Accelerating Transfer Entropy Computation

Shengjia Shao, Ce Guo and Wayne Luk
Department of Computing
Imperial College London
United Kingdom
E-mail: {shengjia.shao12, ce.guo10, w.luk}@imperial.ac.uk

Stephen Weston
Maxeler Technologies
London
United Kingdom
E-mail: weston@maxeler.com

***Abstract*—Transfer entropy is a measure of information transfer between two time series. It is an asymmetric measure based on entropy change which only takes into account the statistical dependency originating in the source series, but excludes dependency on a common external factor. Transfer entropy is able to capture system dynamics that traditional measures cannot, and has been successfully applied to various areas such as neuroscience, bioinformatics, data mining and finance. When time series becomes longer and resolution becomes higher, computing transfer entropy is demanding. This paper presents the first reconfigurable computing solution to accelerate transfer entropy computation. The novel aspects of our approach include a new technique based on Laplace's Rule of Succession for probability estimation; a novel architecture with optimised memory allocation, bit-width narrowing and mixed-precision optimisation; and its implementation targeting a Xilinx Virtex-6 SX475T FPGA. In our experiments, the proposed FPGA-based solution is up to 111.47 times faster than one Xeon CPU core, and 18.69 times faster than a 6-core Xeon CPU.**

I. Introduction

In many applications, one needs to detect causal directions between different parts of the system in order to understand system dynamics and make estimations on its actual physical structure. This often involves observing the system, recording system behaviour as time series of signals, and analysing the time series.

The simplest statistical measure is correlation. However, correlation does not necessarily imply causality. Information-based measures have been proposed to deal with causality. Transfer entropy is an asymmetric information theoretic measure designed to capture directed information flow between variables [1]. Given two processes X and Y, the transfer entropy from X to Y is the amount of uncertainty reduced in Y's future values by knowing the past values of X given past values of Y. Transfer entropy can be used in effectively distinguishing between the driving and the responding variables, which makes it superior to traditional information-based measures such as Time-Delayed Mutual Information, in the sense of detecting causal relationships.

Since its introduction in 2000, transfer entropy has proven to be a powerful tool for various applications. Honey et al. uses transfer entropy to analyse the functional connectivity of different areas in the cerebral cortex [2]. A transfer entropy matrix is built with element (i, j) the transfer entropy from $Area_i$ to $Area_j$. This matrix is then thresholded to derive a binary matrix for functional connectivity (TE Network), which is the estimation of how cortex areas are connected based on transfer entropy analysis. It is found that when using long data samples, TE Network and the actual structural network show up to 80% overlap, whereas the overlap between structural networks and functional networks extracted with mutual information and wavelet-based tools is lower. Ver Steeg and Calstyan use transfer entropy to measure the information transfer in social media [3]. They calculate the transfer entropy from user A to user B ($T_{A\to B}$) and that in the opposite direction ($T_{B\to A}$). If $T_{A\to B}$ is much larger than $T_{B\to A}$, then A is said to have influence on B, but not vice versa. Real data sets from *Twitter* are analysed and result in a network of influence. This allows us to identify 'influential users' and the most important links in a big network, which is beneficial to Data Mining. In addition, transfer entropy is useful in finance. Li et al. uses transfer entropy to analyse the interaction of banks in the financial market [4]. The transfer entropies of several banks' stock prices are calculated, resulting in a matrix to estimate the interbank exposure. The matrix is further refined with disclosed information and some other adjustments. Finally the interbank exposure matrix is used in simulation to analyse what will happen to other banks if a major bank defaults. This helps financial institutions to manage risk, and provides useful information for regulators to prevent financial catastrophes, such as the 2008 crisis, from happening again.

The interesting features of transfer entropy make it ideal for analysing interactions between variables in a complex system. However there are significant theoretical and practical challenges. To compute transfer entropy we will need the probability distribution of the values in the time series. In theory, to calculate this distribution we must have the entire time series. However, in most cases there are only limited samples available, so there is a challenge of computing transfer entropy based on incomplete time series data.

Besides, even if we have the exact probability distribution, transfer entropy could be computationally intensive. The time complexity of transfer entropy computation is $O(R^3)$, where R stands for resolution. When resolution becomes higher, granularity becomes smaller so that transfer entropy with improved accuracy can be obtained. In reality, given the limited computing resource, one often has to set a certain resolution to balance accuracy and time complexity. In Li's work, the resolution is set to $(N/4)^{1/3}$, where N is the number of samples, the length of time series [4]. It is hard to say whether $R = (N/4)^{1/3}$ is good enough, but larger resolution will definitely lead to more accurate results.

Fortunately, transfer entropy computation is parallelisable, so it has the potential to benefit from hardware acceleration. However, the limited CPU-FPGA bandwidth and limited logic resource create challenges for implementation.

In this paper, we develop a novel method to estimate the probability distributions used in transfer entropy computation. Also, we present a reconfigurable architecture for computing transfer entropy and implement it on a commercial FPGA. We optimise memory allocation to effectively reduce I/O bandwidth requirement. In addition, bit-width narrowing is used to cut down BRAM usage, and to further reduce I/O overhead. To control resource usage, we apply mixed-precision optimisation to gain double precision accuracy without sacrificing parallelism. To the best of our knowledge, we are the first to apply reconfigurable computing techniques to transfer entropy computation.

The contributions of this paper are as follows:

- A new method based on Laplace's rule of succession to estimate probabilities used for computing transfer entropy. This method targets common cases in which the complete knowledge of time series is unavailable.
- A novel hardware architecture for computing transfer entropy. Optimised memory allocation and bit-width narrowing are used to effectively reduce I/O overhead and BRAM usage in FPGA. Mixed-precision optimisation is used to gain double precision accuracy with a modest amount of hardware resources.
- Implementation on a Xilinx Virtex-6 FPGA and experimental evaluation using random numbers and historical Forex data. The proposed system is up to 111.47 times faster than a single Xeon CPU core, and 18.69 times faster than a 6-core Xeon CPU.

