Imperial College London

Imperial College London

Department of Computing

Accelerating Transfer Entropy Computation

Author:
Shengjia Shao

Supervisors:
Prof. Wayne Luk
Prof. Stephen Weston

September 5, 2014

Submitted in part fulfilment of the requirements for the degree of MRes in Advanced Computing of Imperial College London

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 behaviour originating in the source series, by excluding dependency on a common external factor. With this advantage, 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 for CPU. This project presents the first reconfigurable computing solution to accelerate the 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.

Acknowledgements

I would like to thank Prof. Wayne Luk for his continuous support during my master academic year, and his insightful suggestions on the conference paper for this project. I am also grateful to Prof. Stephen Weston for his initial idea on transfer entropy. Last but not least, many thanks to Ce Guo for his contributions to Chapter 3 and the helpful discussions we had on technical details.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In many research areas, 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 a time series of signals, and analysing the time series. The simplest statistical measure of dependency is correlation, however, it does not necessarily imply causality. Information-based measures are considered to be more advanced. Transfer entropy is one such measure and is an asymmetric information theoretic measure designed to capture directed information flow between variables [1].

The interesting properties of transfer entropy make it ideal for analysing interactions between variables in a complex system. However, computing transfer entropy is challenging, due to its computationally intensive nature. In this project, we address the problem of computing transfer entropy efficiently and accurately.

1.1 Motivation

The sub-prime mortgage crisis in 2008 resulted in a financial crisis with remarkable global economic impact. One lesson learned there is that the default of a major bank is capable of triggering a domino effect in the banking industry. Since then, the management of inter-bank contagion risk has become very important for both financial institutions and regulators.

The prerequisite of managing interbank exposure is its quantitative measurement, i.e. the interbank risk. Unfortunately, it is difficult to quantify the risk accurately, since it often requires confidential business information. As a result, we could only make estimates based on limited information available.

Traditionally, maximum entropy estimation is used. In this method, aggregated interbank assets and liabilities disclosed in balance sheets are the input information, and the exposure matrix is then derived by maximising its entropy. The major drawback of this method is that the matrix is based on very limited disclosed data, so maximising its entropy will result in a distorted view of risk. In a recent paper, transfer entropy is used to determine the interbank exposure matrix of 16 banks in China, followed by simulation to investigate risk contagion in Chinese banking industry [6].

In addition, transfer entropy has also been successfully applied to various areas such as neuroscience [7], data mining [8], etc. The wide application of transfer entropy creates great demand for its fast and accurate evaluation. Consequently, many researchers in different fields could benefit from this project.

1.2 Objectives

The main aim of this project is to offer a solution to the efficient and accurate computation of transfer entropy, which would be useful for researchers in various areas. In particular, we identify the following objectives:

- To deal with the common situations in which only limited samples of time series data are available
- To explore the application of hardware acceleration techniques to transfer entropy computation
- To evaluate the proposed solution in real cases

1.3 Challenges

We believe that the following challenges need to be addressed when developing the our solution to transfer entropy computation:

- **Limited Time Series Data**: 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, and the straightforward way to compute transfer entropy may lead to inaccurate results.
- **High Resolution**: Transfer entropy is computationally intensive for real-world data series. The time complexity of transfer entropy computation is $O(R^3)$, with R stands for resolution. As resolution increases,

granularity decreases so that transfer entropy with improved accuracy can be obtained. However, higher resolution means much more iterations to be run, which could lead to long execution time.

- **Hardware Limitation**: Transfer entropy computation is parallelisable, so it has the potential to benefit from hardware acceleration techniques, such as reconfigurable computing. However, the limited CPU-FPGA bandwidth and limited FPGA logic resource create challenges for design and implementation.

In the next section we will clarify how these challenges are addressed in this project.

1.4 Contributions

In this project, we develop a novel method to estimate the probability distributions used in transfer entropy computation, which improves accuracy when the time series data available is limited. Also, we present a dataflow architecture for computing transfer entropy and implement it on a commercial FPGA, which greatly boosts performance.

In particular, the contributions of this project 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, addressing the first challenge. Our method eliminates the problem of zero probability that previous methods suffered from.
- To address the second challenge, we develop a novel hardware architecture for computing transfer entropy. This benefits from the huge parallelism achievable on FPGA, the resulting proposed system has the potential to be magnitudes faster than many-core CPU. In addition, our architecture is flexible to make full use of hardware resources, and is able to support ultra large resolution by using multiple FPGAs.
- We optimise memory allocation to effectively reduce I/O requirements. In addition, bit-width narrowing is used to cut down BRAM usage, and to further reduce I/O overhead. Mixed-precision optimisation is used to gain double precision accuracy with a modest amount of hardware resources. These optimisations effectively address the third challenge.
- 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.

To the best of our knowledge, we are the first to apply reconfigurable computing techniques to transfer entropy computation. As a summary of our achievements, a paper "*Accelerating Transfer Entropy Computation*", by Shengjia Shao, Ce Guo, Wayne Luk and Stephen Weston has been submitted to *the 2014 International Conference on Field-Programmable Technology* (ICFPT 2014). The paper is currently under review.

1.5 Organisation of this dissertation

The rest of this dissertation is organised as follows:

- Chapter 2 covers basic background material for transfer entropy.
- Chapter 3 presents our novel method for probability estimation.
- Chapter 4 describes the proposed hardware architecture.
- Chapter 5 provides experimental evaluation and discussion.
- Chapter 6 presents conclusion and probabilities for future work.

Chapter 2

Background

In this chapter, we provide a brief introduction to transfer entropy, focussing on the concept, computation and application. We also review existing work on the hardware acceleration of time series analysis. Then we introduce the essentials of dataflow computing and the Maxeler platform used in this project. Finally, we present the basic background of frequentist probability and Laplace's rule of succession.

2.1 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:

$$X = \{x_1, x_2, \cdots, x_T\} \tag{2.1}$$

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:

$$h_1 = -\sum_{x_{n+1},x_n,y_n} p(x_{n+1}, x_n, y_n) \log_2 p(x_{n+1}|x_n, y_n) \tag{2.2}$$

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

$$h_2 = -\sum_{x_{n+1},x_n,y_n} p(x_{n+1}, x_n, y_n) \log_2 p(x_{n+1}|x_n) \tag{2.3}$$

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:

$$T_{Y\to X} = h_2 - h_1$$
$$= \sum_{x_{n+1},x_n,y_n} p(x_{n+1},x_n,y_n)\log_2\left(\frac{p(x_{n+1}|x_n,y_n)}{p(x_{n+1}|x_n)}\right) \tag{2.4}$$

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

$$T_{X\to Y} = \sum_{y_{n+1},x_n,y_n} p(y_{n+1},x_n,y_n)\log_2\left(\frac{p(y_{n+1}|x_n,y_n)}{p(y_{n+1}|y_n)}\right) \tag{2.5}$$

2.2 Computing Transfer Entropy

Using the definition of conditional probabilities, (2.4) and (2.5) can be rewritten as:

$$T_{Y\to X} = \sum_{x_{n+1},x_n,y_n} p(x_{n+1},x_n,y_n)\log_2\left(\frac{p(x_{n+1},x_n,y_n)p(x_n)}{p(x_n,y_n)p(x_{n+1},x_n)}\right) \tag{2.6}$$

$$T_{X\to Y} = \sum_{y_{n+1},x_n,y_n} p(y_{n+1},x_n,y_n)\log_2\left(\frac{p(y_{n+1},x_n,y_n)p(y_n)}{p(x_n,y_n)p(y_{n+1},y_n)}\right) \tag{2.7}$$

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 (2.6) and (2.7).

In preprocessing, we first go through the time series, counting the number of occurrence 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 occurrence 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:

$$\Delta = \frac{MAX - MIN}{R-1} \tag{2.8}$$

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 will require 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 (2.6) and (2.7). If $R = 100$, there will be 10^6 elements to be accumulated to derive $T_{X \to Y}$. Therefore, the time complexity of transfer entropy computation is $O(R^3)$. As time complexity grows rapidly with R, computing transfer entropy is highly demanding for CPU.

2.3 Applications of Transfer Entropy

Transfer entropy is introduced by Thomas Schreiber in his paper "*Measuring Information Transfer*" in 2000 [1]. In this paper there is an intuitive example to illustrate its usage on detecting causal directions - trying to figure out whether heart beat results in breath or vice versa.

Figure 2.1 shows two time series, the breath rate and the heart rate of a sleeping human. The data is sampled at 2Hz, and normalised to zero mean and unit variance. Using the procedure above, the transfer entropy from heart to breath ($T_{heart \to breath}$) and that in the opposite direction ($T_{breath \to heart}$) can be calculated. Also, we could use other measures, such as mutual information. In Figure 2.2, transfer entropy (solid line and dotted line) is compared with time delayed mutual information (dashed line). Here, r stands for granularity, so small r means high resolution. This figure shows the value of transfer entropy and mutual information versus granularity.

As can be seen from the figure, $T_{heart \to breath}$ is always larger or equal to $T_{breath \to heart}$. Therefore, based on transfer entropy analysis, we could say heart beat results in breath. In contrast, time delayed mutual information failed to distinguish causality because the two dashed lines corresponding to $M_{heart \to breath}$ and $M_{breath \to heart}$ overlap in the figure. So we could see that transfer entropy is indeed superior than traditional information-based measures.

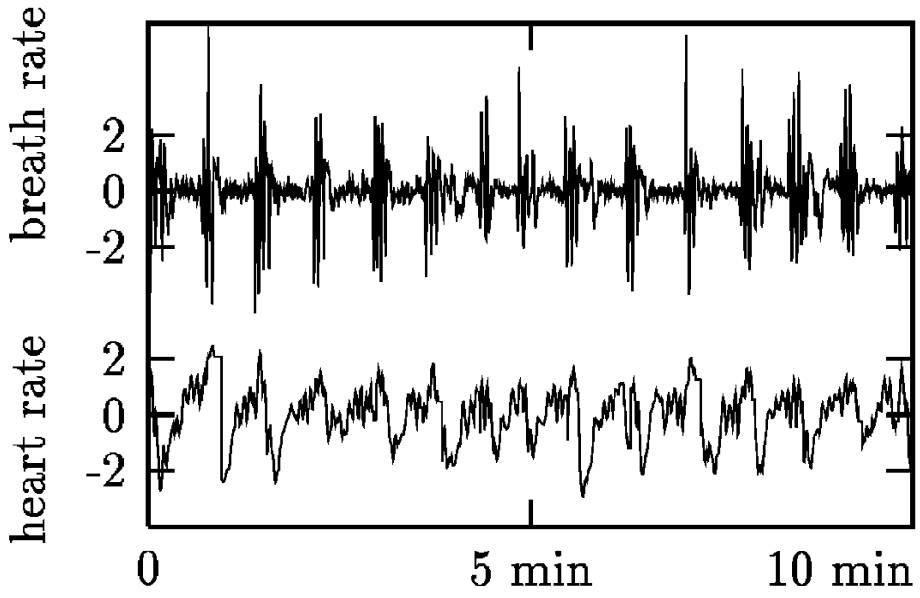

Figure 2.1: time series of the breath rate (upper) and instantaneous heart rate (lower) of a sleeping human. The data is sampled at 2 Hz. Both traces have been normalised to zero mean and unit variance [1].

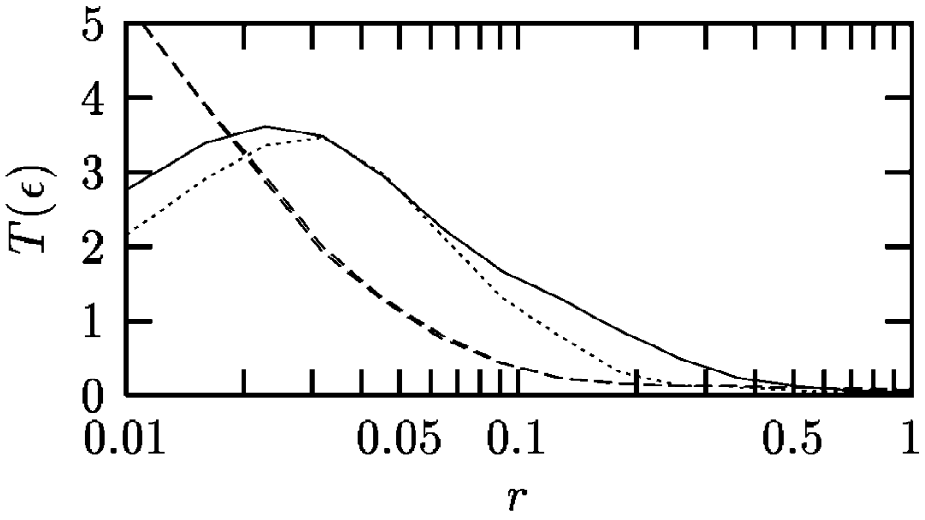

Figure 2.2: Transfer entropies $T_{heart \to breath}$ (solid line), $T_{breath \to heart}$ (dotted line), and time delayed mutual information M($\tau = 0.5s$) (directions indistinguishable, dashed line). r is granularity [1].

