Dataflow Acceleration of scikit-learn Gaussian Process Regression

Abstract—**Big data revolution has sparked the widespread use of predictive data analytics based on sophisticated machine learning tasks. Fast data analysis have become very important, and this fact stresses software developers and computer architects to deliver more efficient design solutions able to address the increased performance requirements. Dataflow computing engines from Maxeler has been recently emerged as a promising way of performing high performance computation, utilizing FPGA devices. In this paper, we focus on exploiting Maxeler's dataflow computing for accelerating Gaussian Process Regression from scikit-learn Python library, one of the most computationally intensive and with poor scaling characteistics machine learning algorithm. Through extensive analysis over diverse datasets, we point out which NumPy and SciPy functions forms the major performance bottlenecks that should be implemented in a dataflow acceleration engine and then we discuss the mapping decisions that enable the generation of parameterized dataflow engines. Finally, we show that the proposed acceleration solution delivers significant speedups for the examined datasets, while it also reports good scalability in respect to increased dataset sizes.**

I. Introduction

Nowadays, organizations collect data from a variety of sources, including business transactions, social media and sensors. Big data is a term that describes this large volume and high throughput data, which can be analysed to make intelligent predictions/decisions based on patterns. Big data analytics helps organisations exploit their data and use it to identify smarter business moves, more efficient operations, better services and achieve higher profits.

Data analytics are heavily depend on machine learning (ML) techniques, a field of computer science that evolved from the study of pattern recognition and computational learning theory in artificial intelligence. Machine learning explores the study and construction of algorithms that can learn from and make predictions on data. Such algorithms operate by building a model from example inputs in order to make data-driven predictions or decisions. Machine learning has found major applications in finance, healthcare, entertainment, robotics, and many more. Although machine learning has been around for decades, two relatively recent trends have sparked its widespread use, i.e. (i) data availability due to the huge amount of digital data being generated from smart devices and Internet of Things and ii) computation and storage capabilities offered by modern hardware that enables intelligent decisions and deliver value proposition.

A commonly used machine learning library is scikit-learn [1], which is a free software library for the Python programming language. It features various classification, regression and clustering algorithms and is designed to interoperate with the Python numerical and scientific libraries NumPy [2] and SciPy [3]. We focus our attention on Gaussian processes [4] that provide a principled, practical, probabilistic approach to learning in kernel machines. This gives advantages with respect to the interpretation of model predictions and provides a well-founded framework for learning and model selection. Theoretical and practical developments of over the last decade have made Gaussian processes a serious competitor for real supervised learning applications. However, its time complexity and the fact that it loses computational efficiency in high dimensional spaces are its main disadvantages. In this paper, we target the acceleration of Gaussian processes kernels, focusing on Gaussian Process Regression (GPR), which among scikit-learns' classification and regression algorithms appear to be some of the most time demanding.

For accelerating scikit-learn's GPR we exploit the efficiency of dataflow computing engines (DFE) introduced by Maxeler Technologies [5]. By carefully instrumenting, profiling and analysing the GPR's performance characteristics for model fitting and prediction, we extract the most computationally demanding kernels amenable for acceleration. Specifically, through extensive performance analysis we show that, the NumPy and SciPy Python functions: i) $scipy.linalg.cholesky()$ performing Cholesky factorization, ii) $scipy.linalg.cho\_solve()$ implementing the Cholesky solver, iii) $scipy.spatial.distance.pdist()$ and $scipy.spatial.distance.cdist()$ performing matrices distance calculation and iv) $numpy.dot()$ implementing matrix multiplication, are forming the performance dominant functions in GPR. To the best of our knowledge, this is one of the first papers that are either targeting to scikit-learn's GPR acceleration or utliizing Maxeler's dataflow computing technology for the accelerating the aforementioned functions. Cholesky factorization and solver are popular linear algebra kernels, thus their acceleration is actively researched in software [6], [7], GPU [8] and ASIC [9]. In [10], [11], their FPGA acceleration is proposed that relies on a direct hardware design solution in which for large size matrices, the FPGA memory interface becomes the bottleneck, mainly due to the limited adoption of the dataflow computing concepts. In this paper, we provide optimized DFE implementations for the aforementioned functions, not only by exploiting the high ILP traditionally offered by dataflow engines, but also applying customized data access and loop-tiling techniques to further improve performance. We show that the proposed DFE implementations can provide significant and scaled speedups for each of the examined kernels, ranging from 2× up to 8×. For the overall GPR $fit()$ and $predict()$ our accelerated solution delivers average speedup of 2.4× and 5.6×, respectively Finally, we note that the accelerated functions are frequently used for the solution of many other problems, thus the proposed high performance dataflow implementations could affect the efficiency of various applications.

