J Sign Process Syst (2014) 77:117–129
DOI 10.1007/s11265-014-0897-9

Pipelined HAC Estimation Engines for Multivariate Time Series

Ce Guo · Wayne Luk

Received: 17 September 2013 / Revised: 12 April 2014 / Accepted: 21 April 2014 / Published online: 29 May 2014

Abstract Heteroskedasticity and autocorrelation consistent (HAC) covariance matrix estimation, or HAC estimation in short, is one of the most important techniques in time series analysis and forecasting. It serves as a powerful analytical tool for hypothesis testing and model verification. However, HAC estimation for long and high-dimensional time series is computationally expensive. This paper describes a pipeline-friendly HAC estimation algorithm derived from a mathematical specification, by applying transformations to eliminate conditionals, to parallelise arithmetic, and to promote data reuse in computation. We discuss an initial hardware architecture for the proposed algorithm, and propose two optimised architectures to improve the worst-case performance. Experimental systems based on proposed architectures demonstrate high performance especially for long time series. One experimental system achieves up to 12 times speedup over an optimised software system on 12 CPU cores.

Keywords Time series · HAC estimation · Big data · Acceleration engine · FPGA

1 Introduction

The study of time series is attracting the attention of researchers from various application areas such as financial risk management, statistical biology and seismology. One of the most important techniques in the study of time series is *heteroskedasticity and autocorrelation consistent (HAC) covariance matrix estimation*, or *HAC estimation* in short. This technique produces an estimation of the long-run covariance matrix for a multivariate time series, which provides a way to describe and quantify the relationship among different data components. The long-run covariance matrix plays a similar role as the ordinary covariance matrix of non-temporal multivariate data. However, HAC estimation of a long-run covariance matrix is different from the estimation of an ordinary covariance matrix for non-temporal data since HAC estimation considers the unique features of time series data such as serial correlation.

Today, HAC estimation becomes a standard method in the study of time series to extract statistical patterns or to verify the reliability of hypotheses. For instance, in research about stock markets [1–3], HAC estimation is used to quantify risks of trading strategies.

The computation of HAC estimation is time-consuming for long or high-dimensional time series. This drawback has become increasingly significant in recent years because the lengths and dimensions of real-world time series have been growing continuously. Data analysts take samples at short time interval to capture microscopic patterns. They also analyse multiple long time series simultaneously to discover causal relationships. However, it is usually necessary to compute HAC estimation as fast as possible in order to seize trading opportunities or to improve medical diagnosis. The conflict between data size and computational efficiency is especially serious in time-critical problems such as high-frequency trading and real-time electroencephalography analysis.

C. Guo (✉) · W. Luk
Department of Computing, Imperial College London, London, UK
e-mail: ce.guo10@imperial.ac.uk

W. Luk
e-mail: w.luk@imperial.ac.uk

This paper introduces a novel reconfigurable engine for speeding up HAC estimation. Our contributions include:

- a pipeline-friendly HAC estimation algorithm derived from a mathematical specification.
- the first pipelined accelerator for HAC estimation based on the proposed algorithm.
- an implementation of this pipelined architecture on a field-programmable gate array (FPGA) platform and an analysis of the related performance results.

Some early results from this paper has been published [4]. This paper reviews and extends this study, including detailed descriptions and examples for the mathematics and algorithms. There are two optimised hardware designs improving the worst-case performance of the architecture in [4]; the FPGA implementations of these designs, and their performance results, are also included.

The rest of the paper is organised as follows. Section 2 briefly describes the HAC estimation problem and review reconfigurable computing solutions for statistical data analysis. Section 3 presents our proposed pipeline-friendly HAC estimation algorithm and discusses its hardware-oriented features. Section 4 describes an initial hardware design that maps our algorithm to a fully-pipelined architecture. Section 5 proposes two optimised hardware architectures to improve the worst-cast performance. Section 6 provides experimental results of implementations of our hardware architectures, and explains experimental observations. Section 7 briefly concludes our work.

2 Background

HAC estimation for long and high-dimensional time series data is computationally demanding, and reconfigurable computing is a promising solution. In this section, we first provide a brief introduction to time series and HAC estimation. Then we review reconfigurable computing solutions to statistical data analysis and discuss how these studies inspire our research.

2.1 Time Series

A *time series* is a sequence of data points sampled from a data generation process at uniform time intervals. In this study, we focus on multivariate time series which are sequences in the form

$$\mathbf{y} = \langle y_1 \, , \; y_2 \, , \; \ldots \, , \; y_T \rangle \tag{1}$$

where T is the length of the time series; each data point y_i is a D-dimensional column vector in the form

$$y_i = [y_{i,1} \;\; y_{i,2} \;\; \cdots \;\; y_{i,D}]' \tag{2}$$

where $y_{i,1} \ldots y_{i,D}$ are *components* of the data point y_i. Note that a single-variable time series can be treated as a particular case of the multivariate time series where each data point contains a single component.

Two main research topics about time series are *pattern analysis* and *forecasting*. The former is a subject where mathematical and algorithmic methods are applied to time series data to extract patterns of interest; the latter is about forecasting future values of a time series using historical values. The HAC estimation problem studied in this research is important to both topics.

2.2 HAC Estimation

Consider a multivariate data generation process which theoretically satisfies

$$\mathbb{E}[y_t] = \mu \tag{3}$$

$$\mathbb{E}[(y_t - \mu)(y_{t-h} - \mu)'] = \Omega_h \tag{4}$$

where Ω_h is the autocovariance matrix of lag h. Suppose we have a time series sample $\mathbf{y}_T$ taken from this process. We can then estimate μ by taking the sample mean over T time steps

$$\hat{\mu} = \bar{\mathbf{y}}_T = \frac{1}{T} \sum_{t=1}^{T} y_t \tag{5}$$

In addition to the mean vector, it is also useful to know how data in different dimensions are correlated. Describing such a correlation is not trivial for time series because a data point may depend on historical states. As a consequence, commonly used correlation measurements for non-temporal data, like the ordinary covariance matrix, are not considered informative [5].

One statistically feasible correlation measurement of multivariate time series is the long-run covariance matrix defined by

$$S = \lim_{T \to \infty} \{T \cdot \mathbb{E}[(\bar{\mathbf{y}}_T - \mu)(\bar{\mathbf{y}}_T - \mu)']\} = \sum_{h=-\infty}^{\infty} \Omega_h \tag{6}$$

Unfortunately, it is impossible to compute the matrix S using Eq. 6 because the length of the required time series is infinite.

Heteroskedasticity and autocorrelation consistent (HAC) estimation is a technique that approximates S using a finite-length time series. This estimation can be achieved by

computing the *Newey-West estimator* [6] which is defined by

$$\hat{S} = \hat{\Omega}_0 + \sum_{h=1}^{H} k(\frac{h}{H+1})(\hat{\Omega}_h + \hat{\Omega}_h') \tag{7}$$

where

- H is the lag truncation parameter which may be set according to the length of the time series [7].
- $k(\cdot)$ is a real-valued kernel function. Following [7], we use lag truncation parameters H in the form

$$H = \lfloor \gamma T^{\frac{1}{3}} \rfloor \tag{8}$$

  where $\lfloor x \rfloor$ is the smallest integer not larger than x; γ is a data-dependent positive real number [7].
- $\hat{\Omega}_h$ is the estimate of the autocovariance matrix with lag h which can be computed by

$$\hat{\Omega}_h = \frac{1}{T} \sum_{t=h+1}^{T} (y_t - \hat{\mu})(y_{t-h} - \hat{\mu})' \tag{9}$$

This estimator can be treated as a weighted sum over a group of estimated autocovariance matrices, where the weights are determined by a kernel function. Discussions about kernel functions can be found in [6] and [8]. Variances of the Newey-West estimator can be found in [5] and [8].