Since its introduction in 2000, transfer entropy has been widely applied to many research areas. Here we list three examples.

Honey et al. use transfer entropy to analyse the functional connectivity of different areas in the cerebral cortex [7]. 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 [8]. 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.

Li et al. use transfer entropy to analyse the interaction of banks in the financial market [6]. 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.

2.4 Accelerating Time Series Analysis

Transfer entropy is a metric used in *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 general, the hardware acceleration of time series analysis is still an emerging area.

Gembris et al. use GPU to accelerate correlation analysis [9]. A GPU version of correlation computation is developed using NVIDIA's CUDA language and implemented on GeForce 8800 GTX GPU. The proposed GPU

solution achieved up to 15 times speed-up against an Intel Pentium 4 CPU running at 3GHz. They also evaluate FPGA performance for correlation analysis. It is reported that a FPGA version implemented on Xilinx Virtex-2 FPGA is up to 10 times faster than the 3GHz Pentium 4 CPU.

Castro-Pareja, Jagadeesh, and Shekhar present a FPGA implementation for mutual information computation in 2004 [10]. Mutual information computing logic is implemented on Altera Stratix I FPGA using 32-bit fixed-point numbers. The proposed FPGA system is up to 86 times faster than a software version running on a 1GHz Intel Pentium III CPU. Lin and Medioni compute mutual information using NVIDIA's GPU [11]. The system is developed using CUDA and tested on GeForce 8800 GTX. In the experiments, GPU achieved 170 times speed-up for computing mutual information and 400 times speed-up for its derivative, compared with a quad-core Intel Xeon CPU running at 2.33GHz.

Guo and Luk design a FPGA accelerator for ordinal pattern encoding, a statistical method for analysing the complexity of time series. They apply it to the computation of permutation entropy [12]. The FPGA system is implemented on a Xilinx Virtex-6 FPGA, and is integrated with a quad-core Intel i7-870 CPU running at 2.93GHz. Experimental results show that the CPU+FPGA hybrid system is up to 11 times faster than the CPU-only solution.

These previous work demonstrates that time series analysis has the potential to benefit from hardware acceleration. As for transfer entropy, as it is a new statistical metric, we are not aware of any published work on its hardware acceleration. Therefore, this work could be the first work on accelerating transfer entropy computation using reconfigurable computing techniques.

2.5 Reconfigurable Computing

Traditionally, software programs are run on CPU. A program, written in programming languages such as C/C++, is compiled into a series of instructions. When running the program, instructions are loaded into computer memory, and read by CPU. CPU decodes and executes the instruction, finally writing results back to memory. This instruction-based model is inherently sequential and the common bottleneck is memory access speed. To address this problem, contemporary CPUs have sophisticated caches and branch prediction logic. While CPU is efficient for everyday jobs, its performance may not be good enough for media and scientific computing tasks, which creates space for hardware acceleration solutions. These solutions deploy non-CPU hardware, such as GPU or FPGA, to do the computation.

Compared with CPU, GPU and FPGA have totally different architectures and computing paradigms. Therefore, they are naturally free from many bottlenecks that hinder CPU performance. Consequently, hardware acceleration could result in huge performance gain, which means being magnitudes faster than the CPU-only solution.

Reconfigurable Computing, also known as *Dataflow Computing*, is a hardware acceleration technique which is able to deliver high performance. It has been an active research area for decades, and commercial solutions are emerging in recent years. In dataflow computing, a *Dataflow Graph* (DFG) is built based on the program to be run, which is a graphical representation of the flow of data through the system. For the ease of understanding, the DFG is similar to the flowchart of the program - the nodes in the DFG are almost identical to the instructions in the software program. Then the DFG is mapped onto hardware. The key difference between dataflow computing and CPU computing is that when running the system, there is no need of fetching and decoding instructions, because DFG has already defined what to do. The dataflow computing system works like a production line with input data streamed in and output results streamed out. This streaming model enables the dataflow system to be highly pipelined to boost throughput. It is not surprising that a dataflow computing system running at 100MHz could outperform an 8-core CPU running at 3GHz.

Typically, a dataflow computing system is composed of host CPU(s) and *Reconfigurable Device(s)*. The reconfigurable device is often based on Field Programmable Gate Array (FPGA), a digital integrated circuit designed to be configured by a customer after manufacturing (field-programmable). Basically, a FPGA could be viewed as a set of basic building blocks which could be used to build a digital circuit for a certain task. This means FPGA could be configured similar to an Application-Specific Integrated Circuit (ASIC) to deliver very high performance. The unique advantage of FPGA lies in the fact that customising a FPGA comes with zero cost while building an ASIC needs millions of money. FPGA enables the user to implement any logic, provided that there are enough resources. In short, FPGA has the potential to bring ASIC-like performance at very low cost, making dataflow computing a very promising solution for high performance computing.

Contemporary FPGAs have massive resources of the following four types:

- **Look-Up Table (LUT)**: LUT is used to implement digital logic. Each LUT supports a small digital logic function, such as a 6-input function. Typically there are hundreds of thousands of LUTs available in one FPGA.

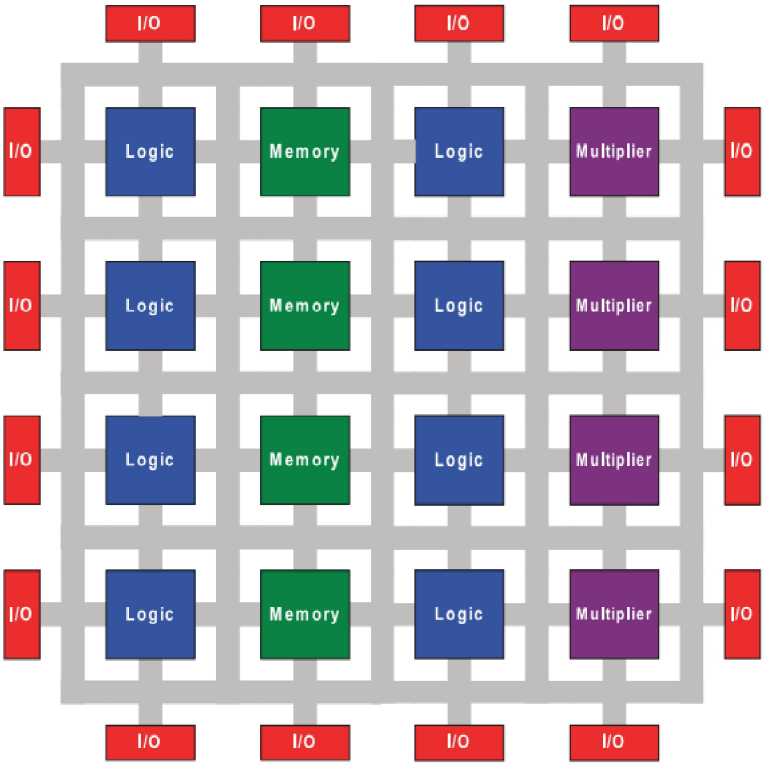

Figure 2.3: General Architecture of FPGA [2]

- **Flip-Flop (FF)**: FF is a small memory cell which is used to temporarily hold the value of an internal node. In a FPGA, FF usually comes together with LUT to enable deep pipelining.
- **Digital Signal Processor (DSP)**: DSP is a digital circuit designed exclusively for some commonly-used arithmetics, such as floating-point addition and multiplication. DSP blocks are embedded in FPGA for the efficient implementation of these arithmetics.
- **Block RAM (BRAM)**: BRAM provides on-chip storage for FPGA. It is used to store a small amount of data (usually several MBs) which are frequently used. Accessing BRAM is much faster than accessing off-chip memories.

Figure 2.3 shows a general architecture of FPGA [2]. Logic blocks (LUT and FF), memory (BRAM) and multiplier (usually DSP) are connected via programmable switch matrices. To map an application to FPGA it is essentially to configure all these programmable elements according to the program. It is worth to pointing that not all things in the program are suitable for FPGA. In fact, *Software/Hardware Partition* is used to divide

the program into software and hardware parts. The software part is the code which is more suitable for CPU, such as control intensive sequential code or random memory access. On the other hand, the parallelisable code which can be efficiently mapped to FPGA will be represented by the DFG, which will be synthesised to FPGA configuration and then loaded to FPGA. When running the system, the host CPU executes the software part while the FPGA executes the hardware part. In this manner, dafaflow computing exploits both the advantages of CPU and those of the FPGA to achieve best performance.

Traditionally, the dataflow design is specified using *Hardware Description Languages* (HDLs), such as VDHL or Verilog. The developer describes the system at *Register Transfer Level* (RTL), i.e. the dataflow of signals between registers and the operations on these signals. RTL design is the typical practice in digital circuit design, and the synthesis process from HDL description to FPGA configuration is well supported. FPGA vendors provide tools for users to configure FPGA using HDL, such as Quartus for Altera FPGAs and ISE for Xilinx FPGAs [13] [14].

The RTL design enables the developer to specify very low-level details of the system. Consequently, the developer could do extensive low-level optimisations to improve performance. However, as the hardware system could be rather complex, using HDL to specify a circuit is often very time-consuming and error-prone. To address this problem, there have been decades of efforts on *High Level Synthesis*, which means to transform a program written in a high level language, such as C, to RTL descriptions. In recent years, the academic research on high level synthesis has resulted in commercial solutions. Xilinx offers Vivado Design Suite which could compile C, C++ or System C into RTL-level description, then traditional tools could be used to map RTL code to FPGA [15]. Meanwhile, Altera now supports OpenCL. OpenCL is a free and open programming model for programming heterogeneous systems such as FPGA and GPU. OpenCL allows the use of a C-based programming language for developing hardware accelerating solutions, which is much easier than using HDL.

Figure 2.4 shows the design flow of dataflow computing [3]. The developer describes the system in a certain language, low-level or high-level, which is the input file of the tool-chain. The system design is then synthesised into a netlist for placement and routing. The placement and routing tool tries to allocate the logic blocks, memories, DSPs and interconnections based on the netlist. The placement and routing result is further simulated to check whether it meets timing requirement set by the developer. If successful, FPGA configuration will be generated.

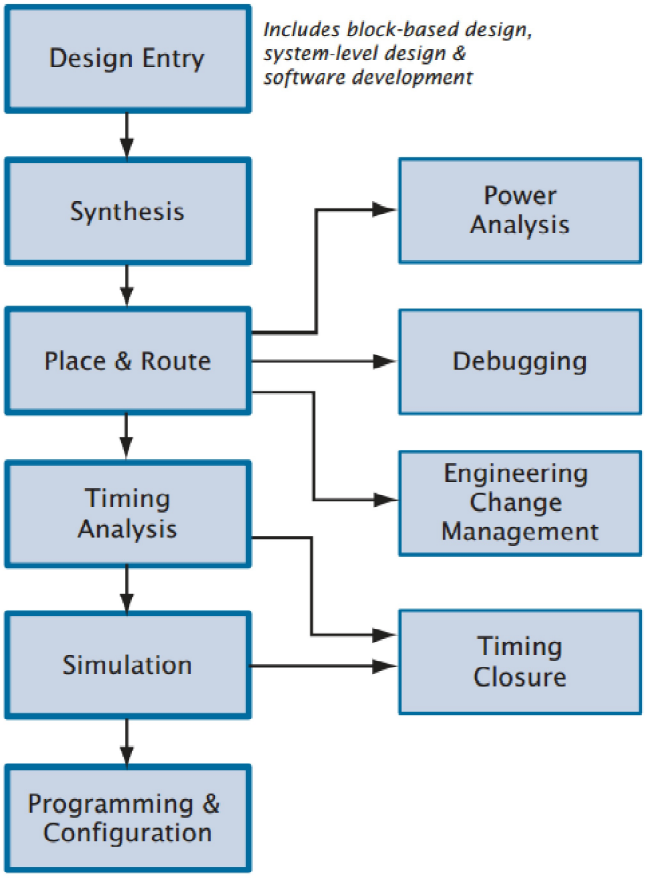

Figure 2.4: Dataflow Computing Design Flow [3]

2.6 Maxeler Dataflow Computing System

In this project, we design a dataflow system for transfer entropy computation. The hardware platform used in our experiments are provided by Maxeler Technologies. Here we make a brief introduction to the Maxeler System and its design flow.

2.6.1 System Architecture

Maxeler offers several dafaflow computing systems ranging from desktop workstation to rack nodes, targeting different applications. Figure 2.5 shows the architecture of MPC-C System, which is used in this project [4].