The rest of this paper is organised as follows. Section II covers basic background material for transfer entropy. Section III presents our novel method for probability estimation. Section IV describes the proposed hardware architecture. Section V presents FPGA implementation details. Section VI provides experimental evaluation and discussion. Finally, Section VII concludes this paper and presents probabilities for future work.

II. Background

In this section, we make a brief introduction to the concept of transfer entropy and how it is computed. Also, we will review existing work on hardware acceleration of time series analysis.

A. Introduction to Transfer Entropy

Transfer entropy is a measure of directed information transfer between two time series. A time series, e.g., stock prices at different times, can be expressed as:

$$X = \{x_1, x_2, \cdots, x_T\} \tag{1}$$

Here, T is the time series' length, which is given by the number of observations. So x_1 is the stock price at the first observation, x_2 is the price at the second observation, etc.

Given two time series X and Y, we define an entropy rate which is the amount of additional information required to represent the value of the next observation of X:

$$h_1 = -\sum_{x_{n+1}, x_n, y_n} p(x_{n+1}, x_n, y_n) \log_2 p(x_{n+1}|x_n, y_n) \tag{2}$$

Also, we define another entropy rate assuming that x_{n+1} is independent of y_n:

$$h_2 = -\sum_{x_{n+1}, x_n, y_n} p(x_{n+1}, x_n, y_n) \log_2 p(x_{n+1}|x_n) \tag{3}$$

Then the *Transfer Entropy* from Y to X can be given by $h_2 - h_1$, which corresponds to the information transferred from Y to X:

$$\begin{aligned} T_{Y \to X} &= h_2 - h_1 \\ &= \sum_{x_{n+1}, x_n, y_n} p(x_{n+1}, x_n, y_n) \log_2 \left(\frac{p(x_{n+1}|x_n, y_n)}{p(x_{n+1}|x_n)} \right) \end{aligned} \tag{4}$$

Similarly, we can define the transfer entropy from X to Y:

$$T_{X \to Y} = \sum_{y_{n+1}, x_n, y_n} p(y_{n+1}, x_n, y_n) \log_2 \left(\frac{p(y_{n+1}|x_n, y_n)}{p(y_{n+1}|y_n)} \right) \tag{5}$$

B. Computing Transfer Entropy

Using the definition of conditional probabilities, (4) and (5) can be rewritten as:

$$\begin{aligned} &T_{Y \to X} \\ &= \sum_{x_{n+1}, x_n, y_n} p(x_{n+1}, x_n, y_n) \log_2 \left(\frac{p(x_{n+1}, x_n, y_n) p(x_n)}{p(x_n, y_n) p(x_{n+1}, x_n)} \right) \end{aligned} \tag{6}$$

$$\begin{aligned} &T_{X \to Y} \\ &= \sum_{y_{n+1}, x_n, y_n} p(y_{n+1}, x_n, y_n) \log_2 \left(\frac{p(y_{n+1}, x_n, y_n) p(y_n)}{p(x_n, y_n) p(y_{n+1}, y_n)} \right) \end{aligned} \tag{7}$$

Given T observations of the time series X and Y, preprocessing is needed to calculate the (joint) probability distributions $p(x_n)$, $p(y_n)$, $p(x_{n+1}, x_n)$, $p(y_{n+1}, y_n)$, $p(x_n, y_n)$, $p(x_{n+1}, x_n, y_n)$ and $p(y_{n+1}, x_n, y_n)$. Then transfer entropy can be calculated by (6) and (7).

In preprocessing, we first go through the time series, counting the number of occurrences for each (joint) value of x_n, y_n, (x_{n+1}, x_n), (y_{n+1}, y_n), (x_n, y_n), (x_{n+1}, x_n, y_n) and (y_{n+1}, x_n, y_n). Then the probability distribution can be obtained by normalising - dividing the number of occurrences by the number of data elements, which is T for x_n, y_n, (x_n, y_n) and $T-1$ for (x_{n+1}, x_n), (y_{n+1}, y_n), (x_{n+1}, x_n, y_n), (y_{n+1}, x_n, y_n).

When computing transfer entropy, quantisation must be taken into consideration. When X has P values and Y has Q values, their joint probability distribution $p(x_{n+1}, x_n, y_n)$ will have $P \times P \times Q$ elements. This can lead to a table which is too big to fit into computer memory.

In practice, quantisation is used to trade off between accuracy and memory resource usage. One can set a *Resolution* (R), which corresponds to the number of values allowed. Then granularity (Δ) is given by:

$$\Delta = \frac{MAX - MIN}{R - 1} \tag{8}$$

Here, MAX and MIN stand for the maximum and minimum values of the time series, respectively. For example, if $R = 100$, time series X and Y are quantised into 100 levels. Then the quantised X and Y are used in preprocessing. As a result, the joint probability distribution $p(x_{n+1}, x_n, y_n)$ will have 10^6 elements. Larger resolution will lead to more quantisation levels, which uses more memory resources to achieve better accuracy.

Besides, the time complexity of transfer entropy computation is determined by the resolution rather than by the length of time series. This is because the computation is based on (joint) probability distributions, and the number of iterations is the number of elements in the joint probability distributions $p(x_{n+1}, x_n, y_n)$ and $p(y_{n+1}, x_n, y_n)$, as shown in (6) and (7). If $R = 100$, there will be 10^6 elements to be accumulated to derive $T_{X\rightarrow Y}$. Therefore, the time complexity of transfer entropy computation is $O(R^3)$. As time complexity grows rapidly with R, computing transfer entropy would be demanding for CPU.