2.3 Hardware Acceleration for Statistical Data Analysis

Hardware acceleration for time series data processing is not a well studied topic. It is only in recent papers where acceleration systems based on graphics processing units (GPUs) and field-programmable gate arrays (FPGAs) are proposed to process time series data. Sart et al. [9] propose both GPU-based and FPGA-based solutions to accelerate dynamic time wrapping (DTW) for sequential data. Wang et al. [10] develop a hardware engine for DTW-based sequence searching in time series. However, the problem investigated in these studies are matching and aligning problems. The underlying statistical patterns of time series data are not examined.

Preis et al. [11] use GPUs to accelerate the quantification of short-time correlations in a univariate time series. The correlations between components of a multivariate time series are not addressed by this work. Gembris et al. [12] present a real-time system to detect correlations among multiple medical imaging signals using GPUs. Their system is based on a simple correlation metric in which serial correlations are not considered. Our work is different from these two papers because both internal and mutual correlations in a multivariate time series are considered in HAC estimation.

Although hardware acceleration of statistical time series analysis has not been well studied, research on accelerating non-temporal multivariate data analysis has been conducted. Various data processing engines have been designed by mapping existing algorithms into hardware architectures. For example, Baker and Prasanna [13] mapped the Apriori algorithm [14] into an FPGA-based acceleration device for improved efficiency. Similar to the Apriori engine, hardware acceleration solutions for k-means clustering [15] and decision tree classification [16] are presented in [17] and [18] respectively.

Sometimes it is impossible or inappropriate to map an existing statistical analysis algorithm to hardware. This is typically due to the operating principles and resource limitations of the hardware platform. In this case, it is necessary to adapt existing algorithms or design new ones. Traditionally, hardware adaptations of data processing algorithms achieve parallelism by committing the same operation on multiple different data instances simultaneously – a form of single-instruction-multiple-data (SIMD) parallelism. For example, Moerland and Fiesler [19] analyse various machine learning algorithms for artificial neural networks. They simplify and parallelise the algorithmic operations, and propose efficient hardware architectures accordingly.

The flexibility of reconfigurable devices enables us to design pipelined data flow engines where different circuits for different computational stages are deployed. In other words, parallelism can also be achieved in a multiple-instruction-multiple-data (MIMD) manner. Data instances are streamed into the engine and processed in series by the pipeline. There is recent research where algorithms are designed or adapted for pipelined data flow engines. For example, Guo et al. [20] propose an FPGA-based hardware engine to accelerate the expectation-maximisation (EM) algorithm for Gaussian mixture models. The authors adapt the original EM algorithm such that it can be mapped to fully-pipelined hardware. The hardware based on this adapted algorithm is shown to be very efficient in their experiments.

While real-time systems can often benefit from the speed and simplicity of hardware implementations, hardware acceleration of time series processing is not a well-studied topic. To the best of our knowledge, although there is recent research on accelerating pattern matching in time series [9, 10], our work is the first to apply reconfigurable computing to time series analysis.

3 Pipeline-Friendly HAC Estimation Algorithm

In this section, we provide a detailed description of the algorithmic methods proposed in [4]. We first explain why we do not map the existing algorithm to hardware. Then we show how the expression of $\hat{S}$ (Eq. 7) can be rewritten to eliminate

conditionals, to parallelise arithmetic, and to promote data reuse. Finally, we present our new estimation algorithm.

Our discussion in this section is based on [4] with additional material including insights behind the equations and an example execution of the pipeline-friendly HAC estimation algorithm.

3.1 Analysis of the Straightforward Algorithm

It is not difficult to design an algorithm following Eq. 7 to compute HAC estimation for a time series. This algorithm is shown in Algorithm 1. The subroutine AUTOCOV(h), the computational steps for a single autocovariance matrix, is described in Algorithm 2 where $\hat{\mu}_m$ is the sample mean of $y_{1,m} \dots y_{T,m}$. We call this algorithm the *straightforward HAC estimation algorithm* in [4] because it is straightforwardly derived from the definition of the Newey-West estimator. This algorithm is implemented in many software packages such as the 'sandwich' econometrics package [21] and the GNU regression, econometrics and time-series library [22].

Algorithm 1 Straightforward HAC Estimation Algorithm

1: $\hat{S} \leftarrow$ AUTOCOV(0)
2: **for** $h \in [1..H]$ **do**
3: $\quad \hat{\Omega} \leftarrow$ AUTOCOV(h)
4: $\quad \hat{S} \leftarrow \hat{S} + k(\frac{h}{H+1})(\hat{\Omega} + \hat{\Omega}')$
5: **return** $\hat{S}$

Algorithm 2 AUTOCOV(h)

1: $\hat{\Omega} \leftarrow \mathbf{0}_{D \times D}$
2: **for** $t \in [(h+1)..T]$**do**
3: $\quad$ **for**$i \in [1..D]$**do**
4: $\quad\quad$ **for**$j \in [1..D]$**do**
5: $\quad\quad\quad \hat{\Omega}_{i,j} \leftarrow \hat{\Omega}_{i,j} + (y_{t,i} - \hat{\mu}_i)(y_{t-h,j} - \hat{\mu}_j)$
6: **return** $\hat{\Omega}$

Taking arithmetic operations as basic operations, the time complexity of the algorithm is $O(D^2HT)$, which means that the execution time is likely to grow linearly with D^2, H and T. Moreover, the lag truncation parameter H should grow with T in order to keep the results statistically feasible [7]. As a consequence, the algorithm may become computational demanding with long and high-dimensional time series.

The most time-consuming part of the algorithm is the computation of the autocovariance matrix. This process is shown in Algorithm 2. Technically, it is straightforward to implement this part in a reconfigurable device. However, memory efficiency is low because only one multiplication and one addition are executed after two data access operations, and we may therefore suffer memory bottleneck [23]. It is critical to find optimisations of the algorithm that avoids such a bottleneck. To make a fundamental difference from CPUs in performance, it is unwise to merely map the straightforward algorithm to a reconfigurable computing platform.

3.2 A Novel Derivation for $\hat{S}$

We build the mathematical foundation of our pipeline-friendly algorithm by deriving a novel expression of $\hat{S}$ (Eq. 7). We first simplify $\hat{S}$ by centralising the data and merging Ω_0 into the weighted sum. This simplification eliminates redundant arithmetic operations and complex conditional logic in the computation. Then we propose an expression of $\hat{S}$ using vector algebra. This expression exposes the parallelism in arithmetic operations and enables considerable data reuse.