In the MPC-C system there are four *Dataflow Engines* (DFEs) and two 6-core Intel Xeon CPUs. Each DFE is composed of one Xilinx Virtex-6 FPGA, 48GB DRAM and peripheral circuits. The four DFEs are connected to CPU via PCI-Express 2.0 interface. Also, they are connected together via a special circular bus called MaxRing. The communication between CPU and DFE is via PCI-E, and that between DFEs is via MaxRing. The whole system is packed in a standard 1U case.

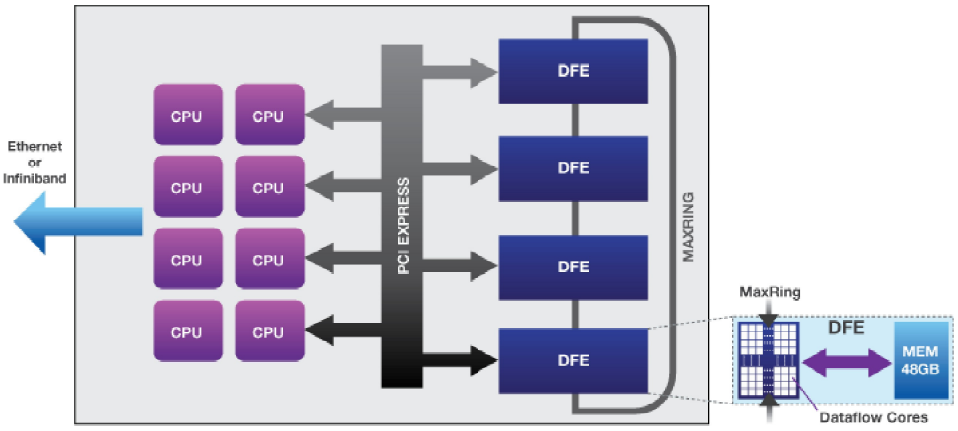

Figure 2.5: System Architecture of Maxeler MPC-C [4]

As can be seen from the above figure, the DFE engines appear as expansions to a standard X86 computer via PCI-E interface. This feature enables the dataflow computing system to be compatible with conventional operating system and software, making them easier to use.

2.6.2 Design Flow

While it's fine to specify a dataflow design targeting Maxeler platform using standard HDL languages, Maxeler provides a powerful high level language, MaxJ, to making things easier for developers. MaxJ is a high level metalanguage based on Java, which is designed specifically for dataflow computing.

Figure 2.6 shows the design flow with Maxeler. When programming with MaxJ, the developer needs to describe the following three parts:

- **Kernel**: The core of the system where computation happens. It is similar to the DFG of the program.
- **Manager**: The I/O interface between the kernel and the rest of the system. It connects Kernel with CPU, BRAM, other DFEs, etc.
- **CPU Application**: The application is written in C code to integrate the dataflow engine with the software part of the program. It is executable in Linux environment.

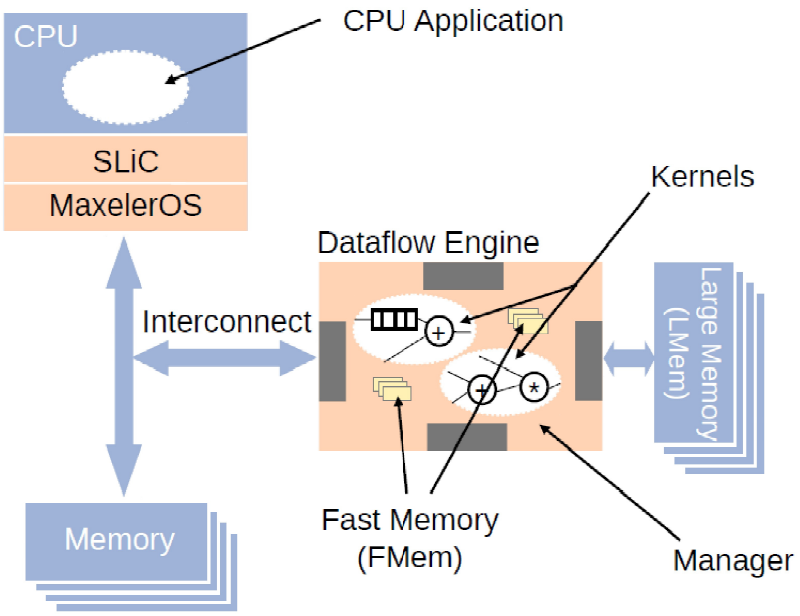

Figure 2.6: Dataflow Design with Maxeler [5]

To compile MaxJ code into FPGA configuration, Maxeler offers MaxCompiler [5]. MaxCompiler first synthesises MaxJ files into VHDL, and then deploy FPGA-vendor tool-chain for the rest of the compiling process.

2.7 Optimising Dataflow Design

The power of dataflow computing mainly lies in its flexibility. In FPGA, everything is programmable. Therefore there exists opportunities to improve performance by dedicated customisation. Developers could tailor both the software and the hardware in order to fit them perfectly for the best performance.

As mentioned earlier, reconfigurable devices, such as FPGA, has massive amount of parallel resources (LUT, FF, DSP, BRAM). Therefore, efficiently exploiting the parallel resources is essential for achieving high performance. There exist many optimisation techniques for dataflow designs, we will briefly review the most commonly used ones in this section.

2.7.1 Algorithm Optimisation

Sometimes a straightforward implementation of a program on dataflow hardware may not lead to good performance. This is because the application must be parallelisable in order to fit the parallel resource on FPGA. The key prerequisite of being parallelisable is no data dependency - if each basic step in the program depends on the previous one, the application must be executed sequentially and it's impossible to make use of the parallel resource. As data dependency is very common among applications, one often needs to modify the algorithm in the application to eliminate such dependency, which can be seen as optimising the algorithm.

Even if there is no data dependency, algorithm optimisation is still useful to improve performance. In many cases, the computationally intensive part of an application is actually standard numerical computations, such as matrix algebra or a numerical equation solver. Then algorithm optimisation could be carried out by selecting a suitable parallel algorithm. Therefore, replacing the original algorithm in the application by a parallel algorithm for the same task often boosts performance significantly. Sometimes there may be no parallel algorithm available for the target computation, so the developer may try to invent one himself, which is an important research area. In addition, some algorithms, such as iterative algorithms, allow developers to trade speed against accuracy. In such cases algorithm optimisation could be achieved by choosing a good balancing point.

When optimising an algorithm, it is important to make sure the modification does not affect output results, or the degradation of accuracy is within an acceptable range, so that correctness is preserved.

2.7.2 Mixed-Precision Optimisation

In dataflow computing, as the datapath on FPGA is fully programmable by the user, there exists a very powerful optimisation strategy called *Mixed-Precision Optimisation*, which is the customisation of the types and bit-sizes of the variables in the datapath.

The complexity of the digital logic for a certain task is closely related with the data type used. In general, integer computation is the simplest, fixed-point numbers need some more effort, and floating-point numbers require a considerable amount of hardware resources. In addition, a less-complex design will usually lead to better performance because it could probably run at a higher frequency. Some times the standard data types, such as `int`, `float` or `double` may be overqualified for the computing task, and mixed-precision optimisation allows the developer to create a custom data type, such as 20-bit integer, to reduce logic complexity and to improve performance. Since datapath is fixed in CPU or GPU hardware, mixed-precision optimisation cannot be implemented in CPU or GPU, which makes it a unique advantage of dataflow computing.

To implement mixed-precision optimisation, it is necessary to know the required precision of the output results. The goal of mixed-precision optimisation is to use simplest data representation to achieve the required result precision. The first step is to investigate the dynamic range of the variables in the system. This could be done by simulating the system and checking the upper bound and lower bound of the internal variable values. The input data sets used in simulation must be representative. Once the dynamic range of a variable is found, one could choose suitable data representation for it. In the case of large dynamic range, floating point representation could be used. For small dynamic range variables, fixed point representation should be enough. Bit width and number of exponent bits and mantissa bits could then be derived from the dynamic range. After changing data representation, simulation should be run again to ensure the requirement of output result precision is still satisfied.

Mixed-precision optimisation could improve performance and reduce logic usage at the same time, making it a very powerful optimisation technique. Usually, one should always seek mixed-precision optimisation opportunities when designing the dataflow system.

2.7.3 Run-time Reconfiguration

Traditionally, FPGA configuration is specified at compile-time and remains unchanged when running the system. *Run-time Reconfiguration* is an emerging technique which reconfigures FPGA at run-time to further improve performance. For example, removing idle functions is an effective way of run-time reconfiguration. Usually, not all parts of the dataflow design are active at every stage. Therefore, by removing the idle portions of a design and using that space to increase parallelism of the active functions, one can make full use of the logic resources available on FPGA to boost performance.

Run-time reconfiguration could be applied with other optimisations at the same time, and is currently an active research area. The challenges of run-time reconfiguration include: (1) to identify reconfiguration opportunities; (2) to generate run-time solutions which preserve functional correctness and accuracy; (3) to reduce the overhead of applying FPGA reconfiguration at run-time [16].

2.8 Frequentist Probability

Frequentist Probability is a commonly-used interpretation of probability. It defines an event's probability as the limit of its relative frequency in a large number of trials, which is a straightforward interpretation:

$$p(X) = \lim_{N \to \infty} \frac{N(X)}{N} \tag{2.9}$$

Here, N is the number of trials, $N(X)$ is the number of occurrence of an event in the trials. If an infinite number of trials can be done, (2.9) will give the probability.

In reality, one can only carry out a large, but finite number of trials, as is the case for observing time series. Therefore, the probability calculated in this manner is actually an estimation of frequentist probability:

$$p(X) \approx \frac{N(X)}{N} \tag{2.10}$$

Natually, with N increasing, the estimated probability given by (2.10) will become more and more accurate. Therefore, it will always be helpful to do more experiments so as to record more data samples.

2.9 Laplace's Rule of Succession

Laplace's Rule of Succession, or *the Rule of Succession*, was introduced in the 18 century by the French mathematician Pierre-Simon Laplace. The formula is used to estimate probabilities when there are only few observations, or for events that haven't been observed at all in a finite data sample.

Theorem 1. *Suppose an experiment can be repeated an indefinitely large number of times (trials), with one of two possible outcomes (success or failure). Assume the unknown probability of success is p. If the trials are independent, and all possible values of p are equally likely, then given r successes in n trials, the probability of success on the next trial is* $\frac{r+1}{n+2}$.

Proof. The experiments are Bernoulli trials with parameter p. We denote the sequence of trials by X_i and results by b_i. For the ease of notation, we record 'success' by 1 and 'failure' by 0, so $b_i \in \{0, 1\}$.

We let $S_n = \sum_{i=1}^{n} X_i$. We would like to find the following probability:

$$P(X_{n+1} = 1 | S_n = r)$$

The probability of success (p) is an unknown number. But its value must lie in the range [0, 1], as required by the definition of 'probability'. Let p be a condition of the trials and apply the rule of conditional probabilities, we have

$$\begin{aligned}
&P(X_{n+1} = 1 | S_n = r) \\
&= \int_0^1 P(X_{n+1} = 1 | p, S_n = r) f(p | S_n = r) dp \\
&= \int_0^1 P(X_{n+1} = 1 | p) f(p | S_n = r) dp \\
&= \int_0^1 p f(p | S_n = r) dp
\end{aligned} \tag{2.11}$$

Here, $f(p|S_n = r)$ is the distribution of p given the fact that we succeeded r times in n trials. It reflects the information we get about p by doing experiments. Recall our experiments, we observed r successes in n trials, so the probability of this particular result is

$$P(S_n = r | p) = \frac{n!}{r!(n-r)!} p^r (1-p)^{n-r} \tag{2.12}$$

The Bayes' Theorem states:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.13}$$

Apply Bayes' Theorem to (2.12), we have

$$f(p|S_n = r) \propto P(S_n = r|p)f(p) \propto \frac{n!}{r!(n-r)!}p^r(1-p)^{n-r} \tag{2.14}$$

The probability distribution of p must normalises to 1. So we have:

$$\begin{aligned} f(p|S_n = r) &= \frac{\frac{n!}{r!(n-r)!}p^r(1-p)^{n-r}}{\int_0^1 \frac{n!}{r!(n-r)!}p^r(1-p)^{n-r}dp} \\ &= \frac{p^r(1-p)^{n-r}}{\int_0^1 p^r(1-p)^{n-r}dp} \\ &= \frac{(n+1)!}{r!(n-r)!}p^r(1-p)^{n-r} \end{aligned} \tag{2.15}$$

Finally, substitute (2.15) into (2.11) and calculate the integral:

$$\begin{aligned} &P(X_{n+1} = 1|S_n = r) \\ &= \int_0^1 pf(p|S_n = r)dp \\ &= \int_0^1 p\frac{(n+1)!}{r!(n-r)!}p^r(1-p)^{n-r}dp \\ &= \frac{(n+1)!}{r!(n-r)!}\int_0^1 p^{r+1}(1-p)^{n-r}dp \\ &= \frac{r+1}{n+2} \end{aligned} \tag{2.16}$$

□

Here we make a comparison between Laplace's rule of succession and frequentist probability. Let's say we run n independent trials end up with r successes. From the perspective of frequentist probability, we can estimate the probability of success to be $p = \frac{r}{n}$. However, Laplace's rule of succession suggests an estimate $p = \frac{r+1}{n+2}$ could be more relevent.

