Pipelined Reconfigurable Accelerator for Ordinal Pattern Encoding

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

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

Stephen Weston
Maxeler Technologies
London, United Kingdom
weston@maxeler.com

***Abstract*—Ordinal analysis is a statistical method for analysing the complexity of time series. This method has been used in characterising dynamic changes in time series, with various applications such as financial risk modelling and biomedical signal processing. Ordinal pattern encoding is a fundamental calculation in ordinal analysis. It is computationally demanding particularly for high query orders and large time series data. This paper presents the first reconfigurable accelerator for this encoding calculation, with four main contributions. First, we propose a two-level hardware-oriented ordinal pattern encoding scheme to avoid sequence sorting operations in the accelerator, enabling theoretically best code compactness. Second, we develop a hardware mapping method by promoting data reuse, by parallelising arithmetic operations, and by pipelining the data path. Third, we conduct an experimental implementation of the proposed system, showing promising accelerated performance compared to software solutions. Finally, we apply the proposed accelerator to the computation of permutation entropy, demonstrating the significant potential for acceleration that would benefit such computation.**

Keywords*—*ordinal analysis; customisable accelerator; permutation entropy

I. Introduction

The study of time series is a crucial topic in many fields. In particular, it is one of the essential foundations of biomedical signal processing and financial computing. One important task for time series analysis is to quantify the complexity of a time series. A complexity measure is expected to reflect the regularity and predictability of the time series. Ordinal analysis is an approach to analyse the complexity of time series by considering how subsequences in the time series are ordered. This approach does not depend on a particular time series model, and it is robust against various types of noise.

A practical problem of ordinal analysis is the lack of fast encoding systems for ordinal patterns. This problem was not significant at the time when the concept of ordinal analysis was introduced. However, in recent years, data analysts began to appreciate the value of microscopic statistical patterns in large time series data. Therefore, the efficiency problem arose. Moreover, the value of temporal data decays as time goes by. It is critical to extract useful information as fast as possible before the data become meaningless. As a result, it is desirable to encode ordinal patterns of a time series as fast as possible.

Reconfigurable computing is a promising acceleration technology for encoding ordinal patterns, but all acceleration solutions we know in existing work are based on CPU platforms. It is challenging to design a reconfigurable accelerator for ordinal pattern encoding because the calculations involve operations such as sequence sorting that do not have efficient hardware implementations.

We address this challenge and propose the first reconfigurable accelerator for ordinal pattern encoding. A highlight of our solution is that we do not follow the encoding schemes in existing software solutions. Instead, we derive a novel encoding scheme and computational routines from the perspective of hardware design. This solution is both fast and scalable, and hence particularly useful for encoding ordinal patterns for large time series data and high query orders. Our key contributions are as follows.

- A two-level ordinal pattern encoding scheme that avoids sequence sorting and reduces data transmission.
- A pipelined hardware architecture for the proposed encoding scheme.
- An experimental implementation on a commercial FPGA-based accelerator.
- A case study applying our encoding engine to the calculation of permutation entropy

The aim of our study is not just to accelerate the ordinal pattern encoding for a particular application. Instead, we hope to accelerate this form of calculation in general. Our proposed accelerator can be applied to most ordinal analysis problems with little or no modification.

The rest of the paper is organised as follows. Section II briefly describes the ordinal pattern encoding problem and reviews acceleration solutions for statistical data analysis. Section III presents our proposed pipeline-friendly ordinal pattern encoding algorithm and discusses its hardware-oriented features. Section IV describes a hardware design based on mapping our algorithm to a fully-pipelined architecture. Section V provides experimental results of an implementation of our proposed hardware design, and explains experimental observations. Section VI describes a case study applying our proposed architecture to calculate the permutation entropy for time series. Section VII concludes this paper and discusses possible future work.

II. Background

Ordinal analysis is a statistical method to investigate the complexity of time series. In this section, we first introduce ordinal analysis and ordinal pattern encoding. We then discuss existing acceleration solutions for ordinal analysis and other time-series processing methods.

A. *Ordinal Pattern Encoding*

Given a sequence of n distinct values $b = (b_1, \dots, b_n)$, the ordinal pattern of b is mathematically described by a permutation $\pi = (k_1, \dots, k_n)$, such that $b' = (b_{k_1}, \dots, b_{k_n})$ is in ascending order. For instance, the sequences $b = (b_1, b_2, b_3) = (5, 2, 4)$ satisfies $b_2 < b_3 < b_1$. Therefore, the ordinal pattern of this sequence is $(2, 3, 1)$.