II. Gaussian Processes in Python's scikit-learn

scikit-learn [1] is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms and it is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy. Every estimator for classification

Algorithm II.1 scikit-learn GaussianProcessRegressor().fit()

```
 1: procedure FIT($X$(inputs), $\mathbf{y}$(targets))
 2:     $X, \mathbf{y} \leftarrow$ check_X_y($X$, $\mathbf{y}$, ...)  check if arrays are compatible for computation.
 3:     $\mathbf{y} \leftarrow$ normalize_y()                          y vector normalization
 4:     self.log_marginal_likelihood_value_ $\leftarrow$ self.log_marginal_likelihood()
 5:     $K \leftarrow$ self.kernel_($X$)    get the kernel matrix $K$ using the method kernel_()
 6:     $K \leftarrow K + {\sigma_n}^2 I$             the noise ${\sigma_n}^2$ has the default value 1e-10
 7:     self.L_ $\leftarrow$ cholesky($K$)
 8:     self.alpha_ $\leftarrow$ cho_solve(self.L_ , $\mathbf{y}$)                  $\alpha = (\mathbf{y}/L)/L^\top$
 9: procedure LOG_MARGINAL_LIKELIHOOD()
10:     $K \leftarrow$ self.kernel_($X$)
11:     $K \leftarrow K + {\sigma_n}^2 I$
12:     $L \leftarrow$ cholesky($K$)
13:     alpha $\leftarrow$ cho_solve($L$ , $\mathbf{y}$)
14:     log_likelihood $\leftarrow -\frac{1}{2}\mathbf{y}^\top \cdot (\text{alpha}) - \sum_i \log L_{ii} - \frac{n}{2}\log 2\pi$      where
    $n$ = training samples
15: return log_likelihood
```

Algorithm II.2 scikit-learn kernels.RBF()

```
 1: procedure KERNEL_RBF($X_1$(test inputs), $X_2$(training inputs))
 2:     if $X_2$ = None then             when it is called from fit() or predict()
 3:         dists $\leftarrow$ pdist($X_1$, metric='sqeuclidean')
 4:         $K$ = np.exp(-0.5 * dists)
 5:         $K$ = squareform($K$)
 6:         np.fill_diagonal($K$, 1)
 7:     else                                     when it is called from predict()
 8:         dists = cdist($X_1$, $X_2$, metric='sqeuclidean')
 9:         $K$ = np.exp(-0.5 * dists)
10: return $K$
```

Algorithm II.3 scikit-learn GaussianProcessRegressor().predict()

```
1: procedure PREDICT($X_*$(test inputs))
2:     $X_* \leftarrow$ check_array($X_*$)                 check the test inputs matrix
3:     $K_* \leftarrow$ self.kernel_($X_*$, $X$)
4:     $\mathbf{y}_{mean} \leftarrow K_* \cdot$ (self.alpha_)
5:     $\mathbf{y}_{mean} \leftarrow$ undo_normal($\mathbf{y}_{mean}$)      only if we normalized y on fit()
6:     $\mathbf{v} \leftarrow$ cho_solve(self.L_, $K_*^\top$)
7:     $\mathbf{y}_{cov}$ = self.kernel_($X_*$) $- K_* \cdot \mathbf{v}$
8: return $\mathbf{y}_{mean}$(predictions), $\mathbf{y}_{cov}$(variance matrix)
```

or regression in scikit-learn machine learning library is a Python class, which implements the methods fit($X, \mathbf{y}$) and predict(X_*), where X is the $n \times D$ training matrix, $\mathbf{y}$ is the $n \times 1$ target vector and X_* is the $n_* \times D$ test matrix.

In this paper, we focus out attention on Gaussian Process Regression [4], which also heavily share its major algorithmic constructs with Gaussian Process classification. There are several ways to interpret Gaussian process (GP) regression models. One can think of a Gaussian process as defining a distribution over functions, and inference taking place directly in the space of functions, the function-space view. Considering inference directly in function space, we use a Gaussian process (GP) to describe a distribution over functions.

By definition, a Gaussian process is a collection of random variables, any finite number of which have a joint Gaussian distribution. A Gaussian process is completely specified by its mean function and covariance function. We define mean function $m(\mathbf{x})$ and the covariance function $k(\mathbf{x}, \mathbf{x}')$ of a real process $f(\mathbf{x})$ as:

$$\begin{aligned} m(\mathbf{x}) &= \mathbb{E}[f(\mathbf{x})] \\ k(\mathbf{x}, \mathbf{x}') &= \mathbb{E}[(f(\mathbf{x}) - m(\mathbf{x}))(f(\mathbf{x}') - m(\mathbf{x}'))] \end{aligned} \tag{1}$$

and write the Gaussian process as

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}),\, k(\mathbf{x}, \mathbf{x}')) \tag{2}$$

where, in our case the random variables represent the value of the function $f(\mathbf{x})$ at location $\mathbf{x}$.