Obviously, the two estimates are almost equal for large n. However, if there are only a small number of trials, the latter is often more meaningful. Let's consider an extreme case: assume one tosses a coin for 3 times and gets three heads. Then frequentist probability would suggest the chance of head to be 100% and that of tail to be 0%, which is obviously problematic. In contrast, Laplace's formula shows the chance of head to be 80% and that of tail to be 20%. Although still not so accurate, but they are far better than frequentist probabilities in the sense that Laplace's formula leaves door open for tails while $p = \frac{r}{n}$ does not.

2.10 Summary

In this chapter, we provides essential background for this project.

We began by introducing transfer entropy. Transfer entropy is a metric based on information theory to measure the amount of information transferred between two time series. To calculate transfer entropy, first go through the time series data to calculate (joint) probabilities required (preprocessing) and then evaluate transfer entropy using the (joint) probabilities (computing). Transfer entropy is superior than 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 been successfully applied to various areas.

Then we discussed the hardware acceleration of time series analysis and the dataflow computing technique. Time series analysis could be demanding for CPU and has the potential to run faster on non-CPU hardware platforms. There has been previous work on accelerating various time series metrics, such as correlation and mutual information, using GPU or FPGA platforms. We believe transfer entropy could also benefit from hardware acceleration, although we are not aware of any published work on it. Therefore, this project could be the first work on accelerating transfer entropy computation. We introduce the concept of dataflow computing and its key hardware - FPGA. We deploy Maxeler MPC-C system in this project. This system is composed of Intel Xeon CPU and Xilinx Virtex-6 FPGAs. Maxeler also provides MaxJ language and MaxCompiler tool chain for developing dataflow designs. We then make a brief introduction on the optimisation techniques on dataflow computing, such as algorithm optimisation, mixed-precision optimisation and run-time reconfiguration.

Finally, we present the background on frequentist probability and Laplace's rule of succession. Frequentist probability defines an event's probability as the limit of its relative frequency in a large number of trials. However, when the number of trials is limited, the estimated probability given by frequentist probability may be problematic. In this case, Laplace's rule of succession could be used to make a more accurate prediction.

Chapter 3

Probability Estimation

This chapter presents a new method, based on Laplace's rule of succession, for estimating probabilities used in transfer entropy computation. We will first go through the traditional frequentist statistics method and point out its problem, and then present our new method.

3.1 Frequentist Statistics

The transfer entropy defined in (2.6) and (2.7) 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 the data observed, i.e., the time series sample. Assume that each of these probabilities follows a multinomial distribution. From the perspective of frequentist statistics, a reasonable set of estimates is

$$\hat{p}(x_{n+1}, x_n, y_n) = \frac{N(x_{n+1}, x_n, y_n)}{T-1} \tag{3.1}$$

$$\hat{p}(y_{n+1}, x_n, y_n) = \frac{N(y_{n+1}, x_n, y_n)}{T-1} \tag{3.2}$$

$$\hat{p}(x_{n+1}, x_n) = \frac{N(x_{n+1}, x_n)}{T-1} \tag{3.3}$$

$$\hat{p}(y_{n+1}, y_n) = \frac{N(y_{n+1}, y_n)}{T-1} \tag{3.4}$$

$$\hat{p}(x_n, y_n) = \frac{N(x_n, y_n)}{T} \tag{3.5}$$

$$\hat{p}(x_n) = \frac{N(x_n)}{T} \tag{3.6}$$

$$\hat{p}(y_n) = \frac{N(y_n)}{T} \tag{3.7}$$

where $N(X)$ is the number of occurrence of pattern X in the data, and T

is the length of the time series.

When computing transfer entropy, one can calculate the transfer entropy by replacing the probabilities in (2.6) and (2.7) with their corresponding estimates. Here we recall (2.6) and (2.7) which are originally introduced in Chapter 2.

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

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

The biggest drawback of frequentist statistics method is the hidden assumption when computing transfer entropy. Relook at (2.6) and (2.7), we could find that all probabilities must be non-zero, or there will be divide-by-zero error or log-zero error. Therefore, in each iteration the computing program will check if any of the probabilities is zero in order to decide whether this iteration will be skipped. Nevertheless, an estimate produced using frequentist probability 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. Unfortunately, this often happens in practice, especially when the resolution is large. The number of observations is usually limited by the experiment equipments, and the problem of incomplete time series data may result in big deviation of the transfer entropy obtained.

3.2 Probability Estimation

To solve the problem of zero probability, we use Laplace's rule of succession to estimate probabilities used in computing transfer entropy, instead of using traditional frequentist probability.

Recall Laplace's rule of succession. The experiment has two possible outcomes (success or failure), the unknown probability of success is p. Assume we carry out n trials and get r successes, then we estimate the p by

$$\hat{p} = \frac{r+1}{n+2} \tag{3.8}$$

Meanwhile, if we use s to denote the number of failures and q to represent

the probability of failure, we could estimate q by

$$\hat{q} = 1 - \hat{p} = 1 - \frac{r+1}{n+2} = \frac{n-r+1}{n+2} = \frac{s+1}{n+2} \tag{3.9}$$

On the other hand, if we use frequentist probability, the results will be:

$$\hat{p}' = \frac{r}{n} \tag{3.10}$$

$$\hat{q}' = \frac{s}{n} \tag{3.11}$$

Compare $\hat{p}, \hat{q}$ with $\hat{p}', \hat{q}'$, we can see 1 is added to the event's number of occurrence, while 2 is added to the number of trials. The key observation here is as follows: we add an imaginary count '1' to the observed number of occurrence to make sure that an event's probability will not be ruled out if it does not happen in our limited trials. Also, we modify the denominator accordingly to make sure the modified probability still sums to one.

With this in mind, we could generalise the above procedure from a Bernoulli trial with two possible outcomes to a general case. We assume the experiment has m possible outcomes:

$$\{b_1, b_2, \cdots, b_m\} \tag{3.12}$$

Assume we carry out n trials, and event b_i occurred r_i times ($i = 1, 2, \cdots, m$).

Naturally, we have

$$\sum_{i=1}^{m} r_i = n \tag{3.13}$$

We denote the probability of event b_i by p_i. Using Laplace's rule of succession, we could estimate p_i by

$$\hat{p_i} = \frac{r_i + 1}{n + m} \tag{3.14}$$

Here, 1 is added to the event's number of occurrence r_i as an imaginary count, and m is added to the denominator accordingly to make sure the modified probability, $\hat{p_i}$, still sums to one:

$$\sum_{i=1}^{m} \hat{p_i} = \sum_{i=1}^{m} \frac{r_i + 1}{n + m} = \frac{(\sum_{i=1}^{m} r_i) + m}{n + m} = \frac{n + m}{n + m} = 1 \tag{3.15}$$

Now we are ready to use Laplace's rule of succession to estimate (joint)

probabilities used for computing transfer entropy. After quantisation, each of the time series X and Y has R possible different values. So the vector (x_{n+1}, x_n, y_n) has R^3 possible different values, (x_n, y_n) has R^2 possible different values, etc. Each of these values could be viewed as an event.

In prepossessing, we go through the time series data, counting the number of occurrence of each pattern (x_{n+1}, x_n, y_n), (x_n, y_n), etc. This is identical to conducting trials. For example, in the case of (x_{n+1}, x_n, y_n), we have

$$m = R^3 \tag{3.16}$$

$$n = T - 1 \tag{3.17}$$

$$r_i = N(x_{n+1}, x_n, y_n) \tag{3.18}$$

(3.16) is because (x_{n+1}, x_n, y_n) has R^3 possible different values; (3.17) is because there are $T-1$ samples, i.e., (x_1, x_0, y_0), ... , (x_T, x_{T-1}, y_{T-1}); (3.18) is by definition.

Substitute (3.16) - (3.18) to (3.14), we have:

$$\hat{p}(x_{n+1}, x_n, y_n) = \frac{N(x_{n+1}, x_n, y_n) + 1}{T - 1 + R^3} \tag{3.19}$$

This is the estimation of joint probability $p(x_{n+1}, x_n, y_n)$ using Laplace's rule of succession. Similarly, we could estimate other (joint) probabilities in the same manner:

$$\hat{p}(y_{n+1}, x_n, y_n) = \frac{N(y_{n+1}, x_n, y_n) + 1}{T - 1 + R^3} \tag{3.20}$$

$$\hat{p}(x_{n+1}, x_n) = \frac{N(x_{n+1}, x_n) + 1}{T - 1 + R^2} \tag{3.21}$$

$$\hat{p}(y_{n+1}, y_n) = \frac{N(y_{n+1}, y_n) + 1}{T - 1 + R^2} \tag{3.22}$$

$$\hat{p}(x_n, y_n) = \frac{N(x_n, y_n) + 1}{T + R^2} \tag{3.23}$$

$$\hat{p}(x_n) = \frac{N(x_n) + 1}{T + R} \tag{3.24}$$

$$\hat{p}(y_n) = \frac{N(y_n) + 1}{T + R} \tag{3.25}$$

As explained earlier, an intuitive interpretation of our treatment is that we set a lower bound of probability (e.g., $\frac{1}{T-1+R^3}$) to each legitimate pattern disregarding the pattern appears in the data. We need to do this because when the time series data is limited, we cannot simply rule out the probability of a certain pattern even if it does not appear in our observations.

With Laplace's rule of succession, the (joint) probabilities used for transfer entropy computation will always be non-zero, which eliminates the divide-by-zero or log-zero problem when using frequentist probabilities.

The lower bound we added decreases with more samples, and only has marginal influence when T is very large. 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 [17]. Similar treatments have been applied to probability inference problems to eliminate side effects of zero probabilities [18].

With (3.19) - (3.25), transfer entropy can be computed using the following two equations:

$$T_{Y\to X} \approx \sum_{x_{n+1},x_n,y_n} \hat{p}(x_{n+1},x_n,y_n)\log_2\left(\frac{\hat{p}(x_{n+1},x_n,y_n)\hat{p}(x_n)}{\hat{p}(x_n,y_n)\hat{p}(x_{n+1},x_n)}\right) \tag{3.26}$$

$$T_{X\to Y} \approx \sum_{y_{n+1},x_n,y_n} \hat{p}(y_{n+1},x_n,y_n)\log_2\left(\frac{\hat{p}(y_{n+1},x_n,y_n)\hat{p}(y_n)}{\hat{p}(x_n,y_n)\hat{p}(y_{n+1},y_n)}\right) \tag{3.27}$$

Note 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 occurrence (N) as inputs.

3.3 Summary

In this chapter, we presented our novel method based on Laplace's rule of succession to estimate probabilities used in computing transfer entropy.

Traditionally, we could use frequentist probability. However, when the samples available are limited, frequentist probability could be zero if a certain pattern does not appear in the samples. This will lead to divide-by-zero or log-zero error as well as introducing bias when computing transfer entropy.

To solve this problem, we derive our probability estimation based on Laplace's rule of succession. We add an imaginary count '1' to the observed number of occurrences to make sure that an event's probability will not be ruled out if it does not happen in our limited trials. Also, we modify the denominator accordingly to make sure the modified probability still sums to one. Our method successfully eliminates the zero-probability problem that frequentist probability suffered from, and could be straightforwardly implemented in software and hardware.

Chapter 4

Hardware Design

In this chapter, 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. We will present the three features in details in Section 4.1 - 4.3. Then we explain how to customise our kernel for different applications. Finally, we provide a performance model for our system.

4.1 Optimised Memory Allocation

The inputs of the FPGA are the number of occurrence tables. Recall the core computation of transfer entropy (equation (3.26) and (3.27)):

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

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

From a computing perspective, they are 3-level nested loops, because 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.

The first optimisation to consider is allocating the number of occurrence tables. Note the CPU-FPGA bandwidth is limited, sending all tables from CPU to FPGA at runtime is obviously very sub-optimal. On the other hand, mapping all tables to on-board DRAM is also unwise because the data still need to be sent from CPU to FPGA during initialisation, which results in a huge initialisation delay.

