of M. Therefore, with reference to (11), our optimisation framework sets M to be equal to b_R and hence M is assigned a value of 100.

In Fig. 4, $P_{pipeline\ max}$ can be seen to be equal to $Pipeline\ Depth$ with a value of 156. Moreover, $P_{bram\ max}$ is shown to have a value of 30 as determined by the amount of on-chip BRAM of the target device. The asymptotic increase

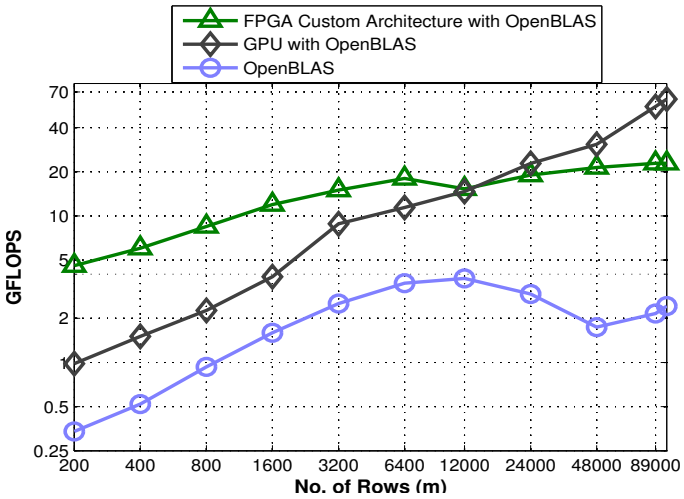

Fig. 5. Individual Solvers' Performance Scaling with respect to m ($n' = 51$, $n_B = 1$)

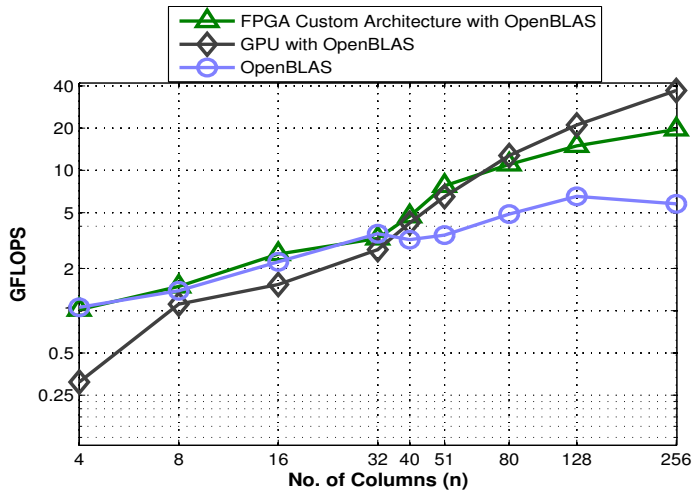

Fig. 6. Individual Solvers' Performance Scaling with respect to n ($m = 6400$, $n_B = 1$)

of the execution time with respect to P comes as a result of the two components of (8). On the one hand, T_{hqr}, which is the latency of Householder QR for a set of P panels, increases linearly with P due to the increased write-back overhead. On the other hand, the overall number of local QRs in the TSQR algorithm as given by (7) decreases exponentially with P and converges to a constant value, as dictated by the inter-stage dependencies of the TSQR tree, similarly to the one shown in Fig. 1. Moreover, because of the ceiling operator of (7), several consecutive values of P map to the same number of local QRs which leads to T_{TSQR} having several local minima. Eventually, after the number of local QRs reaches its asymptotic value, the T_{hqr} component dominates and therefore T_{TSQR} increases linearly with P. Since our objective is to minimise the overall execution time, P_{opt} is set to the minimum value of T_{TSQR} that lies on the left of $P_{bram\ max}$. In the example of Fig. 4, P_{opt} is 16 with a required I/O bandwidth of 145 MB/s and yields speed-ups with respect to $P_{pipeline\ max}$ and $P_{bram\ max}$ of $3.74\times$ and $1.35\times$ respectively.

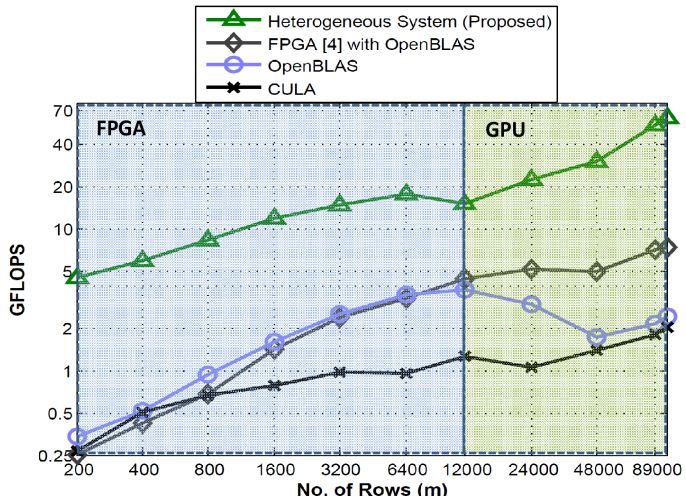

Fig. 7. Performance Scaling with respect to m ($n' = 51$, $n_B = 1$)