C. *Accelerating Time Series Analysis*

Time series analysis methods analyse time series data so as to find patterns, make predictions or calculate various statistical metrics. There are many types of time series analysis methods, but only a few of them have hardware acceleration solutions. In short, the hardware acceleration of time series analysis is still a new topic.

Gembris et al. uses NVIDIA's GPU to accelerate correlation analysis [5]. They also compare CPU, GPU and FPGA for the performance of doing correlation calculation. Lin and Medioni compute mutual information using GPU [6]. Castro-Pareja, Jagadeesh, and Shekhar present the FPGA implementation for mutual information computation [7]. Guo and Luk design a FPGA accelerator for ordinal pattern encoding, and apply it to the computation of permutation entropy [8]. For transfer entropy, as it is a new statistical metric, we are not aware of any published work on its hardware acceleration.

III. Probability Estimation

This section presents a new method, based on Laplace's rule of succession, for estimating probabilities used in transfer entropy computation.

The transfer entropy defined in (6) and (7) depends on the (joint) probabilities, such as $p(x_{n+1}, x_n, y_n)$. The exact values of these probabilities are unknown, but it is possible to estimate them from data. Assume that each of these probabilities follows a multinomial distribution. From the perspective of frequentist statistics, a reasonable set of estimates is

$$\hat{p}(x_{n+1}, x_n, y_n) = \frac{N(x_{n+1}, x_n, y_n)}{T-1} \tag{9}$$

$$\hat{p}(y_{n+1}, x_n, y_n) = \frac{N(y_{n+1}, x_n, y_n)}{T-1} \tag{10}$$

$$\hat{p}(x_{n+1}, x_n) = \frac{N(x_{n+1}, x_n)}{T-1} \tag{11}$$

$$\hat{p}(y_{n+1}, y_n) = \frac{N(y_{n+1}, y_n)}{T-1} \tag{12}$$

$$\hat{p}(x_n, y_n) = \frac{N(x_n, y_n)}{T} \tag{13}$$

$$\hat{p}(x_n) = \frac{N(x_n)}{T} \tag{14}$$

$$\hat{p}(y_n) = \frac{N(y_n)}{T} \tag{15}$$

where $N(X)$ is the number of occurrences of pattern X in the data, and T is the length of the time series.

One can calculate the transfer entropy by replacing the probabilities in (6) and (7) with their corresponding estimates. Note that a hidden assumption in (6) and (7) is that all the probabilities must be non-zero. Nevertheless, an estimate produced using the above equations is zero if the corresponding pattern never appears in the observations. However, $N(X) = 0$ does not necessarily imply $p(X) = 0$, since it may happen because our observations are incomplete.

To solve this problem, we associate each possible pattern with an imaginary count to obtain the following estimates. We add one to the observed number of occurrences, and modify the denominator accordingly to make sure the modified probability still sums to one:

$$\hat{p}(x_{n+1}, x_n, y_n) = \frac{N(x_{n+1}, x_n, y_n) + 1}{T - 1 + R^3} \tag{16}$$

$$\hat{p}(y_{n+1}, x_n, y_n) = \frac{N(y_{n+1}, x_n, y_n) + 1}{T - 1 + R^3} \tag{17}$$

$$\hat{p}(x_{n+1}, x_n) = \frac{N(x_{n+1}, x_n) + 1}{T - 1 + R^2} \tag{18}$$

$$\hat{p}(y_{n+1}, y_n) = \frac{N(y_{n+1}, y_n) + 1}{T - 1 + R^2} \tag{19}$$

$$\hat{p}(x_n, y_n) = \frac{N(x_n, y_n) + 1}{T + R^2} \tag{20}$$

$$\hat{p}(x_n) = \frac{N(x_n) + 1}{T + R} \tag{21}$$

$$\hat{p}(y_n) = \frac{N(y_n) + 1}{T + R} \tag{22}$$

An intuitive interpretation of our treatment is that we set a lower bound of probability to each legitimate pattern disregarding the pattern appears in the data. This lower bound decreases as the length of the data grows. From the view of frequentist statistics, our treatment is an application of Laplace's rule of succession. From the view of Bayesian statistics, the treatment corresponds to mixing likelihood values with a Dirichlet prior with parameter one [9]. Similar treatments have been applied to probability inference problems to eliminate side effects of zero probabilities [10].

TABLE I. DATA ACCESS PATTERNS

Table Name	Number of Elements	Data Request	Access Times	Storage
$N(x_{n+1}, x_n, y_n)$	R^3	K elements per inner loop iteration	Read Once	DRAM in host CPU
$N(y_{n+1}, x_n, y_n)$	R^3	K elements per inner loop iteration	Read Once	DRAM in host CPU
$N(x_{n+1}, x_n)$	R^2	K elements per inner loop iteration	Read R times	BRAM on FPGA
$N(y_{n+1}, y_n)$	R^2	K elements per inner loop iteration	Read R times	BRAM on FPGA
$N(x_n, y_n)$	R^2	one element per middle loop iteration	Read Once	DRAM in host CPU
$N(x_n)$	R	one element per middle loop iteration	Read R times	BRAM on FPGA
$N(y_n)$	R	one element per outer loop iteration	Read Once	BRAM on FPGA

When applying Laplace's rule of succession, we essentially add an 'imaginary count' to each case of the multinomial distribution. For example, consider a dice with 6 faces. If we throw a loaded dice for 10 times, we may get the following histogram over the 6 faces: (1, 4, 0, 2, 1, 2). Following the manner in (9) - (15), it is possible to estimate the probability of each face as (0.1, 0.4, 0, 0.2, 0.1, 0.2). However this straightforward estimation is obviously problematic. Even though Face 3 has never appeared in the 10 attempts, it would be wrong to rule out its possibility - it will probably be observed if we throw the dice for a few more times. In this case, we can apply Laplace's rule of succession to the observed data by adding each count by one - an imaginary count. So the observations become (2, 5, 1, 3, 2, 3); the total number of attempts becomes $10+6=16$. We add 6 to the total count because we add the count of each of the 6 cases by one. This is why R, R^2 or R^3 appear in the denominator in (16) - (22).