Table 4.1: Data Access Patterns

Table Name	Size	Data Request	Access Times	Storage
$N(x_{n+1}, x_n, y_n)$	R^3	K per inner loop	Read Once	host
$N(y_{n+1}, x_n, y_n)$	R^3	K per inner loop	Read Once	host
$N(x_{n+1}, x_n)$	R^2	K per inner loop	Read R times	BRAM
$N(y_{n+1}, y_n)$	R^2	K per inner loop	Read R times	BRAM
$N(x_n, y_n)$	R^2	one per middle loop	Read Once	host
$N(x_n)$	R	one per middle loop	Read R times	BRAM
$N(y_n)$	R	one per outer loop	Read Once	BRAM

A compromise solution is to map some of the tables to on-chip BRAM during initialisation while sending others at run-time. The data access patterns of the number of occurrence tables are summarised in Table 4.1.

Here we briefly explain the elements in Table 4.1:

- **Table Name** is the number of occurrence table's name.
- **Size** refers to the number of elements in the table. Here R is resolution.
- **Data Request** is how many items are accessed in each read. Here K is the number of computing pipes. In the kernel we have K pipes for transfer entropy $T_{Y\to X}$ and $T_{X\to Y}$, respectively.
- **Access Times** is how many times each element in the table is accessed. Here R is resolution.
- **Storage** is the place where the data table is stored. 'host' refers to the DDR3 memory in the host computer; 'BRAM' refers to FPGA's on-chip BRAM.

For example, table $N(x_{n+1}, x_n)$ has R^2 elements. In each inner loop iteration, kernel reads K elements from this table. The whole table will be read R times during the computation, and is stored in FPGA's on-chip BRAM. As FPGA has small but fast on-chip BRAM, usually several MB (the Xilinx Virtex-6 SX475T FPGA used in this project has 4.67MB), two kinds of tables are suitable to be mapped to BRAM:

- small tables
- mid-sized tables which are accessed frequently

Consequently, we map $N(x_n)$ and $N(y_n)$ to BRAM because they are small, and $N(x_{n+1}, x_n)$, $N(y_{n+1}, y_n)$ to BRAM because they are mid-sized tables accessed R times. When resolution (R) is around 1000, the total

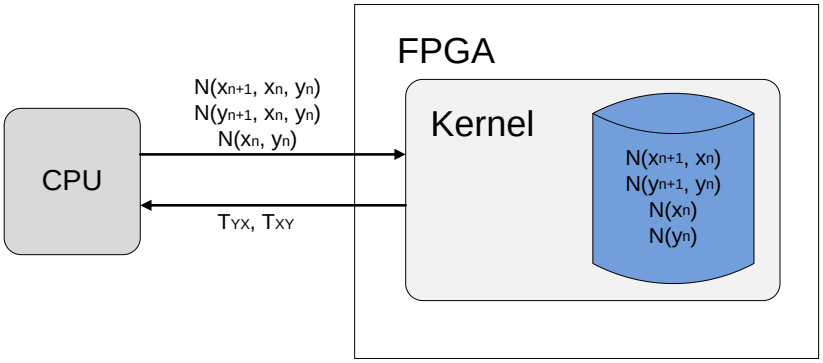

Figure 4.1: System Architecture. Number of occurrence tables $N(x_{n+1}, x_n)$, $N(y_{n+1}, y_n)$, $N(x_n)$ and $N(y_n)$ are mapped to FPGA's BRAM during initialisation. Other tables ($N(x_{n+1}, x_n, y_n)$, $N(y_{n+1}, x_n, y_n)$ and $N(x_n, y_n)$) are streamed at run-time. Results $T_{Y\to X}$ and $T_{X\to Y}$ are sent back to CPU.

size of the 4 tables are just several MB (2.87MB when $R = 1000$, using `uint12` for $N(x_{n+1}, x_n)$ and $N(y_{n+1}, y_n)$, `uint32` for $N(x_n)$ and $N(y_n)$), which would fit in the on-chip BRAM (4.67MB in Xilinx Virtex-6 SX475T). What's more, as the CPU-FPGA bandwidth are magnitudes larger, the overhead of sending MB of data during FPGA initialisation is negligible.

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 run-time. Table $N(x_n, y_n)$ is also streamed to FPGA at run-time 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. As a summary, Figure 4.1 shows the general system architecture with optimised memory allocation.

Figure 4.2 shows the kernel's internal architecture. Inside the kernel, we build $2K$ computing pipes for calculating transfer entropy: K pipes for $T_{Y\to X}$ and the other K pipes for $T_{X\to Y}$. K is a compile-time parameter. In this way, the kernel is flexible to support an arbitrary number of pipes to make full use of FPGA resources. The $2K$ 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 among 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.

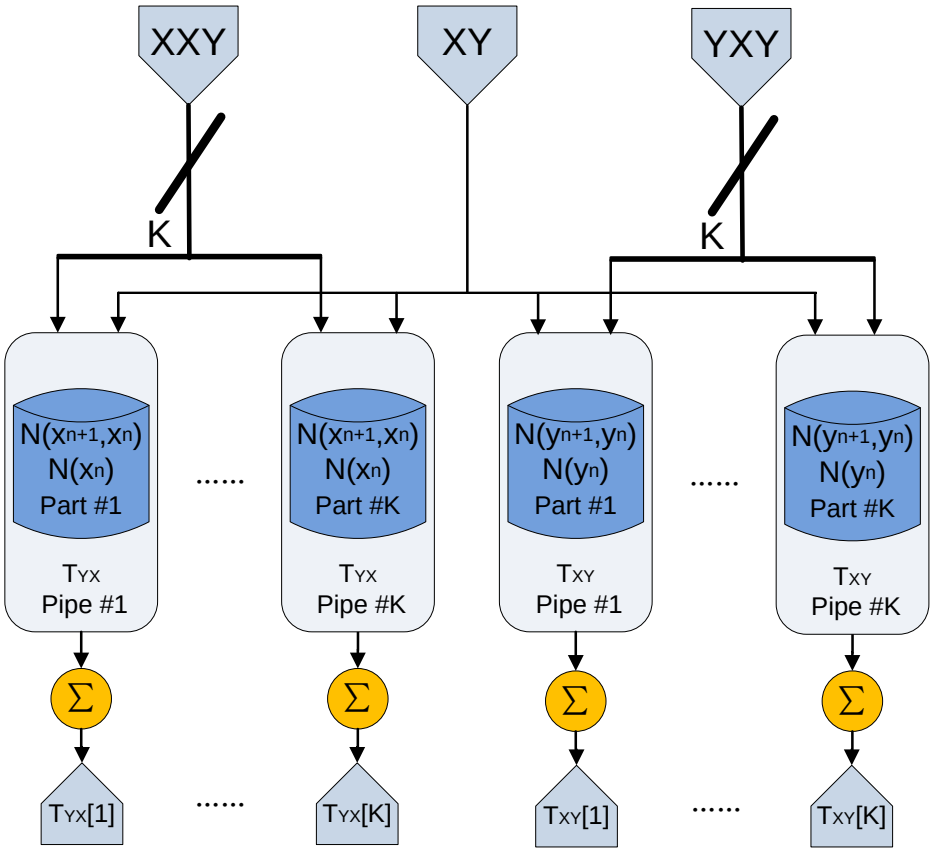

Figure 4.2: Kernel Architecture. This figure shows the datapath of the kernel with control logic omitted. Here XXY, YXY and XY stands for $N(x_{n+1}, x_n, y_n)$, $N(y_{n+1}, x_n, y_n)$ and $N(x_n, y_n)$, respectively. On each cycle, K elements from $N(x_{n+1}, x_n, y_n)$ and $N(y_{n+1}, x_n, y_n)$ are 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.

4.2 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 runtime 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.

In this case, a technique known as bit-width narrowing is used to make full use of the BRAM resource in FPGA. As resolution becomes larger, the number of occurrence tables have more elements but each element in the table is smaller. This is easy to understand since the total number of occurrence will be the number of samples, which is fixed. With is in mind, we may use fewer bits for each table element when the resolution becomes larger.

In our bit-width narrowing scheme, we set the bit-width to the minimum number that could represent the largest element in the table. By using custom unsigned integers ranging from 2-bit to 20-bit instead of the standard 32-bit `int`, we only use 6% - 62% of the original memory space, which enables storing the data tables in 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 that there are only $2R$ elements in total in $N(x_n)$ and $N(y_n)$, so they only take several KB of BRAM when R is around 1000.

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 runtime, 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 always sending 32-bit `int` to FPGA, we use unsigned integers with bit widths ranging from 1-bit to 32-bit, depending on the largest element in the table. As a result, up to 97% bandwidth resources are saved.

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.

4.3 Mixed-Precision Optimisation

In the C program for computing transfer entropy, $log_2()$ and the accumulator are 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, even in single precision.

As the dynamic range of input data is small, we use fixed-point number representation for the accumulator. 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 default setting for the accumulator in our system is 64-bit fixed-point number with 28 integer bits and 36 fractional bits.

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 $log_2()$ logic. 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 number with 8 exponent bits and 32 mantissa bits. We explore the relationship between the number of mantissa bits and accuracy as well as parallelism in Section 5.3.

4.4 Customising the Kernel

The power of dataflow computing lies in customisation. As mentioned in previous sections, we optimise memory allocation and apply bit-width narrowing as well as mixed-precision optimisation to boost performance. All of these techniques employed are customisations.

Here we would make clear how the proposed architecture could be customised for different applications.

- **Step 1: Get Representative Sample Data**
  For any customisations, the prerequisite is getting hold of representative time series data. Here, 'representative' means: (1) these time series should come from the same source as the actual time series to be processed; (2) the length of these time series should be similar to that of the actual time series. For example, if the system is built to process daily GBP-USD Forex rates, then the historical daily GBP-USD rates are considered to be representative.

Table 4.2: Range of the Number of Occurrence Tables

$\min\{N(x_{n+1}, x_n, y_n)\}$	$\max\{N(x_{n+1}, x_n, y_n)\}$
$\min\{N(y_{n+1}, x_n, y_n)\}$	$\max\{N(y_{n+1}, x_n, y_n)\}$
$\min\{N(x_{n+1}, x_n)\}$	$\max\{N(x_{n+1}, x_n)\}$
$\min\{N(y_{n+1}, y_n)\}$	$\max\{N(y_{n+1}, y_n)\}$
$\min\{N(x_n, y_n)\}$	$\max\{N(x_n, y_n)\}$
$\min\{N(x_n)\}$	$\max\{N(x_n)\}$
$\min\{N(y_n)\}$	$\max\{N(y_n)\}$

- **Step 2: Determine Resolution**
  The second step is to determine the resolution (R) for our kernel. In this stage we need to know how large are the numbers in the table. Once we have the representative sample time series, we could run the prepossessing routine to figure out the range of the number of occurrence tables, as shown in Table 4.2. It is worth pointing out that these ranges not only depend on the time series, but also the resolution used - larger resolution will lead to a larger table with smaller elements. The constraint is the BRAM space available on FPGA, because table $N(x_{n+1}, x_n)$ and $N(y_{n+1}, y_n)$ are distributed in the computing pipes and stored in BRAM. One should try different resolution values to figure out the feasible ones:

$$\sum_{i=1}^{K} BRAM\_XX_i + \sum_{i=1}^{K} BRAM\_YY_i < BRAM \tag{4.1}$$

$$SIZE_i = \frac{R^2}{K} \times BitWidth_i \tag{4.2}$$

  Here, (4.1) is the BRAM resource constraint, where $BRAM\_XX_i$ is the size of the ith block of table $N(x_{n+1}, x_n)$ and $BRAM\_YY_i$ the size of the ith block of table $N(y_{n+1}, y_n)$. (4.2) determines the size of each block. As the tables are distributed among K pipes, each block has $\frac{R^2}{K}$ elements. $BitWidth_i$ is the bit-width of block i, which is determined by the largest number in the block.

  Note that not all BRAM space is available for the two tables, as there will be some BRAM used for computing. In our experiments, one Xilinx Virtex-6 SX475T FPGA has about 4.67MB BRAM, but the space available for the data tables is around 3.5MB. Actually table $N(x_n)$ and $N(y_n)$ are also stored in BRAM, but their sizes (several KB) are negligible, so (4.1) does not take them into account.

- **Step 3: Determine Bit-width for each Resolution**
  Given one resolution, we know the range of the tables' elements in this resolution. Therefore, we could determine the minimal bit-width for the number of occurrence tables in this resolution. We use `unsigned integer` for all tables.

  This step is relatively straightforward. There are two considerations: (1) range of the elements in the data tables varies with resolution, so each resolution will lead to a specific bit-width setting; (2) it would be helpful to reserve a reasonable amount of additional space when determining the bit-width to allow for the differences between actual time series and the sample data.

- **Step 4: Determine the Precision for log_2 and the Accumulator**

  As mentioned in Section 4.3, we use fixed-point numbers for the accumulator. To determine the number of integer bits, we will need to run the C program to calculate the transfer entropy of the sample time series.