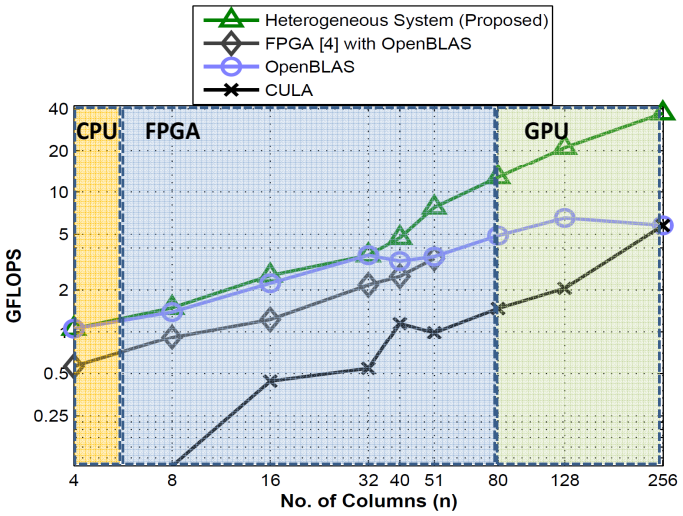

Fig. 8. Performance Scaling with respect to n ($m = 6400$, $n_B = 1$)

C. Heterogeneous Solver against Individual Devices

This section explores the performance gains of using the proposed heterogeneous system over using its individual devices for the solution of linear Least Squares systems. This is investigated by comparing the performance of the highest performer against the rest of the devices. Fig. 5 and 6 show the performance scaling of the individual devices that are part of the proposed heterogeneous system. With respect to Fig. 5 for a relatively small number of columns (e.g. $n' = 51$) a pure software Least Squares solver targeting a multi-core CPU does not provide a competitive high-performance solution. For the particular Least Squares workload size, in the interval $m \in [200, 12000)$ the FPGA solver with OpenBLAS reaches a speed-up in the range $1.04\times$ - $4.67\times$ ($2.50\times$ geometric mean) over GPU with OpenBLAS and $4.09\times$ - $13.54\times$ ($8.16\times$ geometric mean) over standalone OpenBLAS. In the interval $m \in (12000, 8900]$, GPU with OpenBLAS achieves a performance improvement of $1.20\times$ - $2.74\times$ ($1.85\times$ geometric mean) and $2.41\times$ - $25.84\times$ ($5.69\times$ geometric mean) over the FPGA architecture with OpenBLAS and the standalone OpenBLAS respectively.

With respect to the performance scaling with the number of columns as presented in Fig. 6, in the range $n \in [8, 80)$, the FPGA solver overperforms the GPU and the pure software modules by $1.11\times$ - $3.28\times$ ($1.62\times$ geometric mean) and 1.13 - $2.24\times$ ($1.42\times$ geometric mean) respectively. In the range $n \in [80, 256]$, the GPU implementation reaches speed-ups of $1.16\times$ - $1.89\times$ ($1.45\times$ geometric mean) and $2.61\times$ - $6.42\times$ ($3.78\times$ geometric mean) over the FPGA solver and OpenBLAS respectively. Finally, OpenBLAS manages to overperform both the FPGA and the GPU solvers in the range $n \in [4, 8)$ by a factor of $1.04\times$ and $3.41\times$ respectively.

D. Evaluation against Existing Works

Fig. 7 and 8 show the performance comparison results against existing works, indicating the switching points between the FPGA core, the GPU and the CPU implementations. In the first case, the performance scaling with respect to the number of rows of the input Least Squares system is shown. The number of columns, n, is set to 51, which is the maximum number of columns of [4] and allows for a direct comparison with this work. The second case presents the performance scaling with respect to the number of columns for 6400 rows which corresponds to a medium-high Least Squares system size.

Compared to OpenBLAS and CULA, we observe a speed-up in the range $1.13\times$ - $25.84\times$ ($4.70\times$ geometric mean) and $4.10\times$ - $32.67\times$ ($13.30\times$ geometric mean) respectively. Comparison with the best FPGA work yields a speed-up in the range $1.63\times$ - $18.07\times$ ($4.81\times$ geometric mean) with the maximum improvement observed as the matrices become more tall-skinny. The speed-up gains come mainly from the fact that the proposed system employs our mechanism to compute $\boldsymbol{Q_1^\top b}$ without forming $\boldsymbol{Q_1}$ explicitly and, hence, only $\boldsymbol{Q_1^\top b}$ and $\boldsymbol{R}$ are sent back to the CPU for the back-substitution. In contrast, [4] has to transfer back $\boldsymbol{R}$ together with all the intermediate $\boldsymbol{V}$ matrices and $\boldsymbol{t}$ vectors and then use software to reconstruct $\boldsymbol{Q_1}$, compute $\boldsymbol{Q_1^\top b}$ and finally execute the back-substitution, which adds substantial overhead. In the case of multiple Least Squares systems, where a matrix $\boldsymbol{B}$ is used in place of $\boldsymbol{b}$, the same performance pattern is observed.

