Accelerating HAC Estimation for Multivariate Time Series

Ce Guo
Department of Computing
Imperial College London
United Kingdom
ce.guo10@imperial.ac.uk

Wayne Luk
Department of Computing
Imperial College London
United Kingdom
w.luk@imperial.ac.uk

***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 novel 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 then develop a fully-pipelined hardware architecture based on the proposed algorithm. This architecture is shown to be efficient and scalable from both theoretical and empirical perspectives. Experimental results show that an FPGA-based implementation of the proposed architecture is up to 111 times faster than an optimised CPU implementation with one core, and 14 times faster than a CPU with eight cores.**

I. 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. To some extent, the long-run covariance matrix plays a similar role as the ordinary covariance matrix of non-temporal multivariate data. However, HAC estimation involving a long-run covariance matrix is different from estimation involving an ordinary covariance matrix from non-temporal data, since HAC estimation considers the unique features of time series data such as serial correlation.

Today, HAC estimation becomes one of the standard methods 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] [2] [3], HAC estimation is used to quantify the risks of trading strategies.

One drawback of HAC estimation technique is the computational cost. This drawback has become increasingly serious in recent years because the lengths and dimensions of real-world time series have been growing continuously. Data analysts take samples at increasingly short time interval to capture subtle 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.

This paper presents a highly efficient solution to HAC estimation. A highlight of our solution is that we do not design hardware based on existing software-based HAC estimation algorithms. Instead, we derive a novel pipeline-friendly algorithm from a mathematical specification. We then map our algorithm to a hardware architecture, making effective use of on-chip resources without exceeding the limited memory bandwidth. This solution is both fast and scalable, and therefore particularly useful for long and high-dimensional time series data.

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 [4] [5], our work is the first to apply reconfigurable computing to time series analysis. Our key contributions are as follows.

- We derive a pipeline-friendly HAC estimation algorithm by performing mathematical transformations. This algorithm exploits the computational power of the hardware platform with conditional-free logic and parallelised arithmetic. Moreover, it avoids memory bottlenecks with a powerful data reuse scheme.
- We map the proposed algorithm to a fully-pipelined hardware architecture by customising on-chip memory to be a first-in-first-out buffer. This architecture takes full advantage of the pipeline-friendly features of our proposed algorithm in an elegant way.
- We implement our design in a commercial FPGA acceleration platform. With quantitative analysis and experimental results, we demonstrate the performance and scalability of our hardware design, which can be up to 14 times faster than an 8-core CPU implementation.

The rest of the paper is organised as follows. Section II briefly describes HAC estimation problem and review re-

configurable computing solutions for statistical data analysis. Section III presents our proposed pipeline-friendly HAC estimation algorithm and discusses its hardware-oriented features. Section IV describes a hardware design that maps our algorithm to a fully-pipelined architecture. Section IV provides experimental results about an implementation of our hardware design, and explains experimental observations. Section VI provides a brief conclusion of our work.

II. 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.

A. 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

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

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

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

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

B. 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 [6].

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 Equation 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* [7] 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; $k(\cdot)$ is a real-valued kernel function; $\hat{\Omega}_h$ is the estimate of the autocovariance matrix with lag h which can be computed by

$$\hat{\Omega}_h = \frac{1}{T} \sum_{t=h+1}^{T} (y_t - \hat{\mu})(y_{t-h} - \hat{\mu})' \tag{8}$$

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. Some discussions about kernel functions can be found in [7] and [8]. Some variances of the Newey-West estimator can be found in [6] and [8].

C. 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 GPUs and FPGAs are proposed to process time series data. Sart et. al [4] propose both GPU-based and FPGA-based solutions to accelerate dynamic time wrapping (DTW) for sequential data. Wang et al. [5] develop a hardware engine for DTW-based sequence searching in time series. However, the aims of these studies are to solve sequence matching problems, and they do not analyse the underlying patterns of time series data from a statistical perspective.