With this in mind, transfer entropy can be computed using the following two equations. Note that we do not need to calculate $\hat{p}$ specifically. Since to add or to divide by a constant can be implemented in the program (and in hardware) straightforwardly, we would use the observed numbers of occurrences (N) as inputs.

$$T_{Y\to X} \approx \sum_{x_{n+1},x_n,y_n} \hat{p}(x_{n+1},x_n,y_n)\log_2\left(\frac{\hat{p}(x_{n+1},x_n,y_n)\hat{p}(x_n)}{\hat{p}(x_n,y_n)\hat{p}(x_{n+1},x_n)}\right) \tag{23}$$

$$T_{X\to Y} \approx \sum_{y_{n+1},x_n,y_n} \hat{p}(y_{n+1},x_n,y_n)\log_2\left(\frac{\hat{p}(y_{n+1},x_n,y_n)\hat{p}(y_n)}{\hat{p}(x_n,y_n)\hat{p}(y_{n+1},y_n)}\right) \tag{24}$$

IV. HARDWARE DESIGN

In this section, we present our FPGA system architecture for computing transfer entropy. Our system is designed to reduce CPU-FPGA I/O overhead and FPGA logic usage, which is achieved by optimised memory allocation, bit-width narrowing and mixed-precision optimisation.

A. Optimised Memory Allocation

The core computation is equation (23) and (24). From a computing perspective, (23) and (24) are 3-level nested loops, as there are $R \times R \times R$ elements in $N(x_{n+1}, x_n, y_n)$ and $N(y_{n+1}, x_n, y_n)$. Since we are computing $T_{Y\to X}$ and $T_{X\to Y}$ at the same time, we let the iteration of x_{n+1} and y_{n+1} be the inner loop, x_n the middle loop and y_n the outer loop.

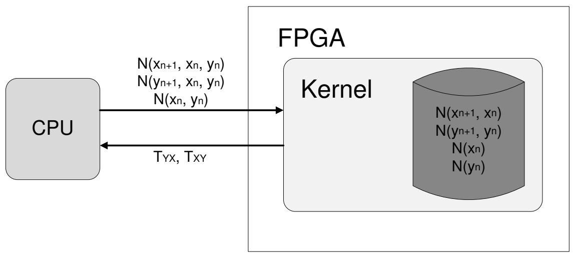

Fig. 1. System Architecture. Number of occurrence tables $N(x_{n+1}, x_n)$, $N(y_{n+1}, y_n)$, $N(x_n)$ and $N(y_n)$ are mapped to the BRAM inside FPGA when initialising. Other tables ($N(x_{n+1}, x_n, y_n)$, $N(y_{n+1}, x_n, y_n)$ and $N(x_n, y_n)$) are sent to FPGA at run-time. Transfer entropy results $T_{Y\to X}$ and $T_{X\to Y}$ are sent back to CPU.

The first optimisation is allocating number of occurrences tables. Since PCI-E bandwidth is limited, we map some small and medium-sized tables to BRAM during initialisation while sending large ones at run-time. In this way, we pipeline data transfer and computing. With bit-width narrowing, the bandwidth required is slightly larger than PCI-E bandwidth. Although there will still be some overhead due to bandwidth limitation, its impact on overall performance is insignificant. On the other hand, if we send all tables to DRAM via PCI-E during initialisation, this data transfer time cannot be overlapped with computing, so the overall time will be longer.

The data access patterns of number of occurrence tables are summarised in Table I. We map $N(x_{n+1}, x_n)$, $N(y_{n+1}, y_n)$, $N(x_n)$ and $N(y_n)$ to BRAM, which are small tables or frequently-accessed medium-sized tables. When resolution (R) is about 1000, the total size of the 4 tables are just several MBs, which would fit in the on-chip BRAM. Meanwhile, as three dimensional tables $N(x_{n+1}, x_n, y_n)$ and $N(y_{n+1}, x_n, y_n)$ are very large tables (10^9 elements in each table if $R = 1000$, so the two tables are in GBs) only used once during the computation, we stream the two tables to FPGA at runtime. Table $N(x_n, y_n)$ is also streamed to FPGA at runtime due to BRAM resource limitation. Note that in every middle loop FPGA only reads one element from $N(x_n, y_n)$, so streaming this table to FPGA only has marginal influence on I/O bandwidth usage.

B. Bit-width Narrowing

As mentioned above, we optimise memory allocation to reduce I/O overhead. $N(x_{n+1}, x_n, y_n)$, $N(y_{n+1}, x_n, y_n)$ and $N(x_n, y_n)$ are sent to FPGA at run-time while other tables are mapped to BRAM during initialisation. While this memory allocation is effective, it requires a large amount of BRAM. Therefore, the resolution supported will be limited by the BRAM resource available.

TABLE II. BIT-WIDTH NARROWING FOR $N(x_{n+1}, x_n)$ AND $N(y_{n+1}, y_n)$

Resolution	200	300	400	500	600	700	800	900	1000	1100	1200
Largest Element	26182	11959	7049	4331	3037	2286	1768	1415	1161	971	837
Format	uint16	uint16	uint16	uint16	uint12	uint12	uint12	uint12	uint12	uint10	uint10
Size (Two Tables in Total)	0.15MB	0.34MB	0.61MB	0.95MB	1.03MB	1.40MB	1.83MB	2.32MB	2.86MB	2.88MB	3.43MB

TABLE III. BIT-WIDTH NARROWING FOR $N(x_{n+1}, x_n, y_n)$ AND $N(y_{n+1}, x_n, y_n)$