  To save logic resource, we move the final normalisation step to CPU. In the kernel the output of $log_2()$ is not multiplied by $\hat{p}(x_{n+1}, x_n, y_n)$ or $\hat{p}(y_{n+1}, x_n, y_n)$, but by $N(x_{n+1}, x_n, y_n) + 1$ or $N(y_{n+1}, x_n, y_n) + 1$. In this way the division logic could be saved, and we need to apply the final normalising on CPU, i.e., to divide the unnormalised transfer entropy result by $T - 1 + R^3$. Therefore, the accumulator in FPGA needs to hold the numbers as large as $T_{X\to Y} \times (T - 1 + R^3)$ or $T_{Y\to X} \times (T - 1 + R^3)$. By running the C program, we could derive the transfer entropy of the time series sample, and then we could tell how large the accumulated sum could be and how many integer bits are needed:

$$BIT_{YX} \geq \left\lceil log_2 \left[T_{Y\to X} \times (T - 1 + R^3) \right] \right\rceil \tag{4.3}$$

$$BIT_{XY} \geq \left\lceil log_2 \left[T_{X\to Y} \times (T - 1 + R^3) \right] \right\rceil \tag{4.4}$$

  It would be useful to reserve a reasonable amount of bit space to allow for fluctuations in the real data.

  For the number of fractional bits, it depends purely on the accuracy we want. In our default setting there are 36 fractional bits, so the precision is equal to $log_{10}(2^{36}) \approx 10.83$ decimal digits.

  For the data representation of $log_2()$, the precision should be comparable to that of the accumulator. Our default setting is 40-bit floating

point number with 8 exponent bits and 32 mantissa bits. This precision is about $log_{10}(2^{32}) \approx 9.63$ decimal digits, slightly smaller than that of the accumulator. It will be shown in Section 5.3 that this precision is accurate enough. However, it is possible to further increase the number of the mantissa bits to 36 or even larger for better accuracy.

In summary, we use 4 steps to customise our kernel for an application: (1) get representative data; (2) determine resolution for the kernel; (3) determine the bit-width for the number of occurrence tables; (4) determine the precision for $log_2()$ and the accumulator. In Chapter 5, we will tailor our kernel for different applications in the case studies.

4.5 Performance Model

In this section, we provide a performance model for dataflow system. In software, the `for` loop for computing transfer entropy has R^3 iterations. In our hardware kernel, as there are K pipes running concurrently, the kernel only needs to run for R^3/K cycles. In other words, the loop is strip-mined by K.

The kernel computing time is given by:

$$T_{Comp} = \frac{1}{Freq} \times \frac{R^3}{K} \tag{4.5}$$

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

$$T_{I/O} = \frac{DATA\_SIZE}{BW} \tag{4.6}$$

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

$$T_{Total} = max\{T_{Comp}, T_{I/O}\} \tag{4.7}$$

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

4.6 Summary

In this chapter, we provided details on our hardware architecture.

The challenges addressed are: (1) limited CPU-FPGA I/O bandwidth; (2) limited logic resources on FPGA. To address the first challenge, we optimised memory allocation and applied bit-width narrowing. The number of occurrence tables which are: (1) small-sized; (2) mid-sized but frequently accessed, are mapped to FPGA BRAM. In this way, a considerable I/O amount is eliminated. Bit-width narrowing is used to reduce the size of BRAM contents so as to allow for larger resolution. Also, it helps to reduce I/O bandwith requirement of the tables to be streamed to FPGA at runtime. To deal with the second challenge, we used mixed-precision optimisation. The accumulator is in fixed-point numbers while $log_2()$ is implemented in floating-point numbers.

We then discussed how to customise the proposed kernel architecture for different applications. First, we need to get representative time series sample. Then we run preprocessing to find the ranges of the number of occurrence tables, i.e. minimum or maximum numbers. The maximum number in the tables varies with resolution. We could try different resolutions to find the largest possible one, which is limited by BRAM resource. Once we determined the resolution, we could figure out the optimal bit-width for the data tables in this resolution and a worst case upper bound for $log_2()$'s input. Finally, we run the C program to calculate the transfer entropy of the sample data to determine the format for the accumulator and $log_2()$.

We also illustrated the general kernel architecture. In the kernel there are K pipes for transfer entropy $T_{Y \to X}$ and $T_{X \to Y}$, respectively. K is a compile-time parameter to allow for parametric parallelism. The kernel also supports ultra-large resolution by distributing computing pipes among multiple FPGAs. Last but not least, a performance model for the kernel is presented.

Chapter 5

Experimental Evaluation

In this chapter, we present the experimental evaluation for the proposed FPGA system for transfer entropy.

We will first introduce our test platform in Section 5.1. Section 5.2 explores the relation between accuracy and parallelism. Then Section 5.3 and 5.4 presents the performance tests in different scenarios. Finally, Section 5.5 discusses the performance bottleneck of the system.

5.1 Platform Specification

We use Maxeler MPC-C platform in our experiments. The dataflow design is described in Maxeler's MaxJ language and compiled to VHDL using Maxeler MaxCompiler. The VHDL description is then synthesised and mapped to FPGA with the Xilinx tool chain. The proposed dataflow design is built on a Maxeler MAX3 FPGA card with one Xilinx Virtex-6 SX475T FPGA running at 80-120MHz. The FPGA card is integrated with the host computer via PCI-E Gen2 x8 interface. The host computer has Intel Xeon X5650 CPU (6-cores running at 2.66GHz) and 48GB DDR3 1600MHz memory.

To evaluate the performance and accuracy of the hardware solution, we build a reference C program in double precision for transfer entropy. This reference program runs 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 achieved using OpenMP library. The FPGA host code and the reference C program are compiled using Intel C Compiler with the highest compiler optimisation.

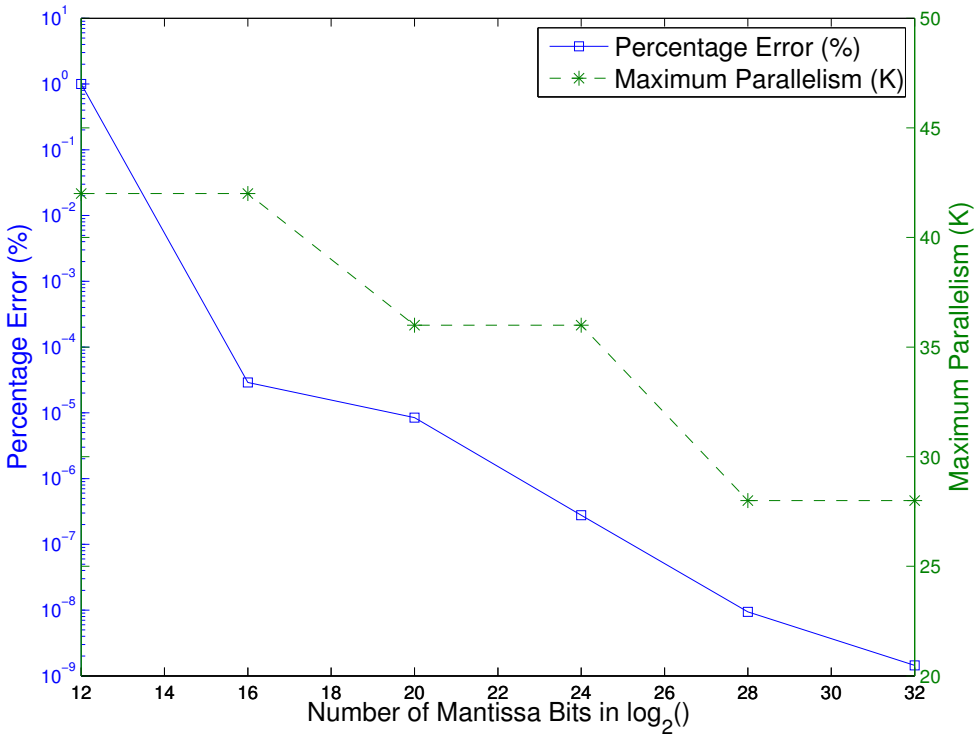

Figure 5.1: Percentage Error (%) and Maximum Parallelism (K) vs. Number of Mantissa Bits in $log_2()$. Test time series are 10^9 random numbers. Resolution is fixed at 1000. Percentage error is measured against CPU result. Parallelism is measured by the number of computing pipes (K) for $T_{X \to Y}$ 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 $T_{Y \to X}$.

5.2 Accuracy versus Parallelism

Figure 5.1 shows the percentage error of transfer entropy and maximum parallelism can be achieved on one Virtex-6 SX475T FPGA as a function of number of the 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 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 default precision setting for $log_2()$ has 32 mantissa bits (40-bit floating point number with 8 exponent bits and 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 resources, creating a very demanding task for placing and routing tools. In some cases, kernel frequency has to be reduced in order to cut down flip-flop usage so that mapping, placing and routing could be successfully done. In the following performance tests, we set $K = 16$ or $K = 24$ although the maximum possible value is 28.

5.3 Case Study - Random Numbers

In this case study, we use 10^9 random numbers as test time series. Section 5.3.1 details kernel customisation; Section 5.3.2 reports hardware resource usage; Section 5.3.3 presents performance evaluation.

5.3.1 Kernel Customisation

Step 1: Get Representative Sample Data

The test time series used in this case study are 10^9 random numbers generated by `rand()` function in the C program.

Step 2: Determine Resolution

We will first figure out the feasible resolution values for the time series. Recall equation (4.1) and (4.2):

$$\sum_{i=1}^{K} BRAM\_XX_i + \sum_{i=1}^{K} BRAM\_YY_i < BRAM$$

$$SIZE_i = \frac{R^2}{K} \times BitWidth_i$$

It is clear that the largest resolution supported is limited by BRAM space in FPGA. In our test platform, there is about 3.5MB BRAM available for the tables. To find out the largest resolution, we run the C program using different resolution values. Table 5.1 shows the largest element in table $N(x_{n+1}, x_n)$ and $N(y_{n+1}, y_n)$ in different resolutions, as well as the BRAM space required. Since the time series is random, it is found that all patterns occur equivalently likely. As a result, the largest numbers in all blocks are quite close to each other. Therefore, we could use the same bit-width for all BRAM blocks. As shown in the table, given the BRAM resource constraint, the largest resolution supported is 1200. Consequently, we set the range of resolution used in this case study to be 200 - 1200.

Step 3: Determine Bit-width for each Resolution

The bit-width of a number of occurrence table is the minimum number of digits needed to represent the largest number in that table. In step 2, we have already figured out the largest number in the tables for each resolution, so this step is rather straightforward. Table 5.1 and 5.2 shows the optimised bit-widths for table $N(x_{n+1}, x_n)$, $N(y_{n+1}, y_n)$, $N(x_{n+1}, x_n, y_n)$ and $N(y_{n+1}, x_n, y_n)$.

Step 4: Determine the Precision for log_2 and the Accumulator

The transfer entropy result given by the reference C program shows that the accumulated sum in the computing pipes could reach `1e8`. Therefore, we use the default precision setting for the accumulator (64-bit fixed-point with 28 integer bits and 36 fractional bits), as 28 integer bits can represent a number up to 2.68×10^8. Also, we use the default setting for $log_2()$ (40-bit floating point with 8 exponent bits and 32 mantissa bits).

5.3.2 FPGA Resource Usage

The FPGA has 48 computing pipes ($K = 24$), 24 for $T_{X \to Y}$ and 24 for $T_{Y \to X}$. The kernel frequency is 80MHz. Table 5.3 shows the hardware resource usage of Xilinx Virtex-6 SX475T FPGA when $R = 1200$. Detailed resource usage of other resolutions are attached in Appendix A.

We deploy 48 transfer entropy computing pipes ($K = 24$). The LUT and FF usages are generally determined by the number of computing pipes (K). BRAM usage depends on resolution, because most of the BRAM is devoted to the number of occurrence 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.

If 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 4.2, 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.

Table 5.1: Bit-width Narrowing for $N(x_{n+1}, x_n)$ and $N(y_{n+1}, y_n)$, using 10^9 Random Numbers as Test Time Series

Resolution	200	300	400	500	600	700
Largest Element	26182	11959	7049	4331	3037	2286
Format	uint16	uint16	uint16	uint16	uint12	uint12
Total Size (MB)	0.15	0.34	0.61	0.95	1.03	1.40

Resolution	800	900	1000	1100	1200
Largest Element	1768	1415	1161	971	837
Format	uint12	uint12	uint12	uint10	uint10
Total Size (MB)	1.83	2.32	2.86	2.88	3.43

Table 5.2: Bit-width Narrowing for $N(x_{n+1}, x_n, y_n)$ and $N(y_{n+1}, x_n, y_n)$, using 10^9 Random Numbers as Test Time Series

Resolution	200	300	400	500	600	700
Largest Element	194	80	46	28	22	17
Format	uint8	uint8	uint6	uint5	uint5	uint5
Total Size (MB)	15.26	51.50	91.55	149.01	257.49	408.89

Resolution	800	900	1000	1100	1200
Largest Element	15	14	11	11	10
Format	uint4	uint4	uint4	uint4	uint4
Total Size (MB)	488.28	695.22	953.67	1269	1648

Table 5.3: FPGA Resource Usage (Resolution = 1200), using 10^9 Random Numbers as Test Time Series

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

5.3.3 Performance Test

For performance comparison, we measure the execution time of transfer entropy computation, which corresponds to the computing time of (3.26) and (3.27). The performance of single Xeon CPU core, 6 Xeon CPU cores and one Virtex-6 FPGA is shown in Figure 5.2.