Preis et. al [9] 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 [10] 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 [11] mapped the Apriori algorithm [12] into an FPGA-based acceleration device for improved

efficiency. Similar to the Apriori engine, hardware acceleration solutions for k-means clustering [13] and decision tree classification [14] are presented in [15] and [16] 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.

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. [17] 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.

III. Pipeline-Friendly HAC Estimation Algorithm

In this section, we propose a pipeline-friendly algorithm, which is designed with considerations of the features of reconfigurable hardware. We first explain why we do not map the existing algorithm to hardware. Then we show how the expression of $\hat{S}$ (Equation 7) can be transformed to eliminate conditionals, to parallelise arithmetic, and to promote data reuse. Finally, we present our new estimation algorithm.

A. Analysis of the Straightforward Algorithm

It is not difficult to design an algorithm following Equation 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* hereafter 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 [18] and the GNU regression, econometrics and time-series library [19].

Algorithm 1 Straightforward HAC Estimation Algorithm

```
1: Ŝ ← AutoCov(0)
2: for h ∈ [1..H] do
3:     Ω̂ ← AutoCov(h)
4:     Ŝ ← Ŝ + k(h/(H+1))(Ω̂ + Ω̂′)
5: return Ŝ
```

Taking arithmetic operations as basic operations, the time complexity of the algorithm is $\mathrm{O}(D^2HT)$, 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 [20]. As a consequence, the algorithm may become extremely computational demanding with long and high-dimensional time series.

Algorithm 2 AutoCov(h)

```
1: Ω̂ ← 0_{D×D}
2: for t ∈ [(h+1)..T] do
3:     for i ∈ [1..D] do
4:         for j ∈ [1..D] do
5:             Ω̂_{i,j} ← Ω̂_{i,j} + (y_{t,i} − μ̂_i)(y_{t−h,j} − μ̂_j)
6: return Ω̂
```

The most time-consuming part of the algorithm is the computation of an autocovariance matrix, 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 from memory bottleneck [21]. So it is critical to find optimisations of the algorithm that would avoid such bottleneck. In order to make a fundamental difference from CPUs in performance, it is unwise to merely map the straightforward algorithm to a reconfigurable computing platform.

B. A Novel Derivation for $\hat{S}$

We build the mathematical foundation of our pipeline-friendly algorithm by deriving a novel expression of $\hat{S}$ (Equation 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

$$u_t = y_t - \hat{\mu} \tag{9}$$

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, Equation 8 can be rewritten as

$$\hat{\Omega}_h = \frac{1}{T} \sum_{t=h+1}^{T} u_t u'_{t-h} \tag{10}$$

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

$$\Omega_0 = \Omega'_0 \tag{11}$$

To unify the computational pattern in Equation 7, we may merge $\hat{\Omega}_0$ to the weighted sum by setting

$$w_h = \begin{cases} \frac{1}{2} & \text{if } h = 0 \\ k(\frac{h}{H+1}) & \text{if } 0 < h \leq H \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

These weights can be regarded as a general form of the kernel function values. They are not related to data and can be precomputed and stored. From Equation 12, the expression of $\hat{S}$ can be simplified as

$$\hat{S} = \sum_{h=0}^{H} w_h(\hat{\Omega}_h + \hat{\Omega}'_h) \tag{13}$$

The autocovariance matrix Ω_h can be expanded using Equation 10 and the expression of $\hat{S}$ can be rewritten as

$$\hat{S} = \frac{1}{T}(\Psi + \Psi') \tag{14}$$

where

$$\Psi = \sum_{h=0}^{H} w_h \sum_{t=h+1}^{T} u_t u'_{t-h} \tag{15}$$

Therefore once Ψ is obtained, $\hat{S}$ can be computed effortlessly. 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

$$G = \lceil \frac{H+1}{c} \rceil - 1 \tag{16}$$

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

$$\Psi = \sum_{g=0}^{G} \sum_{h=gc}^{gc+c-1} w_h \sum_{t=h+1}^{T} u_t u'_{t-h} \tag{17}$$