The computation of $\hat{S}$ only concerns centralised values of data points. In other words, for all data points y_t, only the centralised value $y_t - \hat{\mu}$ is used in the computation. As the centralised value may be used multiple times, we precompute and store them to avoid redundant subtractions. More specifically, we precompute the centralised time series $\mathbf{u} = \langle u_1, u_2, \dots, u_T \rangle$ by

$$u_t = y_t - \hat{\mu} \tag{10}$$

Let u_t be a zero vector if $t \notin [1..T]$. This is for simplicity in the presentation and implementation of related algorithms, which will be illustrated later. By our precomputing scheme, Eq. 9 can be rewritten as

$$\hat{\Omega}_h = \frac{1}{T} \sum_{t=h+1}^{T} u_t u'_{t-h} \tag{11}$$

When $h = 0$, Ω_h is degraded from an autocovariance matrix to an ordinary covariance matrix which is always symmetric. Therefore

$$\Omega_0 = \Omega'_0 \tag{12}$$

By Eq. 7, we merge Ω_0 to the weighted sum by setting

$$w_h = \begin{cases} \frac{1}{2} & \text{if } h = 0 \\ k(\frac{h}{H+1}) & \text{if } 0 < h \le H \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

By introducing the coefficient $w_0 = \frac{1}{2}$, the expression of $\hat{S}$ in Eq. 7 can be simplified as

$$\hat{S} = w_0(\hat{\Omega}_0 + \hat{\Omega}_0') + \sum_{h=1}^{H} w_h(\hat{\Omega}_h + \hat{\Omega}_h') = \sum_{h=0}^{H} w_h(\hat{\Omega}_h + \hat{\Omega}_h') \quad (14)$$

In other words, the term $\hat{\Omega}_0$ is merged to the weighted sum operation. The major consideration behind this merging operation is that we hope to avoid spacial cases in hardware computation. Without merging, $\hat{\Omega}_0$ is not a part of the weighted sum. As a result, when designing the hardware, one needs to fork the main computational logic to handle this particular case. By merging this computation in the main computational logic, we keep the simplicity of the hardware.

The autocovariance matrix Ω_h can be expanded using Eq. 11 and the expression of $\hat{S}$ can be rewritten as

$$\hat{S} = \sum_{h=0}^{H} w_h[(\frac{1}{T}\sum_{t=h+1}^{T} u_t u_{t-h}') + (\frac{1}{T}\sum_{t=h+1}^{T} u_t u_{t-h}')'] = \frac{1}{T}(\Psi + \Psi') \quad (15)$$

where

$$\Psi = \sum_{h=0}^{H} w_h \sum_{t=h+1}^{T} u_t u_{t-h}' \quad (16)$$

Therefore once Ψ is obtained, $\hat{S}$ can be easily computed. Now we introduce a parameter c, which is a positive integer less than or equal to $H+1$. We further define a quantity G as

$$G = \lceil \frac{H+1}{c} \rceil - 1 \quad (17)$$

where $\lceil x \rceil$ is the smallest integer not less than x.

$$\Psi = \sum_{g=0}^{G} \sum_{h=gc}^{gc+c-1} w_h \sum_{t=h+1}^{T} u_t u_{t-h}' \quad (18)$$

The function of the parameter c and the quantity G will be illustrated later. If c is a factor of $(H+1)$ then Eq. 18 obversely holds. If not, some terms with $h > H$ will be calculated, but Eq. 18 still holds in this case because $w_h = 0$ for all $h > H$. The value of a single entry of Ψ can be computed by

$$\Psi_{i,j} = \sum_{g=0}^{G} \sum_{h=gc}^{gc+c-1} w_h \sum_{t=h+1}^{T} u_{t,i} \cdot u_{t-h,j} = \sum_{g=0}^{G} \tilde{w}_{g,c}\, \tilde{r}_{g,c,i,j} \quad (19)$$

where

$$\tilde{w}_{g,c} = \begin{bmatrix} w_{gc} & w_{gc+1} & \dots & w_{gc+c-1} \end{bmatrix} \quad (20)$$

$$\tilde{r}_{g,c,i,j} = \begin{bmatrix} \sum_{t=gc+1}^{T} u_{t,j} \cdot u_{t-gc,i} \\ \sum_{t=gc+2}^{T} u_{t,j} \cdot u_{t-(gc+1),i} \\ \vdots \\ \sum_{t=gc+c}^{T} u_{t,j} \cdot u_{t-(gc+c-1),i} \end{bmatrix} \quad (21)$$

By Eq. 21, we decompose $\tilde{w}_{g,c}$ as a product of two vectors. The aim this decomposition is to seek for data reuse opportunities. The vector $\tilde{w}_{g,c}$ can be constructed using the weights precomputed by Eq. 13. We only need to focus on the computation of $\tilde{r}_{g,c,i,j}$. It can be observed that the structures of all entries in $\tilde{r}_{g,c,i,j}$ are similar. To promote data reuse, we further simplify the expression of $\tilde{r}_{g,c,i,j}$ to observe the data access pattern. Aligning the lower bounds of the summation operators in Eq. 21, we have

$$\tilde{r}_{g,c,i,j} = \begin{bmatrix} \sum_{k=1}^{T-gc} u_{k+gc,j} \cdot u_{k,i} \\ \sum_{k=1}^{T-gc-1} u_{k+gc+1,j} \cdot u_{k,i} \\ \vdots \\ \sum_{k=1}^{T-gc-c+1} u_{k+gc+c-1,j} \cdot u_{k,i} \end{bmatrix} \quad (22)$$

We have defined that $u_t = 0$ when $t > T$. Hence we can set the upper bounds of all summation operations in Eq. 22 to $(T - gc)$. Then the expression of $\tilde{r}_{g,c,i,j}$ can be further simplified:

$$\tilde{r}_{g,c,i,j} = \begin{bmatrix} \sum_{k=1}^{T-gc} u_{k+gc,j} \cdot u_{k,i} \\ \sum_{k=1}^{T-gc} u_{k+gc+1,j} \cdot u_{k,i} \\ \vdots \\ \sum_{k=1}^{T-gc} u_{k+gc+c-1,j} \cdot u_{k,i} \end{bmatrix} = \sum_{k=1}^{T-gc} u_{k,i} \begin{bmatrix} u_{k+gc,j} \\ u_{k+gc+1,j} \\ \vdots \\ u_{k+gc+c-1,j} \end{bmatrix} \quad (23)$$

In Eqs. 22 and 23, we unify the lower bounds and upper bounds of the summation operations respectively. The expression of $\tilde{r}_{g,c,i,j}$ is converted into a scalar product of a vector. Given any $k \in [1..(T-gc)]$, the data entries involved in Eq. 23 constitute a contiguous subsequence of the i-th component of the centralised time series data.

In the mathematical reformulation discussed above, we partition the information collection process of $(H+1)$ different lags into G batches. In each batch, we collect statistical information from up to c lags. If the batch size c does not exactly divide the number of lags $(H+1)$, there must be a batch where the number of lags is less than c. We may still treat such a batch as a complete one by arranging redundant computation.

3.3 Pipeline-Friendly HAC Estimation Algorithm

We design a tangible algorithmic strategy to compute $\hat{S}$ using the equations developed in the last subsection. Following a top-down design approach, we first investigate how $\hat{S}$ can be obtained assuming that all $\tilde{r}_{g,c,i,j}$ can be computed; then we discuss the way to compute $\tilde{r}_{g,c,i,j}$.

Suppose we are able to compute the value of $\tilde{r}_{g,c,i,j}$ for all g, c, i and j. We can then compute all entries of Ψ by Eq. 19. Once all entries of Ψ are obtained, $\hat{S}$ can be computed by Eq. 14. More specially, the computational steps are shown in Algorithm 3.