A. Gaussian process regression in scikit-learn

In scikit-learn, a practical implementation of Gaussian process regression (GPR) is implemented that uses Cholesky decomposition, instead of directly inverting the matrix, since it is faster and numerically more stable. An estimator for regression is a Python object that implements the methods fit(X,$\mathbf{y}$) and predict(X_*). The estimator that implements Gaussian process regression is the class GaussianProcessRegressor. The constructor of an estimator may take as arguments the parameters of the model, e.g. the kernel RBF function.

We briefly analyze the $fit()$ and $predict()$ methods implemented in scikit-lean of the class $GaussianProcessRegressor()$, in order to familiarize with their algorithmic structure and naming conventions to be used later in the performance profiling analysis. In the fit method (Algorithm II.1), the $\mathbf{y}$ vector may be normalized by subtracting the mean of $\mathbf{y}$ from every observation. The $log\_marginal\_likelihood\_value$ is calculated in Line 4, using the corresponding method, which is presented below. The kernel function in Line 5 is the default RBF and the $kernel\_()function$ is demonstrated in Algorithm II.2. Moreover, the cholesky decomposition in Line 7 is computed with the $scipy.linalg.cholesky()$ function and $alpha\_value$ is calculated from $scipy.linalg.cho\_solve()$. The total cost of $fit()$ method is $\mathcal{O}(n^3 + n^2 D)$ due to the cholesky factorization.

For the RBF kernel (Algorithm II.2), the kernel function is $k(\mathbf{x}, \mathbf{x}') = \exp(-\frac{1}{2}|\mathbf{x} - \mathbf{x}'|^2)$, where $|\mathbf{x} - \mathbf{x}'|^2 = \sum_{d=1}^{D}(x_d - {x'}_d)^2$. Therefore, when the kernel is called from $fit()$ for example with the matrix X as argument, first the euclidean distance of every vector $\mathbf{x}$ with every other vector $\mathbf{x}'$ is calculated in line 3. Subsequently, in Line 4 every distance in K matrix passes through the $\exp()$ function and Lines 4-5 finalize K, computing the diagonal elements and the converting K to a square symmetric matrix. However, when we call the kernel from $predict()$ ((Algorithm II.3)) with two arguments, the distance of every $\mathbf{x}$ vector of X and every vector $\mathbf{x}_*$ of X_* is computed in line 8. Next, every distance element in K is given to the $\exp()$ in Line 9. Finally, K is returned. All distances are calculated from $scipy.spatial.distance.pdist()$ and $scipy.spatial.distance.cdist()$ functions. The total computational cost is $\mathcal{O}(n_1^2 D)$ in the first case and $\mathcal{O}(n_1 \cdot n_2 D)$ in the second case. We note that n_1 is the number of samples in X_1, n_2 is the number of samples in X_2 and D is the number of features.

The dominant functions in predict() (Algorithm II.3) is $cho\_solve()$ in Line 6 with a computational cost of $\mathcal{O}(n^2 \cdot n_*)$ and the $dot()$ in line 7 with a cost of $\mathcal{O}(n \cdot n_*^2)$. Additionally, kernel calculation in Line 3 takes $\mathcal{O}(n \cdot n_* \cdot D)$ operations, while kernel in Line 7 has a cost of $\mathcal{O}(n_*^2 \cdot D)$ operations.

III. Profiling analysis of Gaussian process regression in scikit-learn

In this section, we perform a detailed performance analysis of scikit-learn's Gaussian process $fit()$ and $predict()$ methods in order to evaluate performance in a finer manner and extract the computationally intensive code regions to be considered for acceleration. Our goal is to detect the most time consuming functions in scikit-learn's Gaussian process regression Python implementation. Thus, timers are placed in various code

TABLE I
DATASET CHARACTERIZATION.

Name	Num of samples	Num of features
CASP	45730	9
Online News Popularity	39797	60
BlogFeedback	60021	280
slices localization	53500	386

segments of $fit()$ and $predict()$ methods, in order to enable effective performance instrumentation.

For the analysis, we utilise real-datasets, obtained from UCI Machine Learning Repository [12]. As shown in Table I, we explore large and diverge datasets with a scaled number of features, in order to effectively capture the performance dependency to the feature size, i.e. to ensure that the "hot" code regions to be selected for acceleration are not present only in datasets with small feature space. Specifically, the datasets used by estimators for regression are:

- **Physicochemical Properties of Protein Tertiary Structure Data Set (CASP)**: This is a data set of Physicochemical Properties of Protein Tertiary Structure, taken from CASP 5-9. There are 45730 decoys and size varying from 0 to 21 armstrong.
- **Online News Popularity Data Set**: This dataset summarizes a heterogeneous set of features about articles published by Mashable in a period of two years. The goal is to predict the number of shares in social networks.
- **BlogFeedback Data Set**: This dataset contain features extracted from blog posts. The task is to predict how many comments the post will receive.
- **Relative location of CT slices on axial axis Data Set (slices localization)**: The dataset consists of 384 features extracted from CT images. The class variable denotes the relative location of the CT slice on the axial axis of the human body.