Resolution	200	300	400	500	600	700	800	900	1000	1100	1200
Largest Element	194	80	46	28	22	17	15	14	11	11	10
Format	uint8	uint8	uint6	uint5	uint5	uint5	uint4	uint4	uint4	uint4	uint4
Size (Two Tables in Total)	15.26MB	51.50MB	91.55MB	149.01MB	257.49MB	408.89MB	488.28MB	695.22MB	953.67MB	1.269GB	1.648GB

To address this problem, a technique known as bit-width narrowing is used. In our tests, we use random numbers and historical foreign exchange data. As resolution becomes larger, number of occurrences tables become larger but the elements in the table are smaller. Table II shows bit-width narrowing for $N(x_{n+1}, x_n)$ and $N(y_{n+1}, y_n)$, with bit-width selected to represent the largest element in the table. By using custom unsigned integers ranging from 10-bit to 16-bit instead of the standard 32-bit `int`, we only use 31%-50% of the original memory space, which enables mapping the data tables to FPGA BRAM. As for $N(x_n)$ and $N(y_n)$, the largest number is in millions, so standard 32-bit `int` is used. Note there are only $2R$ elements in total in $N(x_n)$ and $N(y_n)$, they only take several KBs of BRAM.

Besides, bit-width narrowing is also used to reduce I/O overhead. Since we stream $N(x_{n+1}, x_n, y_n)$ and $N(y_{n+1}, x_n, y_n)$ from CPU to FPGA at run-time, using fewer bits to represent $N(x_{n+1}, x_n, y_n)$ and $N(y_{n+1}, x_n, y_n)$ will effectively cut down bandwidth usage. Instead of sending 32-bit `int` to FPGA, we use unsigned integers with various bit widths ranging from 4 bits to 8 bits, depending on the largest number in the table. As a result, 75% to 87.5% bandwidth resources are saved. The details of bit-width narrowing for $N(x_{n+1}, x_n, y_n)$ and $N(y_{n+1}, x_n, y_n)$ are shown in Table III.

It is worth pointing out that bit-width narrowing depends on the input data. When using the hardware system to compute the transfer entropy of a particular kind of time series, it is useful to find the range of the elements in the tables in order to determine the optimal bit-width.

C. Mixed-Precision Optimisation

In the C program for computing transfer entropy, $log_2()$ and the accumulator is implemented using IEEE 754 double precision, which is the default standard for scientific computing. However, in FPGA a floating point accumulator will result in excessive hardware resource usage. As the dynamic range of input data is small, it is possible to use fixed-point number representation for the accumulator to reduce resource usage without sacrificing accuracy. In the kernel, there are R^3 numbers to be accumulated. Larger resolution will lead to a larger sum in the accumulator so more integer bits are needed. The accumulator in our system supports 64-bit fixed-point data representation with 28 integer bits and 36 fractional bits. This setting works well in our experiments.

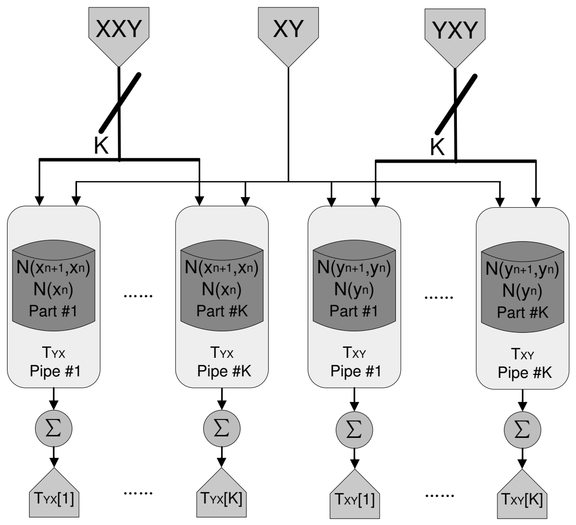

Fig. 2. Kernel Architecture. This figure shows the datapath of the kernel with control logic omitted. Here XXY, YXY and XY stands for $N(x_{n+1}, x_n, y_n)$, $N(y_{n+1}, x_n, y_n)$ and $N(x_n, y_n)$, respectively. On each cycle, K elements from $N(x_{n+1}, x_n, y_n)$ and $N(y_{n+1}, x_n, y_n)$ are sent from CPU to FPGA, feeding the corresponding K pipes. A new value of $N(x_n, y_n)$ is sent to FPGA each middle loop (R/K cycles), and is shared by all pipes.

Inside the kernel, the most resource consuming part is the logic for $log_2()$. Unlike the accumulator, $log_2()$ uses much more resources when it is done in fixed-point rather than in floating-point. Consequently, we use floating point for representing $log_2()$. The resource usage of $log_2()$ is closely related to the number of mantissa bits. In our system, we adopt the format of $log_2()$ to be 40-bit floating point arithmetic with 8 exponent bits and 32 mantissa bits. We will explore the relationship between the number of mantissa bits and accuracy as well as parallelism in section VI.

D. Kernel Architecture

The kernel architecture is shown in Figure 2. Since there is no data dependency between different iterations except for the accumulator, the loop can be effectively strip-mined in hardware to deliver high performance.

We build $2K$ computing pipes for calculating transfer entropy: K pipes for $T_{Y\rightarrow X}$ and the other K pipes for $T_{X\rightarrow Y}$. K is a parameter to be specified at compile-time. The $2K$

TABLE IV. FPGA RESOURCE USAGE (RESOLUTION = 1200)

	LUT	Primary FF	Secondary FF	DSP	BRAM18
Total Available	297600	297600	297600	2016	2128
Total Used	201697	215724	42555	1014	2011
Usage (%)	67.77%	72.49%	14.30%	40.77%	94.50%

pipes correspond to K iterations in the inner loop, so the loop is strip-mined by K. Since the $2K$ pipes read different parts of the table $N(x_{n+1}, x_n)$ and $N(y_{n+1}, y_n)$ with no overlap, we separate each of the tables into K parts and distribute them to the corresponding K pipes. The $2K$ pipes generate K partial sums of transfer entropy $T_{Y \to X}$ and $T_{X \to Y}$, respectively. These partial sums are sent back to CPU, summed and normalised to derive the final result.