Algorithm 3 is an algorithmic framework which does not access the data by itself. It queries the value of $\tilde{r}_{g,c,i,j}$ by invoking the subroutine PASS(g, c, i, j) in which $\tilde{r}_{g,c,i,j}$ is computed by passing through data. We design this subroutine following Eq. 23 and the detailed computational steps are shown in Algorithm 4. In Algorithm 3 and 4, the variables $\tilde{w}$ and $\tilde{r}$ correspond respectively to $\tilde{w}_{g,c}$ and $\tilde{r}_{g,c,i,j}$ in Eq. 19.

Algorithm 3 Pipeline-Friendly HAC Estimation

1: **for** $(i, j) \in [1..D] \times [1..D]$**do**
2: $\quad \Psi_{i,j} \leftarrow 0$
3: $\quad$ **for** $g \in [0..G]$**do**
4: $\quad\quad \tilde{w} \leftarrow \left[w_{gc}\ w_{gc+1}\ \ldots\ w_{gc+c-1}\right]$
5: $\quad\quad \tilde{r} \leftarrow \text{PASS}(g, c, i, j)$
6: $\quad\quad \Psi_{i,j} \leftarrow \Psi_{i,j} + \tilde{w}\,\tilde{r}$
7: **return** $\frac{1}{T}(\Psi + \Psi')$

Algorithm 4 PASS(g, c, i, j)

1: $\tilde{r} \leftarrow \mathbf{0}_{D\times 1}$
2: **for** $k \in [1..(T - gc)]$**do**
3: $\quad \tilde{r} \leftarrow \tilde{r} + u_{k,i} \begin{bmatrix} u_{k+gc,j} \\ u_{k+gc+1,j} \\ \vdots \\ u_{k+gc+c-1,j} \end{bmatrix}$
4: **return** $\tilde{r}$

For example, for a HAC estimation task with $T = 7$, $c = 3$, $h = 4$, the computational steps are presented in Table 1.

We call Algorithm 3 the *pipeline-friendly HAC estimation algorithm* because we consider its most time-consuming subroutine, PASS(g, c, i, j), as an excellent candidate to be mapped to a pipelined hardware architecture. The reasons are as follows.

Table 1 A Running Example: Computing $\Psi_{i,j}$.

g	k	operation
-	-	$\Psi_{i,j} \leftarrow 0$
0	-	$\tilde{w} \leftarrow [w_0\ w_1\ w_2]$
0	-	$\tilde{r} \leftarrow \mathbf{0}_{D\times 1}$
0	1	$\tilde{r} \leftarrow \tilde{r} + u_{1,i}\,[u_{1,j}\ u_{2,j}\ u_{3,j}]'$
0	2	$\tilde{r} \leftarrow \tilde{r} + u_{2,i}\,[u_{2,j}\ u_{3,j}\ u_{4,j}]'$
0	3	$\tilde{r} \leftarrow \tilde{r} + u_{3,i}\,[u_{3,j}\ u_{4,j}\ u_{5,j}]'$
0	4	$\tilde{r} \leftarrow \tilde{r} + u_{4,i}\,[u_{4,j}\ u_{5,j}\ u_{6,j}]'$
0	5	$\tilde{r} \leftarrow \tilde{r} + u_{5,i}\,[u_{5,j}\ u_{6,j}\ u_{7,j}]'$
0	6	$\tilde{r} \leftarrow \tilde{r} + u_{6,i}\,[u_{6,j}\ u_{7,j}\ 0]'$
0	7	$\tilde{r} \leftarrow \tilde{r} + u_{7,i}\,[u_{7,j}\ 0\ 0]'$
0	-	$\Psi_{i,j} \leftarrow \Psi_{i,j} + \tilde{w}\,\tilde{r}$
1	-	$\tilde{w} \leftarrow [w_3\ w_4\ 0]$
1	-	$\tilde{r} \leftarrow \mathbf{0}_{D\times 1}$
1	1	$\tilde{r} \leftarrow \tilde{r} + u_{1,i}\,[u_{4,j}\ u_{5,j}\ u_{6,j}]'$
1	2	$\tilde{r} \leftarrow \tilde{r} + u_{2,i}\,[u_{5,j}\ u_{6,j}\ u_{7,j}]'$
1	3	$\tilde{r} \leftarrow \tilde{r} + u_{3,i}\,[u_{6,j}\ u_{7,j}\ 0]'$
1	4	$\tilde{r} \leftarrow \tilde{r} + u_{4,i}\,[u_{7,j}\ 0\ 0]'$
1	-	$\Psi_{i,j} \leftarrow \Psi_{i,j} + \tilde{w}\,\tilde{r}$

- This subroutine contains absolutely no conditional statements. We consider it beneficial to avoid such statements because they may fork the data path and lead to redundant resource consumption on reconfigurable hardware. The simplicity brought by the conditional-free control logic may also reduce the workload of implementation.
- Many arithmetic operations in the algorithm can be executed in parallel. We can observe from Line 4 in Algorithm 4 that $\hat{r}_{g,c,i,j}$ is computed by taking the sum of the results of vector scalar products. As the components in the vector are independent, the addition and multiplication operations on all the c components of the vector can take place in parallel.
- There is a considerable data reuse pattern behind the subroutine. In the computation of $\hat{r}_{g,c,i,j}$, when $k = k_0$, the accessed data elements are $u_{k_0,i}$ and $u_{k_0+gc,j}$ $\ldots u_{k_0+gc+c-1,j}$; when $k = k_0 + 1$, the accessed data elements are $u_{k_0+1,i}$ and $u_{k_0+gc+1,j}$ $\ldots u_{k_0+gc+c,j}$. All data elements accessed when $k = k_0 + 1$, except $u_{k_0+1,i}$ and $u_{k_0+gc+c,j}$, have been previously accessed when $k = k_0$. We shall design a caching scheme to take advantage of this data reuse pattern. With a perfect caching scheme, only two data elements need to be retrieved from the main memory, and the remaining data elements can be accessed from the cache memory. This is the essential idea to crack the memory bandwidth bottleneck. We will discuss this issue in detail in Section 4.

Admittedly, the time complexity of the pipeline-friendly algorithm is still $O(D^2HT)$ which is not different from the straightforward algorithm. Moreover, the pipeline-friendly design may incur redundant computations when c is not a factor of $H + 1$. However, the pipeline-friendly properties enable us to achieve significant acceleration in practice. The underlying reasons and experimental results will be discussed in Section 4 and in Section 6 respectively.

4 An Initial Hardware Architecture

In this section, we discuss an initial pipelined architecture for the subroutine PASS(g, c, i, j). This architecture is originally proposed in [4]. We first present the hardware architecture and explain its interactions with the host computer. Then we discuss a simple theoretical model for its execution time.

4.1 Hardware Architecture

A simple elementary computational unit, which we call a *bead*, is described in Fig. 1. A bead handles the computation for one component of $\tilde{r}$ in Line 4 in Algorithm 4.

Our proposed architecture is constructed by linking up c beads in a way shown in Fig. 2. More specifically, we use a single buffer register to store a single data element from its input stream. For ease of discussion, we call this buffer register the *broadcasting buffer* hereafter. At the end of each cycle, a data element from the input stream is loaded into the buffer. This broadcasting buffer serves as one input to all the c beads.