The function of the parameter c and the quantity G will be illustrated later. If c is a factor of $(H+1)$ then Equation 17 obversely holds. If not, some terms with $h > H$ will be calculated, but Equation 17 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

$$\Psi_{i,j} = \sum_{g=0}^{G} \tilde{w}_{g,c}\ \tilde{r}_{g,c,i,j} \tag{18}$$

where

$$\tilde{w}_{g,c} = [w_{gc} \quad w_{gc+1} \quad \dots \quad w_{gc+c-1}] \tag{19}$$

$$\tilde{r}_{g,c,i,j} = \begin{bmatrix} \sum_{t=gc+1}^{T} u_{t,j} \cdot u_{t-gc,i} \\ \sum_{t=gc+2}^{T} u_{t,j} \cdot u_{t-(gc+1),i} \\ \vdots \\ \sum_{t=gc+c}^{T} u_{t,j} \cdot u_{t-(gc+c-1),i} \end{bmatrix} \tag{20}$$

The vector $\tilde{w}_{g,c}$ can be constructed using the weight values precomputed by Equation 12. We only need to focus on the computation of $\tilde{r}_{g,c,i,j}$. Aligning the lower bounds of the summation operators in Equation 20, we have

$$\tilde{r}_{g,c,i,j} = \begin{bmatrix} \sum_{k=1}^{T-gc} u_{k+gc,j} \cdot u_{k,i} \\ \sum_{k=1}^{T-gc-1} u_{k+gc+1,j} \cdot u_{k,i} \\ \vdots \\ \sum_{k=1}^{T-gc-c+1} u_{k+gc+c-1,j} \cdot u_{k,i} \end{bmatrix} \tag{21}$$

We have defined that $u_t = 0$ when $t > T$. Hence we can set the upper bounds of all summation operations in Equation 21 to $(T - gc)$. Then the expression of $\tilde{r}_{g,c,i,j}$ can be further simplified:

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

C. Pipeline-Friendly HAC Estimation Algorithm

We shall 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 Equation 18. Once all entries of Ψ are obtained, $\hat{S}$ can be computed by Equation 13. 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 Equation 22 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 Equation 18.

Algorithm 3 Pipeline-Friendly HAC Estimation

1: **for** $(i, j) \in [1..D] \times [1..D]$ **do**
2: $\quad \Psi_{i,j} = 0$
3: $\quad$ **for** $g \in [0..G]$ **do**
4: $\quad\quad \tilde{w} \leftarrow [w_{gc} \quad w_{gc+1} \quad \dots \quad w_{gc+c-1}]$
5: $\quad\quad \tilde{r} \leftarrow$ 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** $\tilde{r}$

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.

First, 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.

Second, many arithmetic operations in the algorithm can be executed in parallel. We can observe from Line 3 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.

Third, 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 IV.

Admittedly, the time complexity of the pipeline-friendly algorithm is still $\mathrm{O}(D^2HT)$ 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 IV and in Section V respectively.

IV. Hardware Design

In this section, we develop a pipelined architecture for the subroutine $\textsc{Pass}(g, c, i, j)$ described in the previous section. We first present the hardware architecture and explain its interactions with the host computer. Then we propose a simple theoretical model for its execution time.

A. 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 3 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 [22] can be used to eliminate fan-outs. The application of such techniques to optimise our design will be reported in a future publication.

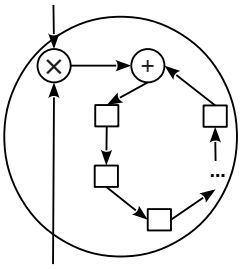


Fig. 1. The structure of a bead: a bead takes two numbers as its inputs. It multiplies the two inputs and accumulates the product using a chain of registers which are shown as boxes in the figure.

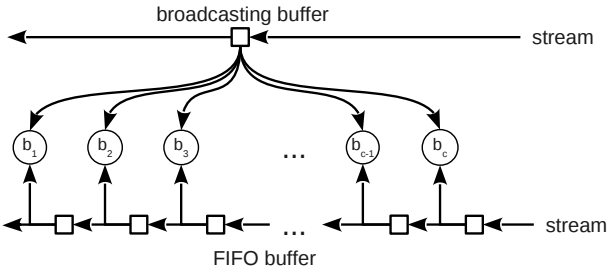


Fig. 2. The proposed architecture: each bead b_i takes one input from the broadcasting buffer and another input from the FIFO buffer.