In Figure III, the pie charts demonstrate the percentage breakdown of the execution time in different functions of the methods. Fit and predict methods were executed on the four datasets with different feature sizes and large training and test sample sizes ($n = 5000, n_* = 5000$). We should note that $kernel\_RBF() : if$ consists of $scipy.spatial.distance.pdist()$, $numpy.exp()$, $scipy.spatial.distance.squareform()$ and $numpy.fill\_diagonal()$. Additionally, $kernel\_RBF() : if$ consists of $scipy.spatial.distance.cdist()$ and $numpy.exp()$. As shown, $scipy.linalg.cholesky()$ in the most time consuming function on fit method, while $kernel\_RBF() : if$ execution time increases as the number of features in the dataset becomes greater. Moreover, $scipy.linalg.cho\_solve()$ and $numpy.dot()$ functions are time demanding on predict method, sharing almost the total time in half. However as the number of features increases, $kernel\_RBF() : if$ and $kernel\_RBF() : else$ gain a greater percentage of the total predict execution time.

Selection for acceleration: According to previous discussion, for the acceleration of scikit-learn Gaussian process regression $fit()$ and $predict()$ methods, the following functions should be implemented in Maxeler's dataflow computational model.

- the function $scipy.linalg.cholesky(K_{n\times n}, ...)$.
- the function $scipy.linalg.cho\_solve(L_{n\times n}, B_{n\times n_*})$.
- the function $numpy.dot(A_{n_*\times n}, B_{n\times n_*})$.
- the set of functions $[scipy.spatial.distance.pdist(A_{n\times D}, ...)$, $numpy.exp(b_{n(n+1)/2})$, $scipy.spatial.distance.squareform$ $(c_{n(n+1)/2})$, $numpy.fill\_diagonal(E_{n\times n})]$, denoted as $pdist$ in the rest of the paper.
- the set of functions $[scipy.spatial.distance.cdist(A_{n_*\times D}$, $B_{n\times D}, ...)$, $numpy.exp(C_{n_*\times n})]$, denoted as $cdist$ in the rest of the paper.

IV. ACCELERATOR DESIGN FOR MAXELER DFES

The aforementioned functions have been accelerated exploiting Maxeler's multiscale dataflow computing technology. Maxelers multiscale dataflow computing [13] is a combination of traditional synchronous dataflow, vector and array processors. Loop level parallelism is achieved in a spatial, pipelined way, where large streams of data flow through a sea of arithmetic units, connected to match the structure of the compute task. DFEs provide two basic kinds of memory: FMem and LMem. FMem (Fast Memory) is on-chip Static RAM (SRAM) which can hold several MBs of data. Off-chip LMem (Large Memory) is implemented using DRAM technology and can hold many GBs of data. The overall system is managed by MaxelerOS, which sits within Linux and also within the dataflow engines manager. MaxelerOS manages data transfer and dynamic optimization at runtime.

Each DFE accelerated program, consists of CPU code written in a C programming language and a hardware configuration file (.max file) generated from MaxJ language. MaxJ code describes the dataflow datapath for a particular algorithm and the manager logic that interfaces the DFE accelerator with the host CPU. All data pre-processing (for example array layout re-ordering) takes place in the CPU and DFEs are called using SLiC[1] interface to compute the operations on data.

In this paper, we focus on the acceleration of the i) Cholesky decomposition and ii) Cholesky solver. For matrix multiplication, we reuse the freely available implementation found in [14]. The matrix distance, i.e. $scipy.spatial.distance.pdist(A, ...)$ and $scipy.spatial.distance.cdist(A, B, ...)$, accelerated functions have been designed reusing the tiled design of matrix multiplication DFE and adapting the implemented arithmetic operators, i.e. calculating $(a-b)^2$ instead of $a\times b$. Due to space limitations, we focus our discussion on the design decisions that enable tiled and high parallel implementations. All implementations consider double precision float arithmetic. A more detailed description, e.g. DFE's operations , MaxJ function implementation etc. can be found in [15].

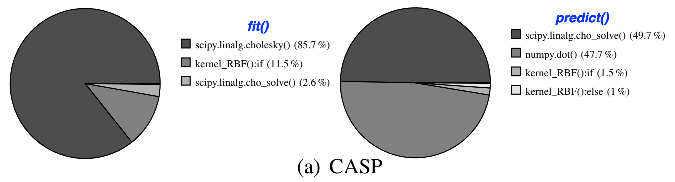
A. Cholesky decomposition acceleration

If $A \in \mathbb{R}^{n\times n}$ is a positive-definite symmetric matrix, then there is a unique Cholesky decomposition that factorises it into a lower triangular matrix and its transpose