Although the broadcasting buffer has a high fan-out, such fan-outs can often be removed automatically by hardware compilers. In addition, techniques such as data pipelining [24] can be used to eliminate fan-outs. The application of such techniques to optimise our design will be reported in a future publication.

We customise on-chip fast memory in the reconfigurable hardware device, e.g. block RAMs, to become a first-in-first-out (FIFO) buffer with c storage units. This buffer is

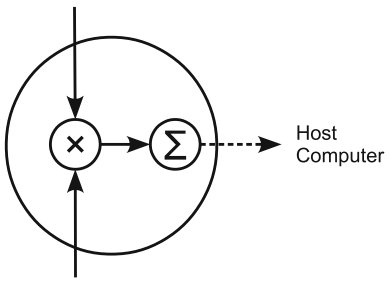

Figure 1 The structure of a bead: a bead takes two numbers as its inputs. It multiplies the two inputs and accumulates the product.

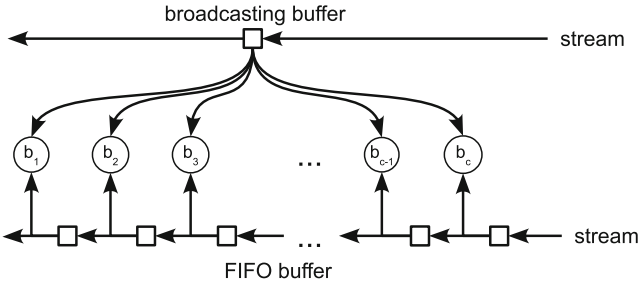

Figure 2 The proposed architecture: each bead b_i takes one input from the broadcasting buffer and another input from the FIFO buffer.

used to store c consecutive data elements of its input stream. At the end of each cycle, every element of the FIFO buffer accepts data from its right neighbour. The previous leftmost element is moved out of the buffer and discarded. The input stream supplies data to the rightmost element in the FIFO buffer. Each storage unit in the FIFO buffer contributes another input to a bead.

Each time when Algorithm 4 is invoked, the data streams $u_{1,i} \ldots u_{T-gc,i}$ and $u_{gc+1,j} \ldots u_{T,j}$ are streamed into the architecture for the computation of $\tilde{r}$. This streaming process can be divided into the following three stages.

1. Initialisation stage: all registers in all beads are reset to zero. The broadcasting buffer is loaded with the first element of the stream $u_{1,i} \ldots u_{T-gc,i}$. The FIFO buffer is filled with the first c elements of the stream $u_{gc+1,j} \ldots u_{T,j}$. In other words, at the end of the initialisation stage, the broadcasting buffer contains $u_{1,i}$ and the FIFO buffer contains $u_{gc+1,j} \ldots u_{gc+c,j}$.
2. Pipeline processing stage: in every cycle, each bead accumulates the product of its inputs. Then both the broadcasting buffer and the FIFO buffer load the next data element from their corresponding input stream. This process runs for $(T - gc)$ cycles before termination.
3. Result summation stage: each bead reports its accumulation result to the host computer. The accumulation result of the q-th bead is the q-th component of $\tilde{r}$. The host computer then revises $\Psi_{i,j}$ according to $\tilde{r}$.

To optimise the overall performance or energy efficiency, one may replace the host computer by a hardware accelerator for finite impulse response filter (FIR). We do not apply this optimisation in this study because the host computer is an indispensable part of our acceleration platform.

The hardware design for Algorithm 4 is similar to some systolic implementations for matrix vector multiplication with regular word-level and bit-level architectures [25]. While techniques such as polyhedral analysis, data pipelining and tiling have been used in deriving such implementations [24], the focus of this paper is to develop our designs

based on mathematical treatment from first principles rather than making use of derived results.

We can observe from the hardware design that the pipeline-friendly features behind our algorithm are fully exploited. The FIFO buffer provides a perfect caching mechanism to take advantage of the data reuse pattern discussed in Section 3. In each cycle, only two data elements are fetched from the input stream, and all the remaining elements are obtained from the FIFO buffer. In other words, the memory bandwidth requirement is both small and constant. When more beads are linked up in the system, the bandwidth requirement remains unchanged, which suggests that the performance may scale up well with the amount of on-chip logical resources without being limited by the memory bottleneck. Furthermore, the architecture is highly modularised. The major business modules of the architecture are the beads and the two buffers. These modules are structurally uncomplicated and can be tested individually, which reduces the potential effort in implementation and debugging.

4.2 Performance Estimation

We are interested in processing long time series in this study, hence the pipeline processing stage would be the most time-consuming one. To compute each entry of Ψ, $\textsc{Pass}(g, c, i, j)$ has to be invoked for $(G + 1)$ times. The number of cycles spent in the g-th invocation is $(T - gc)$. Let F be the clock frequency of the reconfigurable device. The total execution time of this stage in all invocations is

$$\mathfrak{T}_P = \frac{1}{F}\sum_{g=0}^{G}(T - gc) = \frac{(G+1)(2T - Gc)}{2F} \tag{24}$$

For ease of discussion, we will call this time *the theoretical pipeline processing time* hereafter. Let $\mathfrak{T}_\epsilon$ be the total execution time that is not spent on the pipeline processing stage. Then the total computation time is

$$\mathfrak{T} = \mathfrak{T}_P + \mathfrak{T}_\epsilon \tag{25}$$

We do not attempt to model $\mathfrak{T}_\epsilon$ since we consider its value both unpredictable and negligible. $\mathfrak{T}_\epsilon$ is related to the configuration and execution status of the acceleration platform, and this quantity is unlikely to be significant compared to the pipeline processing time, especially when the time series is long.

5 Optimised Hardware Architectures

In this paper, we proposes two alternative hardware designs in addition to the initial one described in the last section. The major objective of the designs is to improve the worst-case performance of the initial architecture.

5.1 Architecture with Multiple FIFO Buffers

The initial hardware architecture proposed in the previous section achieves its best performance when the beads deployed along the pipeline can be fully used during the execution. This requires that $H + 1$ can be exactly divided by the number of beads c. If not, some beads will be idle in certain iterations. We call the maximum number of idle beads the *maximum idleness*. For an implementation of the architecture with c beads, the maximum idleness is $c - 1$.

We reduce the maximum idleness of the engine so that the worst-case performance can be improved. This is particularly beneficial for medium sized data. As we have discussed in the background section, if the length of the data is small, an appropriate lag truncation parameter H could be far less than c. When c is large, the computation can be done by streaming the data into the engine once but there could be many idle beads in this case.

Our proposed architecture is constructed by linking up c beads in a different way from the architecture discussed in the previous section. More specifically, we use a single buffer register to store a single data element from its input stream. However, we use multiple FIFO buffers to collect information about multiple entries of matrix Ψ.

Similar to the initial architecture, a FIFO buffer that caches c data elements is associated with c beads, and each data cell offers an input to a bead. Therefore, if we deploy k FIFO buffers in this architecture, the total number of beads is kc. The broadcasting buffer provides another input to all the kc beads.

This architecture works in a similar way to the one we have discussed in the previous section. The streaming process can still be divided into three stages.