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 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. (i) 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 $u_{gc+1,j} \ldots u_{gc+c,j}$. (ii) 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. (iii) Result summation stage: each bead reports its register values to the host computer. The sum of the register values of the i-th bead is the i-th component of $\tilde{r}$. The host computer then revise $\Psi_{i,j}$ according to $\tilde{r}$.

The hardware design for Algorithm 4 is similar to some

systolic implementations for matrix vector multiplication with regular word-level and bit-level architectures [23]. While techniques such as polyhedral analysis, data pipelining and tiling have been used in deriving such implementations [22], 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 III. 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.

B. 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, $\textsc{Pass}(g, c, i, j)$ would have 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

$$\mathfrak{T}_P = \frac{1}{F}\sum_{g=0}^{G}(T - gc) = \frac{(G + 1)(2T - Gc)}{2F} \tag{23}$$

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

$$\mathfrak{T} = \mathfrak{T}_P + \mathfrak{T}_\epsilon \tag{24}$$

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.

V. Experimental Evaluation

We run two experiments to evaluate our proposed architecture. The first one examines the performance while the second one concerns the scalability. In this section, we first present the general experimental settings and then discuss the two experiments respectively.

A. General Settings

The mathematical derivation of our hardware design targets a Maxeler MAX3 acceleration system. The hardware is described in the MaxJ language and compiled with Maxeler MaxCompiler. The acceleration system is equipped with a Xilinx Virtex-6 FPGA. It communicates with the host computer via a PCI-Express interface. In our implementation, we deploy 384 beads and set the clock frequency to 100MHz.

We also build a CPU-based system by implementing the straightforward HAC estimation algorithm on the CPU platform in a server with an Intel Xeon CPU 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 compiling optimisation. To make a fair comparison, the IEEE single precision floating point numbers are used exclusively in both the hardware and software implementations.

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 [20], the data sets are generated using a vector autoregression (VAR) model [6]. The time series and lag parameters tested in the two experiments are different, and we will present these settings individually.

B. Experiment on Performance

In this experiment, we study the performance of our data flow engine. We enable all the 384 beads on the data flow engine to demonstrate its full power. Time series with different lengths are tested with different lag truncation parameters. More specifically, we use four time series with length $T = 10^5, 10^6, 10^7, 10^8$. Following [20], we use lag truncation parameters H in the form