Given a time series $x = (x_1, \dots, x_T)$ and an integer $n \in [1..T]$, we know that x contains $(T - n + 1)$ contiguous subsequences with length n. Ordinal patterns of these subsequences contain rich information about chaotic properties of the time series.

Ordinal analysis refers to the study of the distribution of these ordinal patterns. One may use such distributions to detect dynamic changes, to estimate the predictability, and to gather information for regression or classification. In order to estimate the ordinal pattern distribution, the first step is to extract all ordinal patterns in the time series and encode them in a predefined format. This calculation is typically referred to as ordinal pattern encoding. The length of the subsequence we consider is the query order of the encoding problem. Ordinal pattern encoding is time-consuming for large time series data or high query orders.

The encoding procedure is computationally intensive in terms of arithmetic operations, but it only requires sequential access to the time series data. We consider it beneficial to encode ordinal patterns with a pipelined data engine using a reconfigurable acceleration device. The abundant logical resources in modern reconfigurable devices enable the arithmetic operations to be efficiently parallelised and pipelined. Moreover, it is practically attainable to use a large off-chip memory to store the time series data. As long as the memory provides adequate reading bandwidth for sequential access, it may feed the data to the accelerator efficiently without causing data starving.

B. *Accelerating Time Series Processing*

Hardware acceleration for time series data processing has not been well-studied. It is only in recent researches that acceleration systems based on GPUs and FPGAs are proposed to process time series data.

Guo and Luk [1] classifies acceleration solutions for time series processing into two categories namely pattern matching accelerators and correlation analysis accelerators. The first type of accelerators focus on matching and aligning problems for time series. For instance, Sart et al. [2] propose acceleration solutions to speed up dynamic time wrapping (DTW) for sequences using GPUs and FPGAs. The aims of these studies are to solve sequence matching problems which do not involve the discovery of underlying patterns of time series data from a statistical perspective. The second type of accelerators aim at discovering the self-correlation in univariate time series or the cross-correlation between dimensions in multivariate time series. For instance, Gembris et al. [3] present a real-time system to detect correlations among multiple medical imaging signals using GPUs.

Ordinal pattern encoding is computationally demanding, but it belongs to neither of the above two categories. The only acceleration solution we know is based on a CPU platform [4]. They accelerate the calculation of permutation entropy, which is a particular ordinal analysis technology. They claim that their experimental implementation to process around 3×10^6 data entries with query order 3 in one second.

III. Two-Level Ordinal Pattern Encoding

We propose a two-level encoding scheme for ordinal patterns for ease of hardware computation. The first level of encoding produces an intermediate representation for an ordinal pattern without sorting. The second level of encoding compresses the intermediate representation to a compact final result. In this section, we describe the two levels of encoding scheme and discuss their properties.

A. *Sorting-Free Ordinal Pattern Encoding*

Permutations are natural representations of ordinal patterns, while sorting is a straightforward way to encode the ordinal pattern of a sequence into a permutation. Specifically, one may first store the data entries in value-index pairs, and then sort these pairs by their values. The encoded permutation can be obtained by collecting the indexes sequentially. For instance, let $(6, 8, 2)$ be a subsequence in a time series. We first associate the index of each number to the number itself and obtain $(6 - 1, 8 - 2, 2 - 3)$. Sorting the pairs by the value, we get $(2 - 3, 6 - 1, 8 - 2)$. Therefore, the ordinal pattern in the permutation representation is $(3, 1, 2)$.

Although this sorting-based method is straightforward, we propose to avoid sorting in our solution because we consider sorting unsuitable for reconfigurable hardware. There are two types of sorting algorithms namely comparison-based sorting and non-comparison-based sorting.

Non-comparison-based sorting algorithms usually have better time complexity compared with comparison-based algorithms. Nevertheless, they may not be effectively applicable to real-world time series data. The amount of fast memory required by a non-comparison sorting algorithm depends on the number of distinct values in the sequence. The number of distinct values may be as large as the length of the time series data, but the amount of fast memory in a hardware platform is usually very limited.

Comparison-based sorting algorithms are mathematically acceptable for time series data. There are papers adopting reconfigurable architectures for sorting sequences [5], [6], [7], [8]. However, these architectures are typically expensive in terms of hardware resources. This is because, to sort a sequence of n values, the theoretical lower bound of the number of comparison operations is $n \log_2 n$. In terms of hardware computation, although it is possible to use any number of comparators to perform the computation, one needs to deploy at least $n \log_2 n$ comparators to make the design fully pipelined. Furthermore, one needs to plan additional data paths and buffers to support at least n data movement operations.