The original loop has R^3 iterations. In our kernel, there are K pipes running concurrently, so the kernel needs to run for R^3/K cycles. Therefore, the kernel computing time is given by:

$$T_{Comp} = \frac{1}{Freq} \times \frac{R^3}{K} \tag{25}$$

Here, $Freq$ is FPGA frequency. Since the FPGA kernel needs data from CPU, we also need to consider the I/O time, which is the data size over bandwidth:

$$T_{I/O} = \frac{DATA\_SIZE}{BW} \tag{26}$$

When running the system, the FPGA can read data and do computation in a pipelined manner. So the total time is the maximum of the computing time and I/O time.

$$T_{Total} = max\{T_{Comp}, T_{I/O}\} \tag{27}$$

When $T_{Comp} > T_{I/O}$, the kernel is bounded by computing. In this case, performance can be improved by increasing parallelism (K) or incresing FPGA frequency. In contrast, when $T_{I/O} > T_{Comp}$, the kernel is bounded by I/O, so reducing I/O overhead is essential.

V. FPGA IMPLEMENTATION

The target hardware of our system is a Xilinx Virtex-6 FPGA. We deploy 48 transfer entropy computing pipes ($K = 24$). Table IV shows the hardware resource usage of Xilinx Virtex-6 SX475T FPGA when $R = 1200$. The LUT and FF usages are generally determined by the number of computing pipes (K). BRAM usage depends on resolution (R), because most of the BRAM is devoted to the number of occurrences tables $N(x_{n+1}, x_n)$ and $N(y_{n+1}, y_n)$. As a result, the resolution supported is limited by BRAM resource available. In the target platform, the largest resolution achievable in one Virtex-6 FPGA is 1200, using 94.50% BRAM.

When resolution is larger than 1200, we can use more FPGAs and distribute the pipes for $T_{X \to Y}$ and $T_{X \to Y}$ among them. As shown in Figure 2, the BRAM for $N(x_{n+1}, x_n)$ and $N(y_{n+1}, y_n)$ is distributed among multiple pipes. Consequently, by distributing the computing pipes, BRAM usage is also distributed. In this way, an arbitrary resolution can be supported, providing that there are enough FPGAs.

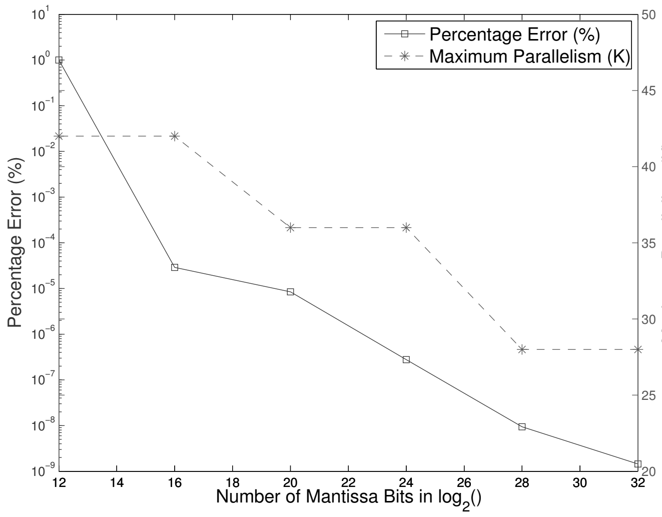

Fig. 3. Percentage Error (%) and Maximum Parallelism (K) vs. Number of Mantissa Bits in $log_2()$. Test time series are 10^9 random numbers. Resolution is fixed at 1000. Percentage error is measured against CPU result. Parallelism is measured by the number of computing pipes (K) for $T_{X \to Y}$ and $T_{Y \to X}$, e.g., if $K = 24$, then there are 48 pipes in total, 24 for $T_{X \to Y}$ and 24 for $T_{Y \to X}$.

VI. EXPERIMENTAL EVALUATION

The proposed system is built on a Maxeler MAX3 FPGA card with one Xilinx Virtex-6 SX475T FPGA running at 80MHz. The FPGA card is integrated with the host computer via the PCI-E Gen2 x8 interface. The host computer has one Intel Xeon X5650 CPU (6-cores running at 2.66GHz) and 48GB DDR3 1600MHz memory. The hardware design is described in the Maxeler MaxJ language and compiled to VHDL using Maxeler MaxCompiler. The VHDL description is then synthesised and mapped to FPGA with the Xilinx tool chain.

To evaluate the performance and accuracy of the hardware solution, we build a reference C program in double precision for transfer entropy to be run exclusively on the Intel Xeon CPU in the host computer. The C program is optimised for memory efficiency. To make performance comparison with 1 CPU core and 6 CPU cores, the C program has 1-thread and 6-thread versions. Multi-threading is done using OpenMP library. The FPGA host code and the reference C program are compiled with the Intel C Compiler with the highest compiler optimisation.