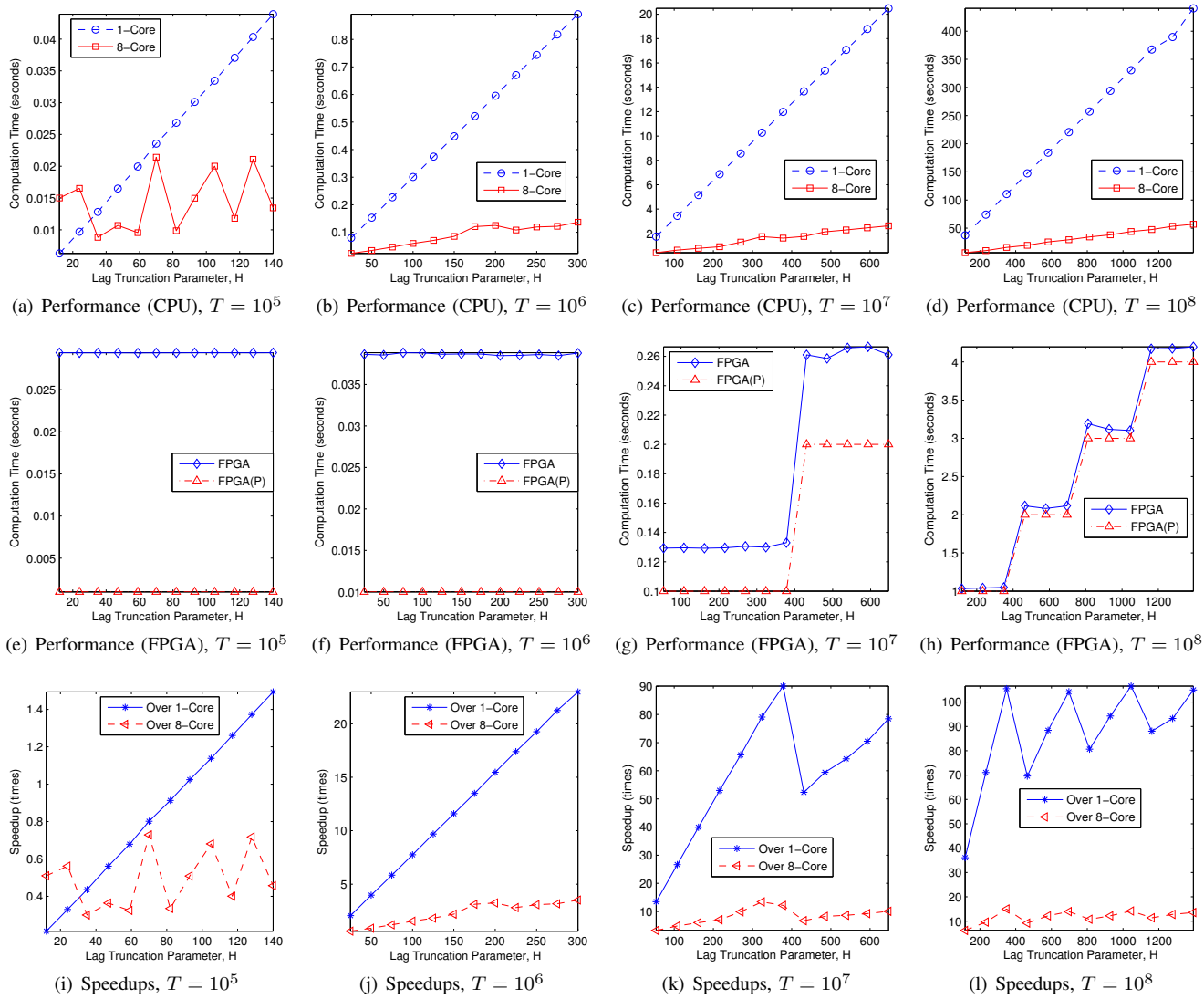
$$H = \lfloor \gamma T^{\frac{1}{3}} \rfloor \tag{25}$$

where $\lfloor x \rfloor$ is the smallest integer not larger than x; γ is a data-dependent positive real number. 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 [20].

Experimental results are shown in Fig. 3. It is clear that the speedup of the FPGA-based system is significant, especially for long time series where $T = 10^6, 10^7, 10^8$. The best speedup obtained in this experiment is 106 times compared to the CPU-based system running on a single core, when $T = 10^8$ and $H = 1045$.

The execution time of all systems increases as the lag truncation parameter grows. However, the growth pattern of the FPGA based system is significantly different from that of CPU-based systems. More specifically, the execution times of the two CPU-based system grow linearly with different slope. This linear growth can be explained by the time complexity of the algorithm. The time of the FPGA-based system increases like stairs. This is because the FPGA-based system handles 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 an zig-zag pattern



(a) Performance (CPU), $T = 10^5$ (b) Performance (CPU), $T = 10^6$ (c) Performance (CPU), $T = 10^7$ (d) Performance (CPU), $T = 10^8$

(e) Performance (FPGA), $T = 10^5$ (f) Performance (FPGA), $T = 10^6$ (g) Performance (FPGA), $T = 10^7$ (h) Performance (FPGA), $T = 10^8$

(i) Speedups, $T = 10^5$ (j) Speedups, $T = 10^6$ (k) Speedups, $T = 10^7$ (l) Speedups, $T = 10^8$

Fig. 3. Results on performance: each column of figures corresponds to a time series length. The first two rows of figures are performance results from the CPU-based system and the FGPA-based system respectively. FPGA(P) is the theoretical pipeline processing time. The third row records the speedup of the FPGA-based system over the CPU-based system.