1. Initialisation stage: all registers in all beads are reset to zero. The broadcasting buffer is loaded with the first element of its input stream. The k FIFO buffers are filled with the first c elements of their corresponding streams.
2. Pipeline processing stage: in every cycle, each bead accumulates the product of its inputs. Then both the broadcasting buffer and each FIFO buffer loads the next data element from their corresponding input streams. Suppose we build the architecture with k FIFO buffers, a total number of $(k + 1)$ data elements are loaded from the memory interface. This is different from the initial architecture where only two data elements are loaded in each cycle.
3. Result summation stage: similar to the initial architecture, the vector r for each FIFO buffer can be collected

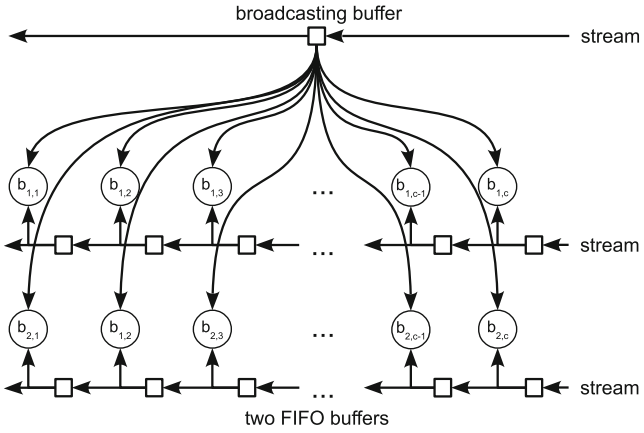

Figure 3 The multi-FIFO architecture: each bead b_i takes one input from the broadcasting buffer and another input from the FIFO buffer.

from the accumulator in a bead. The host computer then revises $\Psi_{i,j}$ according to $\tilde{r}$ for each component j. In each run of the system, k entries in Ψ in the same row can be revised.

After streaming the data into the engine for $G+1$ times, a total number of k entries in the same row in Ψ can be obtained. For instance, Fig. 3 shows a configuration with two FIFO buffers. Suppose that we estimate a 4×4 long-run covariance matrix using this engine. We may link the broadcasting buffer to component 1 of the time series and the two FIFO buffers to component 3 and 4. Then after the computation, we may obtain $\Psi_{1,3}$ and $\Psi_{1,4}$.

The broadcasting buffer in the multi-FIFO architecture may still have a high fan-out like the initial architecture. However, given a reconfigurable device, the fan-out of the multi-FIFO architecture is unlikely to be larger than the fan-out of the initial architecture. This is because the number of outputs of the broadcasting buffer equals to the number of beads, and the number of beads is decided by the amount of hardware resources.

5.2 Architecture with Generalised Beads

Another way to improve the worst-case performance is to use generalised beads. Similar to an ordinary bead, a generalised bead computes one entry in $\tilde{r}$ in Line 4 in Algorithm 4. However we allow the products of multiple pairs of data elements to be accumulated simultaneously. The structure of a generalised bead with two pairs of inputs is shown in Fig. 4.

To compute an entry of Ψ. Suppose we use generalised beads with m pairs of inputs. To link a number of c generalised beads up, we use one broadcasting buffer and one FIFO buffer. The broadcasting buffer is designed to store m data elements. The FIFO buffer is divided into c segments, each of which stores m data elements. The m data elements

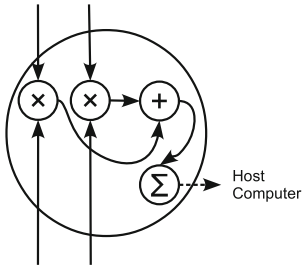

Figure 4 The structure of a generalised bead with two pairs of inputs: a bead takes the product of each pair of inputs. Then the bead sums up the products and accumulates the results.

from the broadcasting buffer and the m values from a segment of the FIFO buffer serve as m pairs of inputs of a generalised bead. For instance, the architecture with $m = 2$ is shown in Fig. 5.

The operating principle of this architecture is also similar to the initial one. The streaming process can still be divided into three stages. Suppose we build the architecture using generalised beads with m pairs of inputs, the three stages can be described as follows.

1. Initialisation stage: all registers in all beads are reset to zero. The broadcasting buffer is loaded with the first c element of its input stream. The FIFO buffer is filled with the first mc elements of their corresponding streams. These operations are different from those of the multi-FIFO architecture where only one data element is loaded for the broadcasting buffer and k for the FIFO buffer.
2. Pipeline processing stage: in every cycle, each bead accumulates the products of all pairs of its inputs. Then both the broadcasting buffer and the FIFO buffer load the next m data elements from their corresponding input streams. A total number of $2m$ data elements are loaded from the memory interface.
3. Result summation stage: each bead reports its accumulation result to the host computer. The accumulation result of the q-th bead is the q-th entry of $\tilde{r}$. The host computer then revises $\Psi_{i,j}$ according to $\tilde{r}$.

5.3 Performance Estimation and Discussion

We use the performance estimation method discussed in the previous section to investigate the theoretical pipeline processing times of the above two architectures.

- The architecture with k FIFO buffers: the performance depends on which entries we need to estimate. The architecture reaches its highest performance when the number of entries we need in each row can be exactly divided by the number FIFO buffers k. In this case, the

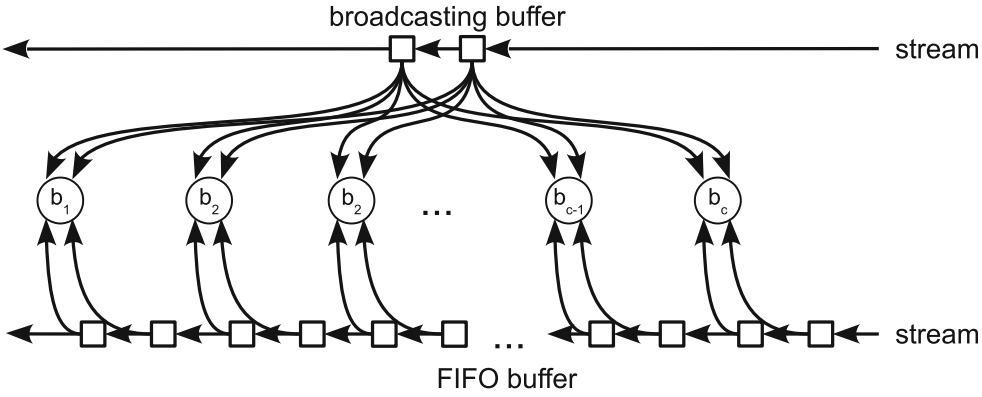

Figure 5 The architecture based on generalised beads: each bead b_i takes two inputs from the broadcasting buffer and the other two inputs from the FIFO buffer.

theoretical pipeline processing time can be calculated by

$$\mathfrak{T}_P = \frac{(G+1)(2T - Gc)}{2kF} \tag{26}$$

The architecture demonstrates its worst performance when only one entry is needed in each row. In this case, $(k-1)$ out of k FIFO buffers do not work in the computation. Therefore the theoretical pipeline processing time is calculated in the same way as the initial architecture.

- The architecture with generalised beads with m pairs of inputs: the theoretical pipeline processing time depends on the number of pairs of inputs of each bead m. Since the engine processes m pairs of data elements, the time can be calculated by

$$\mathfrak{T}_P = \frac{(G+1)(2T - Gc)}{2mF} \tag{27}$$

The performance models are helpful in deciding a design for a given problem. In particular, we list three considerations when making the decision.