As it can be seen in Fig. 8, [4] can process matrices with a maximum of 51 columns before reaching the device's DSP limit, and FPGA reconfiguration is required whenever the number of columns changes. In contrast to that, the proposed heterogeneous system can successfully handle any matrix size, until all the available memory is exhausted. In particular, for the FPGA module, the limiting factor is no longer the DSPs, but the Block RAMs. Our custom architecture is able to process matrices of different sizes without FPGA reconfiguration, with a maximum of 275 columns for the target device, reaching a 84.23% on-chip memory utilisation with potential for bigger matrices given a higher on-chip memory capacity. Finally, for the target FPGA, the maximum value of M is 110 reaching a DSP utilisation of 99.45%.

VII. Conclusion

This paper presents a heterogeneous solver for linear Least Squares systems as a step towards the high-performance solution of modern large-scale Linear Regression problems. The proposed system employs a novel FPGA-based custom architecture, a set of GPU kernels and parallel software libraries in a complementary way. Moreover, an analytical modelling framework is presented for the optimal configuration of the custom architecture and the estimation of its performance and resource utilisation. Experimental evaluation shows that by switching between the different computing platforms in an adaptive way based on the input matrix characteristics, it is possible to sustain a high performance across matrix sizes. As a result, in the common occurrence of massive data sets in various scientific branches, the runtime of computing either the residuals or the actual solution of Least Squares problems can be reduced by a factor of up to $18.07\times$. Potential future work includes an exploration of how we can maximize the utilisation of the FPGA and GPU components of the system by processing multiple Least Squares problems in parallel on the same device and also across devices. Finally, the proposed heterogeneous system could be extended by investigating the use of additional QR-based methods that are particularly suitable for specific Least Squares systems classes, such as square systems, followed by their highly optimised mapping onto the appropriate computing platform. Their potential integration into the system could help us tackle a wider range of real large-scale applications.

References

[1] L. Bottolo and S. Richardson, "Evolutionary Stochastic Search for Bayesian Model Exploration," *Bayesian Analysis*, vol. 5, no. 3, pp. 583–618, 09 2010.

[2] W. Wiedermann, M. Hagmann, and A. von Eye, "Significance tests to determine the direction of effects in linear regression models," *British Journal of Mathematical and Statistical Psychology*, vol. 68, no. 1, pp. 116–141, 2015. [Online]. Available: http://dx.doi.org/10.1111/bmsp.12037

[3] J. W. Demmel, *Applied Numerical Linear Algebra*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997.

[4] A. Rafique, N. Kapre, and G. Constantinides, "Enhancing Performance of Tall-Skinny QR Factorization using FPGAs," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, Aug 2012, pp. 443–450.

[5] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer, "Communication-Avoiding QR Decomposition for GPUs," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, May 2011, pp. 48–58.

[6] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal Parallel and Sequential QR and LU Factorizations," *SIAM J. Sci. Comput.*, vol. 34, no. 1, pp. 206–239, Feb. 2012.

[7] P. G. Constantine and D. F. Gleich, "Tall and Skinny QR Factorizations in MapReduce Architectures," in *Proceedings of the Second International Workshop on MapReduce and Its Applications*, ser. MapReduce '11. New York, NY, USA: ACM, 2011, pp. 43–50.

[8] W. Gander, "Algorithms for the QR-Decomposition," Tech. Rep., 1980.

[9] G. Ballard, J. Demmel, L. Grigori, M. Jacquelin, H. D. Nguyen, and E. Solomonik, "Reconstructing Householder Vectors from Tall-Skinny QR," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2013-175, Oct 2013.

[10] A. Abba, F. Caponio, A. Geraci, and G. Ripamonti, "Implementation of High Efficiency Non-Linear Least-Squares in FPGA Devices for Digital Spectroscopy," in *Nuclear Science Symposium Conference Record (NSS/MIC), 2010 IEEE*, Oct 2010, pp. 1371–1376.

[11] D. Yang, G. Peterson, H. Li, and J. Sun, "An FPGA Implementation for Solving Least Square Problem," in *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, April 2009, pp. 303–306.

[12] A. Björck, *Numerical methods for least squares problems*. SIAM, 1996.

[13] F. de Dinechin and B. Pasca, "Custom Arithmetic Datapath Design for FPGAs using the FloPoCo Core Generator," *Design & Test of Computers, IEEE*, vol. PP, no. 99, p. 1. [Online]. Available: http://dx.doi.org/10.1109/mdt.2011.44