A. Accuracy versus Parallelism

Figure 3 shows the percentage error of transfer entropy and maximum parallelism can be achieved on one Virtex-6 FPGA as a function of the number of mantissa bits used in $log_2()$. The percentage error is measured against the reference C program running on CPU. Naturally, as the number of mantissa bits increases, hardware result becomes more accurate

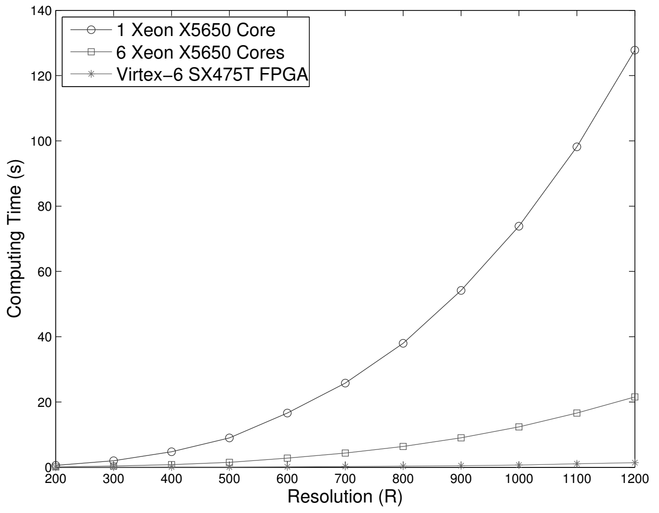

Fig. 4. Performance vs. Resolution using random numbers. Test time series are 10^9 random numbers. The Virtex-6 FPGA has 48 computing pipes ($K = 24$) running at 80MHz. $log_2()$ is done in 40-bit floating point with 8 exponent bits and 32 mantissa bits. Accumulator is set to 64-bit fixed point with 28 integer bits and 36 fractional bits.

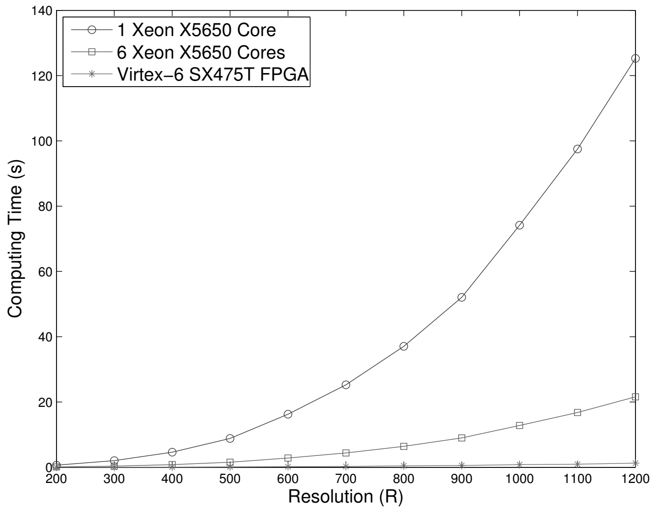

Fig. 5. Performance vs. Resolution using historical foreign exchange data. Test time series are EUR-JPY and GBP-USD rates on 30 Apr 2014. There are 75000 records in each of the time series. The Virtex-6 FPGA has 48 computing pipes ($K = 24$) running at 80MHz. $log_2()$ is done in 40-bit floating point with 8 exponent bits and 32 mantissa bits. Accumulator is set to 64-bit fixed point with 28 integer bits and 36 fractional bits.

while parallelism decreases due to more hardware resources required by logarithm. As shown in the figure, 28 or 32 mantissa bits will lead to the same parallelism, but the latter is more accurate. Therefore, our precision setting for $log_2()$ has 32 mantissa bits. In this case, the percentage error is about 0.000000001%, which is sufficiently accurate. We also need to point out that the maximum parallelism will utilise almost all FPGA resource, creating a demanding task for placement and routing tools. In some cases, kernel frequency has to be reduced in order to cut down flip-flop usage so that mapping, placement and routing could be successfully done. In the following performance tests, we set $K = 24$ although the maximum possible value is 28.

B. Performance versus Resolution - Random Numbers

For performance comparison, we measure the execution time of transfer entropy computation, which corresponds to the computing time of (23) and (24). We run a series of tests using different resolutions ranging from 200 to 1200. The FPGA has 48 computing pipes ($K = 24$), 24 for $T_{X\to Y}$ and 24 for $T_{Y\to X}$. The precision of $log_2()$ is 40-bit floating point with 8 exponent bits and 32 mantissa bits. The accumulator is 64-bit fixed-point with 28 integer bits and 36 fractional bits. Time series X and Y used in the experiments contain 10^9 random numbers. The FPGA runs at 80MHz. The performance results of single Xeon CPU core, 6 Xeon CPU cores and one Virtex-6 FPGA is shown in Figure 4.

From the figure, FPGA demonstrates high performance for transfer entropy computation. The maximum speedup is achieved when $R = 1000$. In this case, the proposed FPGA implementation is respectively 111.47 times and 18.69 times faster than a single CPU core and a 6-core CPU. This high performance is achieved by the massive amount of parallelism available in hardware. In CPU, 6-cores are used for $T_{X\to Y}$ and $T_{Y\to X}$, so there are actually 3 pipes for both $T_{X\to Y}$ and $T_{Y\to X}$. In comparison, our hardware solution could deploy 24 pipes for $T_{X\to Y}$ and 24 for $T_{Y\to X}$, which can deliver higher performance than that for CPU.

In addition, FPGA has great energy efficiency compared with CPU. We measured the run-time power of the host computer using a power meter, and compared the energy consumption of CPU-only implementation and that of FPGA implementation for computing transfer entropy. It is discovered that on average, the FPGA implementation consumes 3.80% of the energy consumed by the CPU-only implementation. In other words, the FPGA is about 26.31 times energy efficient than the CPU when computing transfer entropy.

C. Performance versus Resolution - Forex Data

As a case study, we use real time series in our performance test - historical foreign exchange data. The test time series are EUR-JPY and GBP-USD rates on 30 Apr 2014. There are 75000 records in each time series.

We run the performance test using the same settings as the previous test for random numbers. The performance results are shown in Figure 5. The highest speed-up is achieved when resolution is 800: the FPGA is 111.29 times faster than one Xeon CPU core, and 18.74 times faster than a 6-core Xeon CPU. As seen from Figure 4 and Figure 5, performance is unrelated with the test time series used. This is because the transfer entropy computation takes (joint) probability distribution tables as inputs, not the original time series. We can see the proposed FPGA system is able to deliver high performance for real applications.