First, in general, the maximum performance of the multi-FIFO design is better than that of the generalised-beads-based one. The value k in Eq. 26 can usually be assigned with a larger number than m in Eq. 27. This is because when increasing m by one in the generalised-beads-based design, a broadcasting buffer with the c outputs is created. It takes additional hardware resources to resolve the high fan-out of the newly created broadcasting buffer. In contrast, the multi-FIFO design does not suffer this problem.

Second, in the case where only a small number of entries in each row of the long-run covariance matrix are required, the generalised-beads-based design is faster. This is because if the number of required entries in a row, denoted by k', is fewer than k, then $(k-k')$ FIFO buffers are idle during the computation. Therefore, the performance shown in Eq. 26 cannot be achieved. For example, given a 10-dimensional time series representing 10 stocks, the size of the complete long-run covariance matrix S is 10×10. If we are interested in the role of the first stock in the market, we only need to evaluate $S_{1,i}$ and $S_{i,1}$ for all $i \in [1..10]$. In other words, from the second row to the tenth row, we only need to calculate one entry in each row. In this case, the generalised-beads-based design is preferred.

Third, the initial design may be good enough for some particular lag truncation parameters. All the three designs process the lags in batches. Equations 17, 24, 26 and 27 suggest that a design achieves its best performance if the lag truncation parameter H is set to such a value that $(H+1)$ can be exactly divided by the batch size. Therefore, when $(H+1)$ can only be exactly divided by the batch size of the initial design rather than the other two designs, the initial design is very likely to be the best choice.

6 Experimental Evaluation

We run a series of experiments to evaluate the performance of our proposed architectures. In this section, we first present the general experimental settings, and then discuss the results.

6.1 Experimental Settings

The experimental settings in this study are similar to the one in [4]. All the hardware architectures in our experiments are described in the MaxJ language and compiled with Maxeler MaxCompiler. The acceleration system is a Maxeler MAX3 acceleration card equipped with a Xilinx Virtex-6 V6-SX475T FPGA. The acceleration card communicates with a host computer via a PCI Express interface. We build the following three experimental implementation using the platform (i) the initial architecture with 384 beads; (ii) the multi-FIFO architecture with 5 FIFO buffers, each of which is associated with 77 beads; (iii) the generalised-beads-based design with 128 generalised beads, each of which takes 3 pairs of inputs. We set the clock frequency to 100MHz for all the three systems.

We also build a CPU-based system by implementing the straightforward HAC estimation algorithm on the CPU platform in a server with 12 Intel Xeon X5650 cores running at 2.67GHz. The experimental code is written in the C programming language with the OpenMP library, and compiled with Intel C compiler with the highest compilation optimisation. To make a fair comparison, the IEEE single precision floating point numbers are used exclusively in both the hardware and software implementations.

In order to simulate the computation of different types of data, for each data set we select 12 different values of γ in the range $0.25 \leq \gamma \leq 3.00$ which is slightly wilder than the range investigated in [7].

The computational efforts for all entries of the long-run covariance matrix are identical. Therefore we describe the performance in terms of the computation time of a single entry. Following the experiment scheme in [4] and [7], two time series are generated using a vector autoregression (VAR) model [5]. The lengths of the two time series are respectively 10^6 and 10^8.

6.2 Performance Results

Experimental results are shown in Fig. 6. It is clear that the speedup of the FPGA-based system is significant, especially for long time series where $T = 10^8$. The best speedup is obtained with the multi-FIFO architecture. When $T = 10^8$ and $H = 1045$, it is 113 times faster than a single CPU core, and 12 times faster than twelve cores. The architecture based on generalised beads demonstrates lower performance than the multi-FIFO architecture, but this architecture delivers consistent performance regardless which entries in the long-run covariance matrix are queried.

The execution times of all systems increase as the lag truncation parameter grows. However, the growth patterns of the FPGA based systems are significantly different from those of the CPU-based systems. More specifically, the execution times of the two CPU-based systems grow linearly with different slope. This linear growth can be explained by the time complexity of the algorithm. The execution times of the FPGA-based systems increase like stairs. This is because the FPGA-based systems handle the computation for different lags in batches.

Due to the difference in the growth pattern in execution time, it is not surprising that the speedup of the FPGA-based system over the CPU-based one appears a zig-zag pattern in a periodical manner, as shown in Fig. 6. This is because the speedup falls down due to the idleness of beads, which we have discussed in the previous section.

We may also obtain two further observations from this zig-zag pattern. One observation is that the amplitude of the zig-zag pattern shrinks as the lag truncation parameter grows, since the redundant computation times become less significant compared to their corresponding total execution

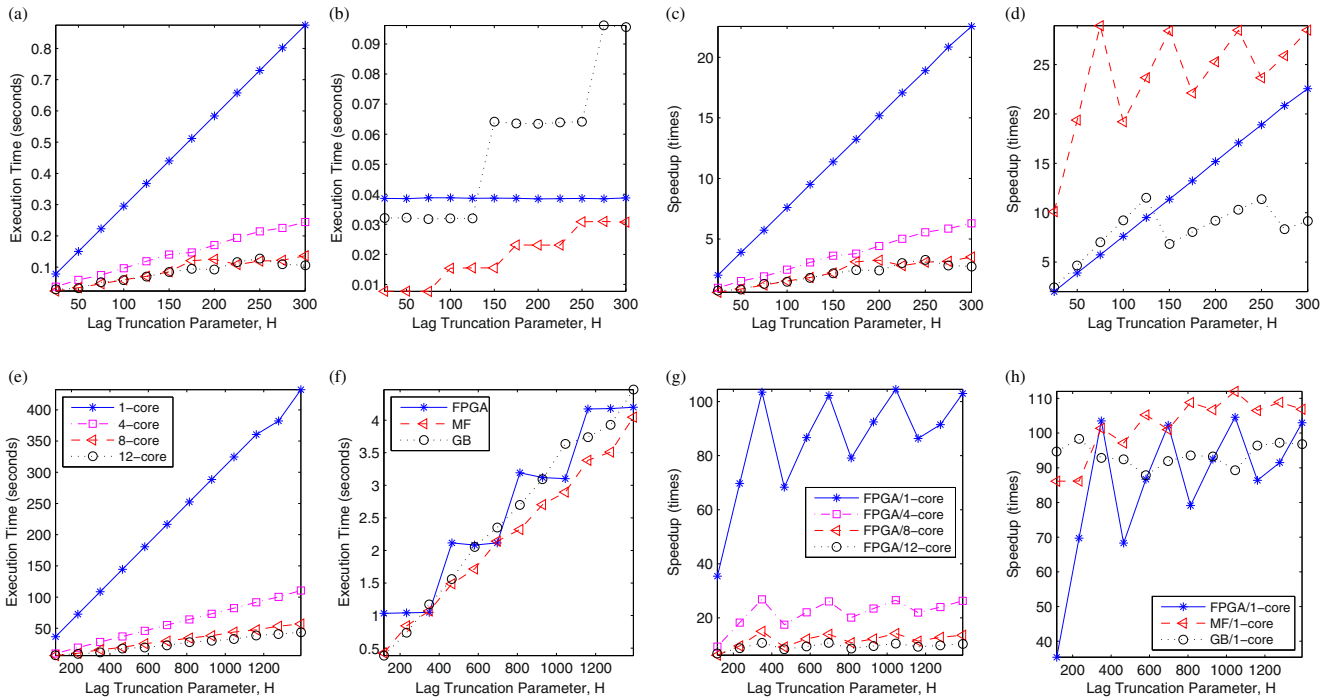

Figure 6 Results on performance: the first two columns of figures are performance results from the CPU-based system and the FPGA-based system respectively. The third column and the fourth columns record the speedups of the FPGA-based systems over the CPU-based ones. FPGA, MF, GB correspond respectively to the initial architecture, the multi-FIFO architecture and the generalised-beads-based design.

times. The other observation is that the shaking amplitudes of the two optimised systems are significantly smaller than the initial one because the maximum idleness is reduced. As a result, the performance is less dependent on the lag truncation parameter. In other words, the optimisations enhance the reliability of the system by improving the worst-case performance.