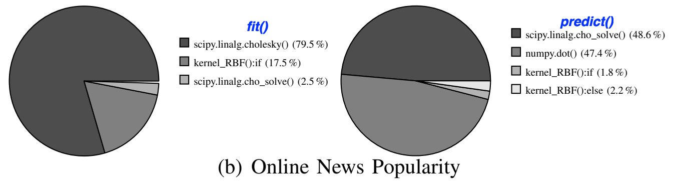
$$A = LL^\top \tag{3}$$

Analyzing the equation $A = LL^\top$, we obtain the following formula for the entries of L:

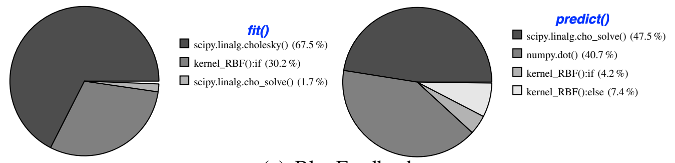
[1]SLiC (Simple Live CPU) interface is an automatically generated interface to the dataflow program, to call dataflow engines from attached CPUs



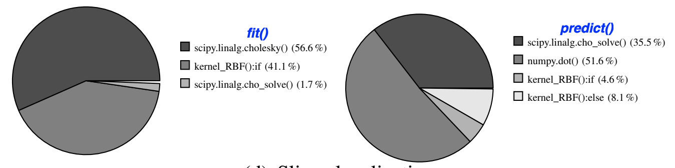
(a) CASP



(b) Online News Popularity



(c) BlogFeedback



(d) Slices localization

Fig. 1. Distribution analysis of computation latency for the $fit()$ and $predict()$ methods in Python's scikit-learn GPR.

$$\begin{aligned} l_{jj} &= \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2} \\ l_{ij} &= \frac{1}{l_{jj}}\left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}\cdot l_{jk}\right), \quad \text{for } i > j. \end{aligned} \tag{4}$$

Note that since older values of a_{ij} are not required for computing newer elements, they may be overwritten by the value of l_{ij}, hence the algorithm may be performed in place using the same memory for matrices A and L. Thus, we can compute the (i,j) entry of L if we know the data dependent entries to the left and above. The computation is usually arranged in either of the following orders i) the Cholesky-Banachiewicz algorithm (Row-Cholesky) starts from the upper left corner of the matrix L and proceeds to calculate the matrix row by row, ii) the Cholesky-Crout algorithm (Column-Cholesky) starts from the upper left corner of the matrix L and proceeds to calculate the matrix column by column.

The major issue in developing a Maxeler accelerated Cholesky implementation is how the matrix A should be transferred to the DFE and where should we place the output matrix L. Considering the data dependencies, in order to compute a column of L, a huge part of previously computed L-values must be accessed. For the computation of each new element in a column, a whole row of previously calculated L-values should be fetched to the computational area. The key to efficient dataflow implementations is to orchestrate the data movements to maximize the reuse of data while it is in the chip and minimize movement of data in and out of the chip. Therefore, in order to compute L elements as fast as possible without being constrained by the transfer speeds (bandwidth), we decide to keep L matrix on Fmem. However, Fmem cannot hold the whole L matrix for big values of n. For this reason we partition A into blocks so that the corresponding L blocks will fit in Fmem. A block-partitioned Cholesky algorithm has been implemented. Initially, A ($n \times n$ matrix) breaks into $n_{tiles} \times n_{tiles}$ tiles of size $n_B \times n_B$, where $n = n_{tiles} \cdot n_B$,

$$A = \begin{bmatrix} A_{1,1} & A_{2,1}^\top & A_{3,1}^\top & \dots & A_{n_{tiles},1}^\top \\ A_{2,1} & A_{2,2} & A_{3,2}^\top & \dots & A_{n_{tiles},2}^\top \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{n_{tiles},1} & A_{n_{tiles},2} & A_{n_{tiles},3} & \dots & A_{n_{tiles},n_{tiles}} \end{bmatrix} \tag{5}$$

Now in the first stage, we compute $L_{1,1}$ passing $A_{1,1}$ tile to the DFE: $L_{1,1}$ = cholesky_DFE($A_{1,1}$). In the same way we compute $L_{2,1}$, $L_{3,1}$... . However those tiles require data from $L_{1,1}$. Therefore, $L_{2,1}$ = cholesky_DFE($A_{2,1}$, $L_{1,1}$), $L_{3,1}$ = cholesky_DFE($A_{3,1}$, $L_{1,1}$) and so on.