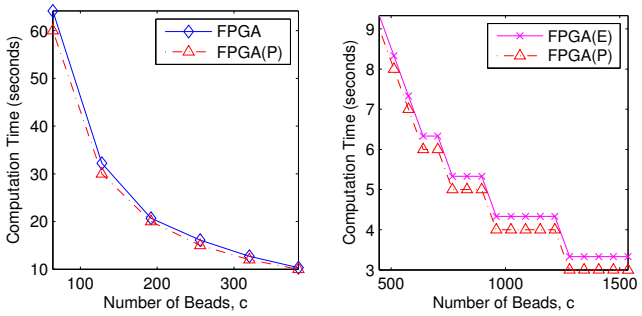
in a periodical manner, as shown in Fig. 3(k) and 3(l). We may also obtain further interesting observations from this zig-zag pattern. For example, the amplitude of the zig-zag pattern shrinks as the lag truncation parameter grows, since the redundant computation time becomes less significant compared to the total execution time.

C. Experiment on Scalability

As the memory bandwidth requirement of the proposed architecture is constant, deploying more beads along the pipeline is a direct way to boost performance. We analyse how the number of beads influences performance by running a controlled experiment to demonstrate the scalability of our system. In this experiment we use the largest problem in the previous experiment where the series length $T = 10^8$ and we set a large truncation parameter $H = 3839$. The execution times of the CPU-based system are 1186.31s and 151.26s on a single core and eight cores respectively.

We test the performance of the FPGA-based system with different numbers of beads. The maximum number of beads that we manage to deploy is 384; as a consequence we only test the configurations where the number of beads is no more than 384. The experimental results are plotted in Fig. 4(a) along with the corresponding theoretical pipeline processing time. When c, the number of beads, is 384, the execution time reaches 10.67s, achieving 111 and 14 times speedup over the CPU-based system on one core and eight cores respectively.

There is a significant trend that the execution time is shortened as the number of beads c increases. Another trend, which is not significant in the figure, is that the gap between the experimental execution time and the theoretical pipeline processing time becomes narrower as the number of beads c increases. The gap decreases from 2.2934s when $c = 64$, to 0.6683s when $c = 384$. As shown in Fig. 4, the theoretical pipeline processing time continues to decrease when more beads are deployed along the pipeline. We can obtain a



(a) Tested Scenario (b) Extrapolated Scenario

Fig. 4. Results on scalability: FPGA(P) is the theoretical pipeline processing time. FPGA(E) is the estimated performance of the FPGA-based system in the extrapolated scenario.

conservative estimate of the performance of a system with more than 384 beads, by extrapolating the gap value of 0.6683s for 384 beads as an upper bound for designs with more than 384 beads. The computation time values can then be estimated by adding this gap value to the corresponding theoretical pipeline processing time. This estimation suggests that our proposed architecture scales well with the amount of available computational resources.

VI. Conclusion

This paper presents a reconfigurable computing solution to heteroskedasticity and autocorrelation consistent (HAC) covariance matrix estimation for multivariate time series. To the best of our knowledge, our approach is the first to apply FPGAs to statistical analysis of time series.

Rather than providing a hardware design for an existing HAC estimation algorithm, we derive a novel algorithm which is designed exclusively for hardware implementation. This algorithm exploits the capabilities of a reconfigurable computing platform and avoids limitations like the memory bottleneck. We then propose an efficient and scalable hardware architecture based on our algorithm and analyse its performance using both theoretical and empirical approaches. Our experimental system implemented in a Vertex-6 FPGA achieves up to 111 times speedup over a single-core CPU, and up to 14 times speedup over an 8-core CPU.

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

Acknowledgement

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] N. Jegadeesh and S. Titman, "Returns to buying winners and selling losers: Implications for stock market efficiency," *The Journal of Finance*, vol. 48, no. 1, pp. 65–91, 1993.