To avoid sorting, we first encode the ordinal pattern of a sequence into a Lehmer code rather than a permutation. Let $x = (x_1, \ldots, x_n)$ be a sequence of length n, the Lehmer code of x is also a sequence with length n in the form $\mathfrak{L}(x) = (l_1, \ldots, l_n)$ where

$$l_i = \#\{x_j : i < j, x_j < x_i\} \tag{1}$$

and $\#S$ is the cardinality of set S.

In other words, l_i is the number of entries in the sequence appearing after x_i and smaller than x_i. For instance, the Lehmer code of the sequence $x = (3, 9, 5, 1, 7, 4, 8)$ is $\mathfrak{L}(x) = (1, 5, 2, 0, 1, 0, 0)$. One property of this encoding scheme is that one ordinal pattern uniquely maps to a Lehmer code. In other words, the Lehmer code may serve as an alternative to the permutation in terms of representing ordinal patterns.

Divergent algorithms are available for encoding the ordinal pattern of a sequence to a Lehmer code. For instance, it is easy to compute a Lehmer code from the corresponding permutation representation, but this strategy brings no improvement in efficiency. One may also encode a sequence by calculating Equation 1 for each entry, but this makes no fundamental difference from sorting the sequence in terms of resource consumption.

We aim to develop an encoding scheme that can effectively use the on-chip hardware resources of a reconfigurable device. Therefore, we hope to parallelise as many encoding operations as possible. Two properties of the Lehmer code [9] draw our attention:

1) If the Lehmer code of $x = (x_1, \ldots, x_n)$ is $\mathfrak{L}(x) = (l_1, \ldots, l_n)$, then the Lehmer code of $x' = (x_2, \ldots, x_n)$ must be $\mathfrak{L}(x') = (l_2, \ldots, l_n)$. In other words, a Lehmer code entry l_i is independent of all data entries x_k such that $k < i$. Hence the removal of data entry x_1 directly result in the removal in the corresponding Lehmer code entry l_1.
2) If the Lehmer code of $x' = (x_2, \ldots, x_n)$ is $\mathfrak{L}(x') = (l_2, \ldots, l_n)$, then the Lehmer code of $x'' = (x_2, \ldots, x_n, x_{n+1})$ must be in the form $\mathfrak{L}(x'') = (l_2^+, \ldots, l_n^+, 0)$, where

$$l_i^+ = \begin{cases} l_i + 1 & \text{if } x_i > x_{n+1} \\ l_i & \text{otherwise} \end{cases} \tag{2}$$

In other words, if we append one entry to a data sequence, the corresponding Lehmer code entry must be zero because it is structurally the last data entry. If an entry x_i in the original data sequence is larger than the newly appended entry, then the Lehmer code entry l_i needs to grow by one because a new inversion is produced.

The two properties are key insights for our hardware design. They enable the possibility for efficient hardware-based ordinal pattern encoding. The sequences we encode are not isolated. These sequences are subsequences from large time series data. Therefore, as long as we encode the sequences in their natural temporal order, a sequence to be encoded is largely overlapped with the previous encoded sequence. The above two properties enable us to encode a sequence by updating the last encoding result. Let $x = (x_1, \ldots, x_T)$ be a time series of length T, and let n be the query order. Assume that the Lehmer code of a sequence $(x_{k+1}, \ldots, x_{k+n})$ has been computed. We can easily obtain the Lehmer code of $(x_{k+2}, \ldots, x_{k+n+1})$ using the two properties.

To encode the ordinal pattern of a sequence with length n, we only need $n - 1$ comparison operations and at most $n - 2$ addition operations. This resource consumption is significantly less than the sorting-based method. Therefore, given a fixed amount of hardware resources, it is expected that we can parallelise more encoding operations with Lehmer codes than using permutations.

B. Compressed Ordinal Pattern Encoding

In a typical ordinal patterns analysis problem, encoding is only the first step. The encoding results need to be further processed in a general-purpose computer or another hardware engine. Hence, it is needed to transmit the encoding result from the reconfigurable accelerator to a different device. To avoid potential data bandwidth bottleneck, the representation of an ordinal pattern should be as compact as possible to allow more data to be transferred per unit time.

We compress the Lehmer code into an unsigned integer using the factorial number system [10], [11]. Let $\mathfrak{L}(x) = (l_1, \ldots, l_n)$ be a Lehmer code of order n. We know that each entry l_i must be a non-negative integer no larger than $(n - i)$. This is because there are no more than $(n - i)$ data entries appearing after x_i. This property implies that a Lehmer code can be treated as a number in the factorial number system.

$$\mathfrak{C}(x) = \overline{l_n l_{n-1} \ldots l_2 l_1}_! = \sum_{i=1}^{n} l_i \cdot (i - 1)! \tag{3}$$

For example, the Lehmer code $(2, 3, 1, 0, 0)$ can be treated as $\overline{23100}_!$ in the factorial number system, which equals to $2 \times 4! + 3 \times 3! + 1 \times 2! + 0 \times 1! + 0 \times 0! = 68$ in decimal or 1000100 in binary. In other words, we can use an unsigned integer to represent an ordinal pattern.