At this point, the first column of L tiles has been calculated. Before moving on to process the second column of tiles we should calculate the matrix

$$S^{(1)} = \begin{bmatrix} L_{2,1} \\ L_{3,1} \\ \vdots \\ L_{n_{tiles},1} \end{bmatrix} \cdot \begin{bmatrix} L_{2,1}^\top & L_{3,1}^\top & \dots & L_{n_{tiles},1}^\top \end{bmatrix} \tag{6}$$

which is symmetric. This matrix multiplication is performed with the corresponding DFE accelerator found in [14]. Now using $S^{(1)}$ we can proceed in the second column of tiles of L. In order to compute $L_{2,2}$, Figure 2 shows that for $*$ element all light gray previously computed L elements are needed. Particularly the product of the two light gray rows of $L_{2,1}$ must be subtracted from a_*. This product can be taken directly from $S_{1,1}^{(1)}$, which contains all row-to-row products which are required from the computation of $L_{2,2}$. In the same way $S_{2,1}^{(1)}$ contains all row-to-row products for the computation of $L_{3,2}$, $S_{3,1}$ for $L_{4,2}$ and so on. Finally we store a submatrix of $S^{(1)}$

$$S_{acc}^{(1)} = \begin{bmatrix} L_{3,1}L_{3,1}^\top & \dots & L_{3,1}L_{n_{tiles},1}^\top \\ \vdots & \ddots & \vdots \\ L_{n_{tiles},1}L_{3,1}^\top & \dots & L_{n_{tiles},1}L_{n_{tiles},1}^\top \end{bmatrix} \tag{7}$$

In the second stage, we get $L_{2,2}$ = $cholesky\_DFE$ $(A_{2,2}, S_{1,1}^{(1)})$ and $L_{3,2}$ = $cholesky\_DFE$ ($\mathrm{A}_{3,2}, L_{2,2}, S_{2,1}^{(1)}$), $L_{4,2} = cholesky\_DFE$ $(A_{4,2}, L_{2,2}, S_{3,1}^{(1)})$, ... At this point, the second column of L tiles has been calculated. Before moving on the third column of tiles we calculate the matrix $S^{(2)}$, in the same way as $S^{(1)}$ before adding also $S_{acc}^{(1)}$:

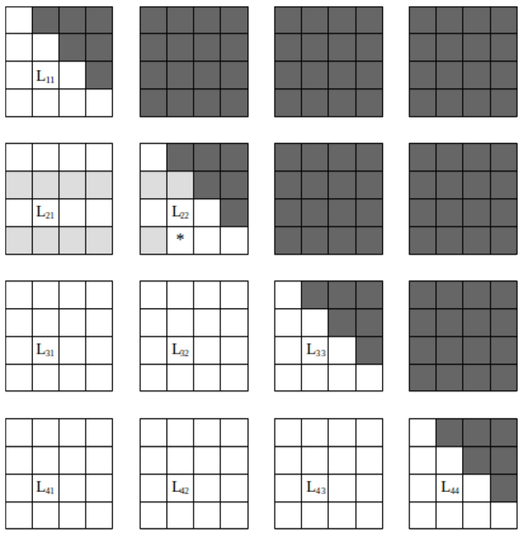


Fig. 2. Tiled Cholesky decomposition. L matrix partitioned into tiles. Data dependency for $*$ element.

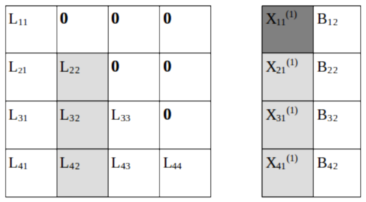


Fig. 3. Light gray tiles are sent to the DFE. The first dark gray tile is completely calculated.

$$S^{(2)} = \begin{bmatrix} L_{3,2} \\ \vdots \\ L_{n_{tiles},2} \end{bmatrix} \cdot \begin{bmatrix} L_{3,2}^{\top} & \dots & L_{n_{tiles},2}^{\top} \end{bmatrix} + S_{acc}^{(1)} \tag{8}$$

Using $S^{(2)}$ we can proceed in the third column of tiles of L, repeating the process described above, and finally compute the whole L.

B. Cholesky solve acceleration

Let A be a symmetric positive definite matrix of size $n \times n$ and B an $n \times m$ matrix. In this case the system $AX = B$ has a unique solution, since A can be factorized using the Cholesky decomposition to a product $LL^{\top}$. The linear system can be now written as $LL^{\top}X = B$. Setting $L^{\top}X = Y$ the system can then be solved by forward-substitution in lower triangular system $LY = B$, followed by back-substitution in upper triangular system $L^{\top}X = Y$.

However, $L^{\top}$ is an upper triangular matrix and the system $L^{\top}X = Y$ can be solved in an analogous way with the lower triangular system. This is performed, by transforming $L^{\top}$ to L' by applying $L'_{i,j} \leftarrow L_{n+1-i,n+1-j}$ and generating the upside-down Y' matrix through $Y'_{i,j} \leftarrow Y_{n+1-i,j}$, we transform the back-substitution to a forward-substitution. Specifically, we solve $L'X' = Y'$ and get X', which deliver X by applying the previous upside-down transformation on X'. In the rest of the section, we describe the lower triangular solver DFE utilised for Cholesky solve acceleration.