D. Bottleneck

Although the FPGA has already shown impressive speed-up against many-core CPU, it still has the potential to be even faster. We discover that the bottleneck of our system is CPU-FPGA bandwidth.

In our tests, there are 48 computing pipes ($K = 24$) in the system running at 80MHz. When resolution = 1200, the computing time should be 0.9s, according to (25). However, the actual time measured in the experiments is 1.41s. Therefore, the kernel is clearly bounded by I/O bandwidth. When resolution = 1200, the elements in tables $N(y_{n+1}, x_n, y_n)$ and $N(x_{n+1}, x_n, y_n)$ could be represented using 4-bit unsigned integer. So the total data size for the two tables are $2 \times 1200^3 * 4/8 \approx 1.61GB$. Using (26), we can estimate the actual bandwidth to be around 1.14GB/s. This is the same in other experiments using different resolutions, where the actual bandwidth is about 1.1-1.3GB/s.

Since there are 48 computing pipes, on each cycle the FPGA kernel needs 24 bytes of data from CPU. The kernel runs at 80MHz, so the I/O bandwidth requirement is 1.92GB/s. In our hardware platform, the FPGA card is connected to a CPU via the PCI-E 2.0 x8 interface with a theoretical speed of 4GB/s in each direction. However, as there are various overheads in the PCI-E channel, the actual bandwidth of PCI-E 2.0 x8 is about 3GB/s. Futhermore, due to the limitation of the PCI-E interface chip on the FPGA card, the actual bandwidth in the experiments is about 1.3GB/s. As a result, the FPGA is actually waiting for data.

Here we offer a theoretical prediction. If the interface chip on an FPGA board could fully support PCI-E interface, we will have about 3GB/s bandwidth available, so the system is no longer bounded by I/O bandwidth. In this case, the FPGA could be about $\frac{1.92}{1.3} \approx 1.37$ times faster than now, which means 25.61 times faster than the 6-core Xeon CPU.

A better interface chip could be available in the future. Besides, for the current hardware platform, one possibility would be exploring more advanced data compression techniques than bit-width narrowing to further reduce bandwidth requirement. In this case, the CPU could send compressed tables $N(y_{n+1}, x_n, y_n)$ and $N(x_{n+1}, x_n, y_n)$ to FPGA in order to save bandwidth. As shown in Table IV, there are still plenty of logic resources available, so it is possible to build a decompressor in FPGA.

VII. Conclusion and Future Work

This paper features the first reconfigurable computing solution to transfer entropy computation. A novel probability estimation technique based on Laplace's rule of succession is used to estimate the probability distributions used for computing transfer entropy. The CPU-FPGA I/O overhead is reduced by exploiting on-chip BRAM to store the data tables which are frequently read. Bit-width narrowing is also used to further reduce bandwidth requirement and save BRAM resources. In addition, we use mixed-precision optimisation to find the best trade-off between accuracy and hardware resource usage, achieving double precision at a moderate logic cost.

We implement the kernel on Maxeler MAX3 platform with a Xilinx Virtex-6 SX475T FPGA. The proposed system is evaluated using random numbers and historical Forex data. The experimental results show that the hardware solution achieves up to 111.47 times speedup over a single Xeon CPU core and 18.69 times speedup over a 6-core Xeon CPU. The work shows the potential of reconfigurable computing for calculating transfer entropy.

Future work includes using advanced data compression techniques to further reduce I/O overhead, introducing run-time reconfiguration to further optimise efficiency, customising the hardware architecture for various applications such as bioinformatics and data mining, and exploring methods for automating such customisation.

Acknowledgments

This work is supported in part by the Lee Family Scholarship, by the United Kingdom Engineering and Physical Sciences Research Council, by the European Union Seventh Framework Programme under grant agreement number 257906, 287804 and 318521, by the Maxeler University Programme, and by Xilinx.

References

[1] T. Schreiber, "Measuring information transfer," *Phys. Rev. Lett.*, vol. 85, pp. 461–464, Jul 2000.

[2] C. J. Honey, R. Kötter, M. Breakspear, and O. Sporns, "Network structure of cerebral cortex shapes functional connectivity on multiple time scales," *Proceedings of the National Academy of Sciences*, vol. 104, no. 24, pp. 10 240–10 245, 2007.

[3] G. Ver Steeg and A. Galstyan, "Information transfer in social media," in *Proceedings of the 21st International Conference on World Wide Web*, ser. WWW '12. ACM, 2012, pp. 509–518.

[4] J. Li, C. Liang, X. Zhu, X. Sun, and D. Wu, "Risk contagion in Chinese banking industry: A transfer entropy-based analysis," *Entropy*, vol. 15, no. 12, pp. 5549–5564, 2013.

[5] D. Gembris, M. Neeb, M. Gipp, A. Kugel, and R. Männer, "Correlation analysis on GPU systems using NVIDIA's CUDA," *J. Real-Time Image Process.*, vol. 6, no. 4, pp. 275–280, Dec. 2011.

[6] Y. Lin and G. Medioni, "Mutual information computation and maximization using GPU," in *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, June 2008, pp. 1–6.

[7] C. R. Castro-Pareja, J. M. Jagadeesh, and R. Shekhar, "FPGA-based acceleration of mutual information calculation for real-time 3D image registration," in *Electronic Imaging 2004*. International Society for Optics and Photonics, 2004, pp. 212–219.

[8] C. Guo, W. Luk, and S. Weston, "Pipelined reconfigurable accelerator for ordinal pattern encoding," *IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, 2014.

[9] K. P. Murphy, *Machine learning: a probabilistic perspective*, 2012.

[10] C. D. Manning and P. Raghavan, *Introduction to information retrieval*, 2012, vol. 1.