FPGA demonstrates high performance for transfer entropy computation. The maximum speed-up is achieved when $R = 1000$. In this case, the proposed FPGA implementation is 111.47 times and 18.69 times faster than a single CPU core and a 6-core CPU, respectively. This high performance is achieved by the massive amount of parallelism in hardware. In CPU, 6 cores are used for computing, so there are actually 3 pipes for $T_{X\to Y}$ and $T_{Y\to X}$, respectively. 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 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.

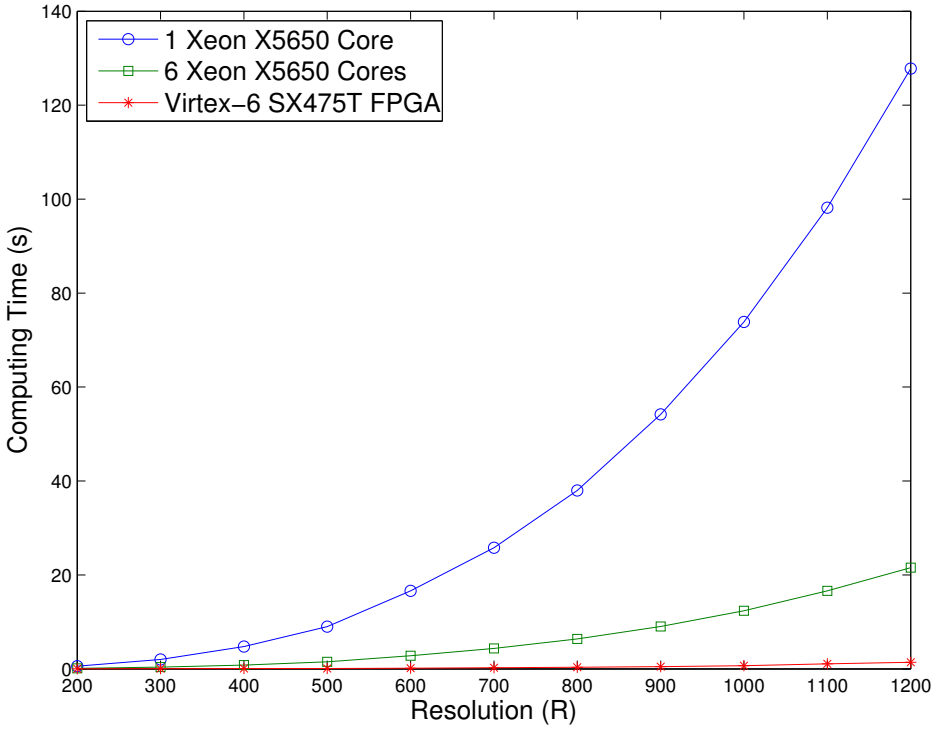

Figure 5.2: Performance vs. Resolution using random numbers. Test time series are 10^9 random numbers. The Virtex-6 FPGA has 48 computing pipes ($K = 24$) running at 80MHz. $log_2()$ is implemented in 40-bit floating point with 8 exponent bits and 32 mantissa bits. Accumulator is set to 64-bit fixed point with 28 integer bits and 36 fractional bits.

5.4 Case Study - Forex Data

In the second case study, we use historical Forex data as test inputs. The time series are EUR-USD and JPY-USD tick data from 23 Mar 2014 to 28 Mar 2014. The tick data is synchronised and differenced, resulting in 797469 records in each of the time series.

5.4.1 Kernel Customisation

Step 1: Get Representative Sample Data

The test data used in this case study come from the real world. The time series are historical EUR-USD and JPY-USD Forex rates from Chicago Mercantile Exchange (CME). The time series are the Forex rate tick data from 23 Mar 2014 to 28 Mar 2014. After synchronisation and differencing, there are 797469 records in each time series.

We must understand the characteristics of the time series when customising the kernel. There is fundamental difference between real finance data and random numbers. For random time series, x_n and x_{n+1} are independent, so x_{n+1} can be quite different from x_n. In the Forex tick data, Forex rate is sampled in a microsecond scale. The key difference is that Forex rate can hardly change in $1\mu s$, so x_{n+1} will be quite close to x_n. After differencing, most of the elements in the time series will become 0 or ± 1.

This interesting property will lead to ill-balanced number of occurrence tables. Take $N(x_{n+1}, x_n)$ as an example. As most of the elements in the time series will become 0 or ± 1, the occurrence count of $(0, 0)$, $(0, \pm 1)$, $(\pm 1, 0)$ will be very high. In contrast, the patterns such as $(0, 10)$ rarely occurs, resulting in a low count. As a result, only a small part of the table has large numbers, which is very different from the tables in the previous case study, where the large numbers tend to distribute evenly. To achieve high performance in this case, we must deal with the sparse tables efficiently.

Step 2: Determine Resolution

In spite of the fact that most elements in the number of occurrence tables are rather small (less than 10), the largest number of the table can reach 10^5. Therefore, using a fixed bit-width for all elements in a sparse table, i.e. the bit-width to represent 10^5, will waste a lot of memory resource. It would be desirable if we could use narrow bit-width for the small numbers, and wide bit-width for the large ones.

Our kernel architecture has the ability to achieve this. As mentioned in Section 4.1, the tables $N(x_{n+1}, x_n)$ and $N(y_{n+1}, y_n)$ are separated into K blocks and distributed among the K computing pipes. Therefore, instead of using the same bit-width for all blocks, we could customise the bit-width for each block. Table 5.4 and 5.5 show the optimised bit-width settings for the BRAM blocks storing table $N(x_{n+1}, x_n)$ and $N(y_{n+1}, y_n)$ when resolution is 1200. Here, XX_i and YY_i stand for the ith block of $N(x_{n+1}, x_n)$ and $N(y_{n+1}, y_n)$, respectively. As can be seen from Table 5.4 and 5.5, it is clear that large bit-width is only needed in one or two blocks of the table, while other blocks could use narrow bit-width settings.

With detailed customisation, we only use 1.65MB in total for $N(x_{n+1}, x_n)$ and $N(y_{n+1}, y_n)$, which could fit in BRAM space in FPGA (3.5MB). In contrast, without customisation for each block in table, we will need to use `uint20` for the two tables, resulting in 6.87MB BRAM space requirement. Therefore, this customisation scheme is essential for the kernel to support resolution 1200 or even larger in one FPGA. For the ease of comparison, we also set the range of resolution to be 200-1200 in this case study.

Table 5.4: Bit-Width Narrowing for $N(x_{n+1}, x_n)$, using Forex Data, Resolution R = 1200

Block Name	XX_1	XX_2	XX_3	XX_4	XX_5	XX_6
Largest Element	23	1	1	738618	1	1
Format	uint6	uint2	uint2	uint20	uint2	uint2
Block Name	XX_7	XX_8	XX_9	XX_10	XX_11	XX_12
Largest Element	39	16	5	5	13306	2
Format	uint6	uint6	uint4	uint4	uint14	uint2
Block Name	XX_13	XX_14	XX_15	XX_16	XX_17	XX_18
Largest Element	2	221	6	1	6	172
Format	uint2	uint8	uint4	uint2	uint4	uint8
Block Name	XX_19	XX_20	XX_21	XX_22	XX_23	XX_24
Largest Element	2	1	13289	5	3	11
Format	uint2	uint2	uint14	uint4	uint2	uint4

Table 5.5: Bit-Width Narrowing for $N(y_{n+1}, y_n)$, using Forex Data, Resolution R = 1200

Block Name	YY_1	YY_2	YY_3	YY_4	YY_5	YY_6
Largest Element	1	22	0	0	52	5
Format	uint2	uint6	uint2	uint2	uint6	uint4
Block Name	YY_7	YY_8	YY_9	YY_10	YY_11	YY_12
Largest Element	0	2	766148	0	0	53
Format	uint2	uint2	uint20	uint2	uint2	uint6
Block Name	YY_13	YY_14	YY_15	YY_16	YY_17	YY_18
Largest Element	0	0	12	11	0	0
Format	uint2	uint2	uint4	uint4	uint2	uint2
Block Name	YY_19	YY_20	YY_21	YY_22	YY_23	YY_24
Largest Element	7281	0	0	7319	0	1
Format	uint14	uint2	uint2	uint14	uint2	uint2

Table 5.6: FPGA Resource Usage (Resolution = 1200), using Forex Data as Test Time Series, $K = 24$

	LUT	Primary FF	Secondary FF	DSP	BRAM18
Available	297600	297600	297600	2016	2128
Used	243657	238565	62784	822	1289
Usage (%)	81.87%	80.16%	21.10%	40.77%	60.57%

Step 3: Determine Bit-width for each Resolution

We have already figured out the bit-width settings of table $N(x_{n+1}, x_n)$ and $N(y_{n+1}, y_n)$ in the previous step. In this step, we apply bit-width narrowing to the two big tables $N(x_{n+1}, x_n, y_n)$ and $N(y_{n+1}, x_n, y_n)$. It is found that in the two big tables, most elements are also quite small. To make use of this property, we use different bit-width settings for the two tables in different iterations. For example, when resolution is 1200, there are 1200 iterations in the outer loop. We use `uint1` for iteration [1, 600] and [861, 1200], `uint8` for iteration [601, 710] and [741, 860]; and `uint32` for iteration [711, 740].

In this way, the standard 32-bit unsigned integer `uint32` is only used in 30 iterations out of 1200. Compared with the original scheme of using `uint32` at all times, our customisation saved 91% bandwidth, which is the key to deal with CPU-FPGA bandwidth bottleneck.

Step 4: Determine the Precision for log_2 and the Accumulator

For $log_2()$ computing in hardware, we still use our default precision setting (40-bit floating point with 8 exponent bits and 32 mantissa bits). For the accumulator, as the accumulated sum can reach `1e9`, we use 64-bit fixed point data format with 32 integer bits and 32 fractional bits, which could represent numbers up to 4.29×10^9.

5.4.2 FPGA Resource Usage

For resolution 1200 and 1104, we build 48 pipes in FPGA, 24 for $T_{X \to Y}$ and 24 for $T_{Y \to X}$ ($K = 24$). The 48 pipes run at 100MHz. For resolution 208 - 1008, we build 32 pipes, 16 for $T_{X \to Y}$ and 16 for $T_{Y \to X}$ ($K = 16$). The 32 pipes run at 120MHz. This setting is for the ease of compilation.

Table 5.6 shows FPGA resource usage when resolution = 1200 and $K = 24$. It can be seen that with our bit-width narrowing strategy, the BRAM usage is effectively controlled. The resource usage in other resolution is attached in the Appendix B.

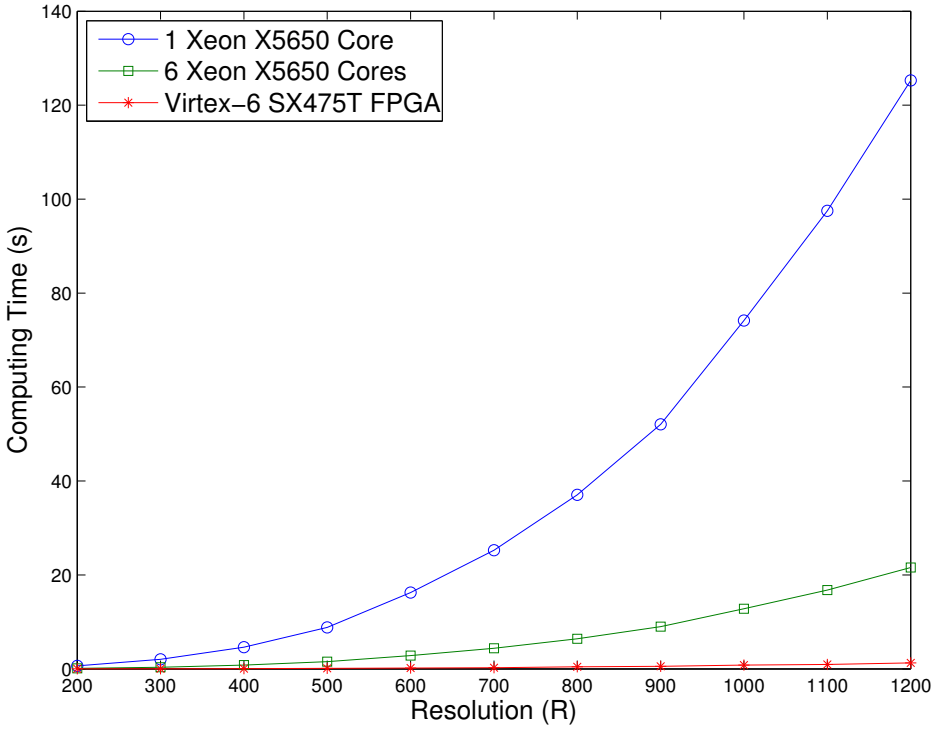

Figure 5.3: Performance vs. Resolution using historical Forex data. The Virtex-6 FPGA has 48 computing pipes ($K = 24$) running at 100MHz ($R = 1200, 1104$) or 32 computing pipes ($K = 16$) running at 120MHz ($R \leq 1008$). $log_2()$ is implemented in 40-bit floating point with 8 exponent bits and 32 mantissa bits. Accumulator is set to 64-bit fixed point with 32 integer bits and 32 fractional bits.