Algorithm IV.1 DFE Kernel

```
1:  procedure LTS_DFE ($L$, $B$ (which are columns of tiles))
2:      for All tiles in column do
3:          if first tile then
4:              for $i \leftarrow 1$, TILE_SIZE do
5:                  for $j \leftarrow 1$, TILE_SIZE do
6:                      $X_{ij}^{firsttile} \leftarrow B_{ij}^{firsttile} - \sum_{k=1}^{i-1} L_{ik}^{firsttile} X_{kj}^{firsttile}$
7:                      $X_{ij}^{firsttile} \leftarrow X_{ij}^{firsttile} / L_{ii}^{firsttile}$
8:          else
9:              for $i \leftarrow 1$, TILE_SIZE do
10:                 for $j \leftarrow 1$, TILE_SIZE do
11:                     $X_{ij}^{othertile} \leftarrow B_{ij}^{othertile} - \sum_{k=1}^{TILE\_SIZE} L_{ik}^{othertile} X_{kj}^{firsttile}$
12: return $X$
```

1) Lower triangular solver DFE: If L is a $n \times n$ lower triangular matrix and B an $n \times m$ matrix, the linear system $LX = B$ can be solved by forward-substitution. The first equation $l_{1,1} \cdot x_{1,j} = b_{1,j}$ only involves $x_{1,j}$ and thus one can solve for $x_{1,j}$ directly. The second equation only involves $x_{1,j}$ and $x_{2,j}$, and thus can be solved once one substitutes in the already solved value for $x_{1,j}$. Continuing in this way, the k-th equation only involves $x_{1,j}$ ... $x_{k,j}$ and one can solve for $x_{k,j}$ using the previously solved values for $x_{1,j}$ ... $x_{k-1,j}$. The resulting formula is: $x_{n,j} = \dfrac{b_{n,j} - \sum_{k=1}^{n-1} l_{n,k} \cdot x_{k,j}}{l_{n,n}}$.

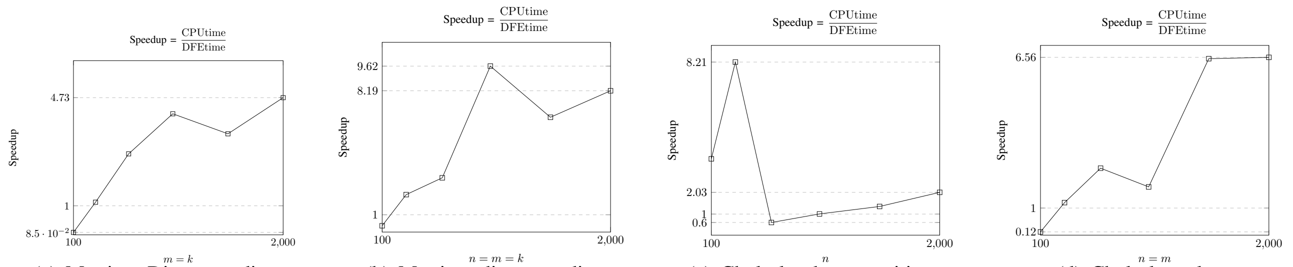
Regarding to Maxeler acceleration, the first step is to split $L_{n \times n}$ and $B_{n \times m}$ into tiles, where $nt = \dfrac{n}{TILE\_SIZE}$, $mt = \dfrac{m}{TILE\_SIZE}$ and $L_{i,i}$ will be lower triangular tiles.

$$L = \begin{bmatrix} L_{1,1} & \dots & \mathbf{0} \\ L_{2,1} & \dots & \mathbf{0} \\ \vdots & \ddots & \vdots \\ L_{nt,1} & \dots & L_{nt,nt} \end{bmatrix}, B = \begin{bmatrix} B_{1,1} & \dots & B_{1,mt} \\ B_{2,1} & \dots & B_{2,mt} \\ \vdots & \ddots & \vdots \\ B_{nt,1} & \dots & B_{nt,mt} \end{bmatrix} \tag{9}$$

Algorithm IV.1 shows the corresponding DFE kernel. For simplicity let us assume that $nt = 4$ and $mt = 2$, though the algorithm works for every nt and mt size. The first column of tiles from L and B are transfered to the DFE and perform one run. The result obtained from the DFE run is a column of tiles $X_1^{(1)}$ which overwrites B. In the second step another DFE run is performed with inputs shown in Figure 3 in light gray. This process continues until $X_{41}^{(4)}$ is obtained, and at this point the first column of X tiles is fully calculated. The same procedure can be repeated for the second column of B tiles, resulting in the computation of the whole X matrix.

V. Experimental evaluation

In this section we experimentally evaluate the efficiency of the proposed design solutions in terms of performance gains. The original scikit-learn scripts (CPU) ran on of Intel Core i5-3230M @ 2.60GHz processor, while the accelerated version (DFE) has been simulated and synthesized for the MAX4 architecture. All the DFE accelerators have been designed/-coded in MaxJ language and they have parameterized in order to automatically scale according to the applied tile size. All accelerators have been synthesized, placed and routed with a clock frequency of 200 MHz. To achieve best performance, the DFE have been synthesized to support the maximum



(a) Matrices Distance pdist (b) Matrices distance cdist (c) Cholesky decomposition (d) Cholesky solve

Fig. 4. Speedup results of DFE accelerated functions in respect to increasing dataset sizes.