7 Conclusion

This paper presents a reconfigurable acceleration solution to heteroskedasticity and autocorrelation consistent (HAC) covariance matrix estimation for multivariate time series.

Rather than providing a hardware design for an existing HAC estimation algorithm, we use an algorithm designed exclusively for hardware implementation. This algorithm exploits the capabilities of a reconfigurable computing platform and avoids limitations like the memory bottleneck. Based on our algorithm, we present three hardware architectures namely the initial architecture, the multi-FIFO architecture and the generalised-beads-based design. We analyse their performance using both theoretical and empirical approaches. An experimental implementation of the multi-FIFO architecture using a Virtex-6 V6-SX475T FPGA achieves up to 113 times speedup over a single-core CPU, and up to 12 times speedup over an 12-core CPU. The implementation of the generalised-beads-based design shows slightly lower performance than the multi-FIFO architecture, but its performance does not depend on which long-run covariance matrix entries are queried by the user. In general, the two optimised architectures effectively improve the worst-case performance of the initial architecture.

This work shows the potential of reconfigurable computing for time series processing. Future work includes developing hardware accelerators for other time series processing techniques, such as regression analysis, forecasting and knowledge discovery.

Acknowledgments The authors would like to thank the anonymous reviewers for their constructive comments. This work is supported in part by the China Scholarship Council, by the European Union Seventh Framework Programme under grant agreement number 257906, 287804 and 318521, by UK EPSRC, by Maxeler University Programme, and by Xilinx.

References

1. Jegadeesh, N., & Titman, S. (1993). Returns to buying winners and selling losers: Implications for stock market efficiency. *The Journal of Finance*, *48*(1), 65–91.
2. Bollerslev, T., Tauchen, G., Zhou, H. (2009). Expected stock returns and variance risk premia. *Review of Financial Studies*, *22*(11), 4463–4492.
3. Bekaert, G., Harvey, C., Lundblad, C., Siegel, S. (2011). What segments equity markets *Review of Financial Studies*, *24*(12), 3841–3890.
4. Guo, C., & Luk, W. (2013). Accelerating HAC estimation for multivariate time series. In *International Conference on Application-specific Systems*: Architectures and Processors.
5. Hamilton, J.D. (1994). In *Time series analysis*: Cambridge University Press.
6. Newey, W.K., & West, K.D. (1987). A simple, positive semi-definite, heteroskedasticity and autocorrelation consistent covariance matrix. *Econometrica: Journal of the Econometric Society*, *55*, 703–708.
7. Newey, W.K., & West, K.D. (1994). Automatic lag selection in covariance matrix estimation. *Review of Economic Studies*, *61*, 631–653.
8. Andrews, D. (1991). Heteroskedasticity and autocorrelation consistent covariance matrix estimation. *Econometrica: Journal of the Econometric Society*, *59*, 817–858.
9. Sart, D., Mueen, A., Najjar, W., Keogh, E., Niennattrakul, V. (2010). Accelerating dynamic time warping subsequence search with GPUs and FPGAs. In *International Conference on Data Mining*, (pp. 1001–1006).
10. Wang, Z., Huang, S., Wang, L., Li, H., Wang, Y., Yang, H. (2013). Accelerating subsequence similarity search based on dynamic time warping distance with FPGA. In *International Symposium on Field-Programmable Gate Arrays*.
11. Preis, T., Virnau, P., Paul, W., Schneider, J.J. (2009). Accelerated fluctuation analysis by graphic cards and complex pattern formation in financial markets. *New Journal of Physics*, *11*(9), 093024.
12. Gembris, D., Neeb, M., Gipp, M., Kugel, A., Männer, R. (2011). Correlation analysis on GPU systems using NVIDIA's CUDA. *Journal of real-time image processing*, *6*(4), 275–280.
13. Baker, Z., & Prasanna, V. (2005). Efficient hardware data mining with the apriori algorithm on FPGAs. In *International Symposium on Field-Programmable Custom Computing Machines*, (pp. 3–12).
14. Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. In *International Conference on Very Large Databases (VLDB)*, (pp. 487–499).
15. MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *Berkeley Symposium on Mathematical Statistics and Probability*, (pp. 281–297).
16. Quinlan, J.R. (Mar. 1986). Induction of decision trees. *Machine Learning*, *1*(1), 81–106.
17. Saegusa, T., & Maruyama, T. (2006). An FPGA implementation of K-means clustering for color images based on kd-tree. In *International Conference on Field Programmable Logic and Applications*, (pp. 1–6).
18. Narayanan, R., Honbo, D., Memik, G., Choudhary, A., Zambreno, J. (2007). An FPGA implementation of decision tree classification. In *Conference on Design, Automation and Test in Europe*, (pp. 1–6).
19. Moerland, P., & Fiesler, E. (1997). In *Neural network adaptations to hardware implementations, The Idiap Research Institute*. Switzerland: Tech. Rep.
20. Guo, C., Fu, H., Luk, W. (2012). A fully-pipelined expectation-maximization engine for Gaussian mixture models. In *International Conference on Field-Programmable Technology*.

21. Zeileis, A. (2004). In *Econometric computing with HC and HAC covariance matrix estimators*: Distributed with the sandwich R package.
22. Cottrell, A., & Lucchetti, R. (2012). In *Gretl user's guide*: Distributed with the Gretl library.
23. Lin, C.Y., So, H.K., Leong, P.H.W. (2011). A model for matrix multiplication performance on FPGAs. In *International Conference on Field Programmable Logic and Applications*, (pp. 305–310).
24. Jacob, A.C., Buhler, J.D., Chamberlain, R.D. (2010). Rapid RNA folding: analysis and acceleration of the zuker recurrence. In *International Symposium on Field-Programmable Custom Computing Machines*, (pp. 87–94).
25. Urquhart, R., & Wood, D. (1984). Systolic matrix and vector multiplication methods for signal processing. *IEE Proceedings*, *131*(6), 623–631.

Ce Guo received the M.Sc. degree in artificial intelligence and is pursuing a Ph.D. degree in the Department of Computing, Imperial College London, London, U.K. His current research interests include reconfigurable accelerators and accelerator-friendly algorithms for predictive modelling, machine learning and financial computing.

Wayne Luk received the M.A., M.Sc., and D.Phil. degrees in engineering and computing science from the University of Oxford, Oxford, U.K. He is Professor of Computer Engineering with Imperial College London, London, U.K. He was a Visiting Professor with Stanford University, Stanford, CA, USA. His current research interests include theory and practice of customizing hardware and software for specific application domains, such as multimedia, networking, and finance.