5.4.3 Performance Test

We run the performance test to compare the performance of FPGA, one CPU core and 6 CPU cores. The performance results are shown in Figure 5.3. The highest speed-up is achieved when resolution is 1104: the FPGA is 103.93 times faster than one Xeon CPU core, and 17.89 times faster than a 6-core Xeon CPU.

As seen from Figure 5.2 and Figure 5.3, FPGA demonstrates consistent speed-up for both random numbers and historical Forex data. We can see the proposed FPGA system is able to deliver high performance for real applications. Our case study shows FPGA has great potential for transfer entropy analysis targeting financial data such as Forex rates, which makes FPGA an ideal candidate for quantitative finance.

5.5 Bottleneck

Although 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 I/O bandwidth.

In our tests using random numbers, there are 48 computing pipes ($K = 24$) in the system running at 80MHz. When resolution = 1200, the computing time is 0.9s, according to (4.5). However, the actual measured time in the experiments is 1.41s. Therefore, the kernel is clearly bounded by I/O speed.

When $R = 1200$, 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 (4.8), we can estimate the actual bandwidth to be about 1.14GB/s. This is the same with other experiments using different resolutions, where the actual bandwidth is about 1.1-1.3GB/s.

Since there are 48 computing pipes, each cycle the FPGA kernel needs 24 bytes 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 CPU via PCI-E Gen2 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 Gen2 x8 is about 3GB/s. Furthermore, 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 FPGA board could fully support PCI-E Gen2 x8 interface, we will have about 3GB/s bandwidth available, so our 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 5.3, there are still plenty of logic resources available, so it is possible to build a decompressor on FPGA.

5.6 Summary

This chapter features the experimental evaluation for the proposed dataflow design accelerating transfer entropy computation.

Our test platform is a Maxeler MPC-C system with Intel Xeon 6-core CPU and Xilinx Virtex-6 FPGAs. We use random numbers and historical forex data as test inputs. The proposed system supports resolution up to 1200 in one Xilinx-6 SX475T FPGA. Our hardware solution achieves double precision accuracy as a moderate logic cost. The dataflow system further benefits from great parallelism achievable in FPGA, as we build 16-24 pipes for $T_{X \to Y}$ and $T_{Y \to X}$, respectively. The proposed hardware solution is up to 111.47 times faster than a single CPU core and 18.69 times faster than a 6-core Xeon CPU. The FPGA solution has the potential to be even faster if an interface chip fully compatible with PCI-E Gen2 x8 standard is available.

Chapter 6

Conclusion and Future Work

The aim of this chapter is to give a summary of our achievements in accelerating transfer entropy computation and suggest probabilities for future work. In section 6.1 we summarise the key contributions in this project. In section 6.2, we discuss a list of potential future work.

6.1 Summary of Achievements

This project features the first reconfigurable computing solution to transfer entropy computation.

Our first objective is to deal with the common situations in which only limited samples of time series data are available. Limited data affect the accuracy of transfer entropy by affecting the estimated (joint) probabilities used during computation. Hence we introduce a novel probability estimation technique based on Laplace's rule of succession. This method is used to estimate the (joint) probabilities for computing transfer entropy. Compared with the traditional frequentist statistics approach, the proposed method successfully eliminates the zero probability problem when a certain pattern does not occur in the samples.

Apart from the novel theoretical contribution, we explore hardware acceleration for computing transfer entropy. We identify the challenge to be limited FPGA-CPU I/O bandwidth and limited FPGA logic resources. We deploy an optimised memory allocation scheme to map small and frequently accessed data tables to FPGA BRAM, so I/O amount is effectively reduced. In addition, bit-width narrowing is used to further reduce bandwidth requirement and save BRAM resources. To cut down logic usage, we use mixed-precision optimisation to find the best trade-off between accuracy and hardware resource utilisation, achieving double precision only at a moderate logic cost.

To evaluate the proposed solution, we implement our dataflow design on a Maxeler MPC-C system. The FPGA kernel is run on a Xilinx Virtex-6 SX475T FPGA. We do extensive experimental evaluation using random numbers and historical Forex data. The proposed system achieves up to 111.47 times speed-up over a single Xeon CPU core and 18.69 times speed-up over a 6-core Xeon CPU, showing exciting performance.

A conference paper, named "*Accelerating Transfer Entropy Computation*", by Shengjia Shao, Ce Guo, Wayne Luk and Stephen Weston has been submitted to *the 2014 International Conference on Field-Programmable Technology* (ICFPT 2014). This paper reflects the contributions of this project. The paper is currently under review.

6.2 Future Work

The work shows the potential of dataflow computing for calculating transfer entropy. Although we have met the original goals set out in Section 1.2, we would like to highlight future work that could be done, which further improves the quality of this project.

- **Advanced Data Compression**

  As discussed in Section 5.6, the current system is bounded by CPU-FPGA I/O bandwidth. Beyond the bit-width narrowing technique used in this project, there are possibilities to deploy advanced data compression techniques to further compress the data tables which are streamed to FPGA at run-time. A compressor could be built in host C program while decompressor could be build on FPGA using resources remaining. This will help to further reduce I/O overhead and improve performance.

- **Automatic Customisation**

  In the experimental evaluation section, we test the proposed solution using random numbers and historical forex data. There exist many more fields where transfer entropy is used. It would be beneficial to find customised versions of the dataflow design for different applications, as well as ways to automate these customisations.

- **Run-time Reconfiguration**

  The current FPGA system is static. As run-time reconfiguration is a powerful technique which is able to further boost the performance of a FPGA design, an exciting possibility is to explore whether there are parameters in the dataflow design that can be adaptively optimised at run-time.

Appendix A

FPGA Resource Usage - Random Numbers

This table is the detailed FPGA resource usage in the case study in Section 5.3. We use 10^9 random numbers as test time series. There are 48 computing pipes on a Xilinx Virtex-6 SX475T FPGA, running at 80MHz. Here, 'Available' is the resource available in one Xilinx Virtex-6 SX475T FPGA, 'R' is resolution.

Table A.1: FPGA Resource Usage for Resolution from 192 to 1200

	LUT	Primary FF	Secondary FF	DSP	BRAM18
Available	297600	297600	297600	2016	2128
R=192	151591	170676	29400	680	414
R=288	152341	171746	28710	686	534
R=384	123026	135036	26247	544	596
R=480	121436	137488	23557	550	756
R=600	122383	135310	24750	544	820
R=696	122731	134891	25165	544	980
R=792	151976	170456	28108	680	1210
R=912	154454	168447	30363	686	1450
R=1008	155323	168630	30264	686	1770
R=1104	169493	183949	33310	846	1692
R=1200	201697	215724	42555	1014	2011

Appendix B

FPGA Resource Usage - Forex Data

This table is the detailed FPGA resource usage in the case study in Section 5.4. We use historical Forex data as test time series. For resolution 1200 and 1104, we build 48 pipes in FPGA, 24 for $T_{X \to Y}$ and 24 for $T_{Y \to X}$ ($K = 24$). The 48 pipes run at 100MHz. For resolution 208 - 1008, we build 32 pipes, 16 for $T_{X \to Y}$ and 16 for $T_{Y \to X}$ ($K = 16$). The 32 pipes run at 120MHz. Here, 'Available' is the resource available in one Xilinx Virtex-6 SX475T FPGA, 'R' is resolution.

Table B.1: FPGA Resource Usage for Resolution from 208 to 1200

	LUT	Primary FF	Secondary FF	DSP	BRAM18
Available	297600	297600	297600	2016	2128
R=208	161685	170011	32776	544	351
R=304	161115	172668	30132	544	384
R=400	162055	170556	32348	544	435
R=496	161368	173216	29662	550	501
R=608	160735	175451	27473	544	574
R=704	161358	173509	29307	544	665
R=800	161885	175082	27919	544	772
R=896	161208	176229	26762	544	853
R=1008	162128	172889	30055	550	1125
R=1104	244890	242950	58915	816	1213
R=1200	243657	238565	62784	822	1289

Appendix C

Performance Data - Random Numbers

This table is the detailed performance results in the case study in Section 5.3. Test time series are 10^9 random numbers. This table corresponds to Figure 5.2. Here, 'R' is resolution; 'CPU_1' is the CPU computing time using one core; 'CPU_6' is the CPU computing time using 6 cores; 'FPGA' is the FPGA computing time; 'SpeedUp_1' is the FPGA speed-up factor against one CPU core; 'SpeedUp_6' is the FPGA speed-up factor against 6 CPU cores.

Table C.1: Performance Data - Random Numbers

R	CPU_1 (μs)	CPU_6 (μs)	FPGA (μs)	SpeedUp_1	SpeedUp_6
192	590082	101320	15126	39.01x	6.70x
288	1991579	372840	50503	39.43x	7.38x
384	4740705	823596	68553	69.15x	12.01x
480	9007084	1521974	83049	108.46x	18.32x
600	16628953	2801819	153077	108.63x	18.30x
696	25805482	4350634	247297	104.35x	17.59x
792	37993861	6383770	341359	111.30x	18.70x
912	54167182	9024529	483412	112.05x	18.67x
1008	73870908	12385822	662712	111.47x	18.69x
1104	98190384	16619120	1092233	89.89x	15.21x
1200	127799347	21555613	1414017	90.38x	15.24x

Appendix D

Performance Data - Forex Data

This table is the detailed performance results in the case study in Section 5.3. Test time series are historical Forex rates. This table corresponds to Figure 5.3. Here, 'R' is resolution; 'CPU_1' is the CPU computing time using one core; 'CPU_6' is the CPU computing time using 6 cores; 'FPGA' is the FPGA computing time; 'SpeedUp_1' is the FPGA speed-up factor against one CPU core; 'SpeedUp_6' is the FPGA speed-up factor against 6 CPU cores.

Table D.1: Performance Data - Forex Data

R	CPU_1 (μs)	CPU_6 (μs)	FPGA (μs)	SpeedUp_1	SpeedUp_6
208	651921	118006	9957	65.47x	11.85x
304	2034286	356006	23648	86.02x	15.05x
400	4632337	803670	55661	83.22x	14.43x
496	8832081	1529045	106025	83.30x	14.42x
608	16262400	2819432	172524	94.26x	16.34x
704	25248084	4380763	257196	98.16x	17.02x
800	37049445	6409638	435248	85.12x	14.72x
896	52056427	8996697	536010	97.12x	16.78x
1008	74166082	12814518	795641	93.21x	16.10x
1104	97517934	16793175	938244	103.93x	17.89x
1200	125284751	21568302	1264596	99.07x	17.05x

Bibliography

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

[2] I. Kuon, R. Tessier, and J. Rose, "FPGA architecture: Survey and challenges," *Found. Trends Electron. Des. Autom.*, vol. 2, pp. 135–253, Feb. 2008.

[3] Altera, "Introduction to Quartus II." `http://www.altera.co.uk/literature/manual/archives/intro_to_quartus2.pdf`.

[4] Maxeler Technologies, "Maxeler MPC-C." `http://www.maxeler.com/products/mpc-cseries/`.

[5] Maxeler Technologies, *Multiscale Dataflow Computing*. 2013.

[6] 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.

[7] 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. 10240–10245, 2007.

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

[9] 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, pp. 275–280, Dec. 2011.

[10] 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*, pp. 212–219, International Society for Optics and Photonics, 2004.

[11] 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*, pp. 1–6, June 2008.

[12] 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.

[13] Altera, "Altera Quartus." `http://www.altera.co.uk/products/software/quartus-ii/about/qts-performance-productivity.html`.

[14] Xilinx, "Xilinx ISE." `http://www.xilinx.com/products/design-tools/ise-design-suite/`.

[15] Xilinx, "Xilinx Vivado." `http://www.xilinx.com/products/design-tools/vivado/`.

[16] X. Niu, T. Chau, Q. Jin, W. Luk, and Q. Liu, "Automating elimination of idle functions by run-time reconfiguration," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pp. 97–104, April 2013.

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

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