TABLE II
MAXIMUM SUPPORTED TILING AND RESOURCE UTILISATION.

Description	Max. Tiling	LUTs	FFs	BRAMs	DSPs
Mat. dist.	192×192	76.22%	33.02%	60.15%	40.24%
Chol. decomp.	336×336	46.69%	37.59%	100.00%	71.73%
Chol. solver	416×416	55.85%	37.36%	86.75%	84.97%

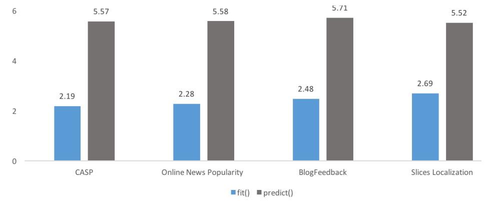


Fig. 5. Speedup of scikit-learn fit() and predict() methods for input matrix size 2000×2000.

parallelism. Since the number of parallel computations is equal to the tile size, this equates to maximising the tile size. However, the maximum tile size is limited due to Fmem size, DSPs for computing multiplications, LUTs and FFs. Table II shows the maximum tile size as well as the FPGA's resource utilisation for each DFE.

Figure IV-A shows the speedup achieved by the designed DFEs for each accelerated function, in respect to software execution. We note that software execution utilizes both the CBLAS [6] and LAPACK [7] libraries for optimized implementation of targeted functions. We evaluate the performance gains considering scaled matrices sizes. It is important to mention that the achieved speedup increases as moving towards larger dataset sizes, showing that the proposed designs forms a scalable solution. Matrices distance report a speedup of 4.7× for pdist and 8.2× for cdist, cholesky decomposition 2× while the Cholesky solver speedup reaches 6.5× for 2000×2000 input matrices. In cholesky decomposition it can be observed that there is a region where DFE implementation is not so efficient in respect to the software implementation, due to the fact that data movement to the DFE dominates the computation acceleration. However, as shown this effect is eliminated for larger dataset sizes, while the speedup slope remains positive, i.e. larger speedup gains are expected for larger datasets. Finally, Figure 5 reports the speedup achieved for the overall $fit()$ and $predict()$ of scikit-learn methods considering the datasets of section III. Average speedup of 2.4× and 5.6× for $fit()$ and $predict()$, respectively. We note that in the examined $fit()$ and $predict()$ implementations DFE-to-DFE communication has been performed through the host CPU, thus extra speedup gains are expected for designs utilising the MaxRing interconnection for high bandwidth DFE-to-DFE communication.

VI. CONCLUSIONS

This paper focused on the acceleration of Gaussian Process Regression, one of the most expressive and time-demanding predictive model in machine learning, exploiting Maxeler's dataflow computing technology. Our solution targets Python's scikit-learn package, in which a set of high demanding software functions have been accelerated delivering a measured average speedup of 8× in respect to software solution, and showing good scalability for increased dataset sizes. It is worth mentioning that the accelerated scikit-learn functions are frequently used for the solution of many other problems, thus the proposed high performance dataflow implementations could affect the efficiency of various applications.

REFERENCES

[1] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[2] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science and Engg.*, 13(2):22–30, March 2011.

[3] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2016-09-09].

[4] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[5] Maxeler Technologies. http://www.maxeler.com, 2016.

[6] CBLAS library. http://www.netlib.org/clapack/cblas/, 2016.

[7] LAPACK library. http://www.netlib.org/lapack/, 2016.

[8] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[9] Ardavan Pedram, Andreas Gerstlauer, and Robert A. van de Geijn. Algorithm, architecture, and floating-point unit codesign of a matrix factorization accelerator. *IEEE Trans. Computers*, 63(8):1854–1867, 2014.

[10] Antonio Roldao and George A. Constantinides. A high throughput fpga-based floating point conjugate gradient implementation for dense matrices. *ACM Trans. Reconfigurable Technol. Syst.*, 3(1):1:1–1:19, January 2010.

[11] Depeng Yang, Gregory D. Peterson, and Husheng Li. High performance reconfigurable computing for cholesky decomposition. In *in Proceedings of the Symposium on Application Accelerators in High Performance Computing (UIUC 09*, 2009.

[12] UCI Machine Learning Repository. http://www.archive.ics.uci.edu/ml/, 2016.

[13] Oliver Pell and Vitali Averbukh. Maximum performance computing with dataflow engines. *Computing in Science and Engg.*, 14(4):98–103, July 2012.

[14] Maxpower library. https://www.github.com/maxeler/maxpower, 2016.

[15] Suppressed for blind review.