[2] T. Bollerslev, G. Tauchen, and H. Zhou, "Expected stock returns and variance risk premia," *Review of Financial Studies*, vol. 22, no. 11, pp. 4463–4492, 2009.

[3] G. Bekaert, C. Harvey, C. Lundblad, and S. Siegel, "What segments equity markets?" *Review of Financial Studies*, vol. 24, no. 12, pp. 3841–3890, 2011.

[4] D. Sart, A. Mueen, W. Najjar, E. Keogh, and V. Niennattrakul, "Accelerating dynamic time warping subsequence search with GPUs and FPGAs," in *International Conference on Data Mining*, 2010, pp. 1001–1006.

[5] Z. Wang, S. Huang, L. Wang, H. Li, Y. Wang, and H. Yang, "Accelerating subsequence similarity search based on dynamic time warping distance with FPGA," in *International Symposium on Field-Programmable Gate Arrays*, 2013.

[6] J. D. Hamilton, *Time series analysis*. Cambridge University Press, 1994.

[7] W. Newey and K. West, "A simple, positive semi-definite, heteroskedasticity and autocorrelation consistent covariance matrix," *Econometrica: Journal of the Econometric Society*, vol. 55, pp. 703–708, 1987.

[8] D. Andrews, "Heteroskedasticity and autocorrelation consistent covariance matrix estimation," *Econometrica: Journal of the Econometric Society*, vol. 59, pp. 817–858, 1991.

[9] T. Preis, P. Virnau, W. Paul, and J. J. Schneider, "Accelerated fluctuation analysis by graphic cards and complex pattern formation in financial markets," *New Journal of Physics*, vol. 11, no. 9, p. 093024, 2009.

[10] D. Gembris, M. Neeb, M. Gipp, A. Kugel, and R. Männer, "Correlation analysis on gpu systems using nvidias cuda," *Journal of real-time image processing*, vol. 6, no. 4, pp. 275–280, 2011.

[11] Z. Baker and V. Prasanna, "Efficient hardware data mining with the apriori algorithm on FPGAs," in *IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2005, pp. 3–12.

[12] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *International Conference on Very Large Databases*, 1994, pp. 487–499.

[13] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Berkeley Symposium on Mathematical Statistics and Probability*, 1967, pp. 281–297.

[14] J. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, Mar. 1986.

[15] T. Saegusa and T. Maruyama, "An FPGA implementation of k-means clustering for color images based on kd-tree," in *International Conference on Field Programmable Logic and Applications*, 2006, pp. 1–6.

[16] R. Narayanan, D. Honbo, G. Memik, A. Choudhary, and J. Zambreno, "An FPGA implementation of decision tree classification," in *Conference on design, automation and test in Europe*, 2007, pp. 1–6.

[17] C. Guo, H. Fu, and W. Luk, "A fully-pipelined expectation - maximization engine for Gaussian mixture models," in *International Conference on Field-Programmable Technology*, 2012, pp. 182–189.

[18] A. Zeileis, "Econometric computing with HC and HAC covariance matrix estimators," Distributed with the 'sandwich' R package, 2004.

[19] A. Cottrell and R. Lucchetti, "Gretl user's guide," Distributed with the Gretl library, 2012.

[20] W. Newey and K. West, "Automatic lag selection in covariance matrix estimation," *Review of Economic Studies*, vol. 61, pp. 631–653, 1994.

[21] C. Lin, H. So, and P. Leong, "A model for matrix multiplication performance on FPGAs," in *International Conference on Field Programmable Logic and Applications*, 2011, pp. 305–310.

[22] A. C. Jacob, J. D. Buhler, and R. D. Chamberlain, "Rapid RNA folding: analysis and acceleration of the zuker recurrence," in *International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2010, pp. 87–94.

[23] R. Urquhart and D. Wood, "Systolic matrix and vector multiplication methods for signal processing," *IEE Proceedings*, vol. 131, no. 6, pp. 623–631, 1984.