To analysis the code compactness problem with a quantitative approach, we measure the minimum number of bits required to store or transmit a single encoding result for different encoding schemes.

- The compactness of the permutation representation is not satisfactory. To store a permutation of order n, one needs to store n unsigned integers, each of which is in the range $[1..n]$. Each integer requires $\lceil \log_2 n \rceil$ to store. As a result, the number of required bits is

$$b_P(n) = n \lceil \log_2 n \rceil \tag{4}$$

- The compactness of the Lehmer code representation is better than the permutation representation. As we have discussed, each entry l_i of a Lehmer code $\mathfrak{L}(x)$ must be in the range $[0..(n-i)]$. It is unnecessary to record the n-th entry because it is is always zero. Aside from the n-th entry, the i-th entry takes $\lceil \log_2(n-i+1) \rceil$ bits

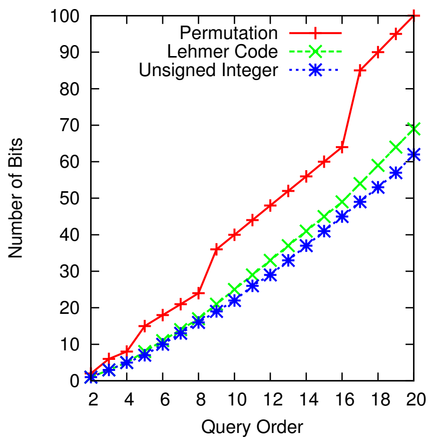


Fig. 1: Number of bits required by each encoding scheme

to represent. As a result, the number of bits required by Lehmer code is

$$b_L(n) = \sum_{i=1}^{n-1} \lceil \log_2(n-i+1) \rceil \tag{5}$$

- The integer representation of the Lehmer code is more compact than the other two methods. For any given positive query order n, the code must be an non-negative integer less than $n!$. Therefore, the number of bits required to represent an ordinal pattern is

$$b_C(n) = \lceil \log_2(n!) \rceil \tag{6}$$

We plot $b_P(n)$, $b_L(n)$ and $b_C(n)$ against the query order n in Fig. 1. It can be observed form the figure that $b_C(n) \leq b_L(n) \leq b_P(n)$ for all $1 \leq n \leq 20$. In fact, we can mathematically prove that this inequality holds for all query orders. Moreover, we have proved that the number of bits taken by a code in the unsigned integer representation is the theoretical lower bound. In other words, the encoding results from any other encoding scheme cannot be more compact than the results from the proposed encoding scheme. Therefore, by translating a Lehmer code to an unsigned integer using the factorial number system, we actually compress the code with the best possible compression ratio. This compactness of the compressed code guarantees the lowest possible bandwidth consumption from the reconfigurable accelerator to another device.

The aim of encoding is to model the distribution of ordinal patterns. Therefore, a necessary post-processing procedure is to build histograms from the encoding results. The design of an efficient histogram algorithm relies on the knowledge of the set of all distinct encoding results. A designer may take advantage of this set to simplify the design of the histogram.

The integer representation of Lehmer code ensures that the permutation pattern of a sequence uniquely maps to an unsigned integer in the range $[0..(n!-1)]$. In particular, sequences in ascending order correspond to 0; sequences with length n in descending order correspond to $(n!-1)$. To build a histogram, one only needs to maintain the counts of ordinal patterns in a zero-based array of size $n!$ indexed by the encoding result.

In contrast to the integer representation, it is difficult to build histograms for the other two representations. While it is still possible to maintain the counts using an array indexed by the encoding result, the array would be significantly longer than that of the integer representation, because the maximum encoding result is larger. In this case, one has to carefully design a hash table for the histogram to reduce memory consumption [12]. The hash table operations take more time than the array operations of the integer representation.

IV. Pipelined Reconfigurable Accelerator for Ordinal Pattern Encoding

We develop a hardware mapping method for the ordinal pattern encoding scheme discussed in the previous section taking advantage of the hardware-friendly properties. In this section, we first illustrate the structure of the architecture and explain the functions of the components. We then describe an experimental implementation based on a commercial FPGA acceleration platform.

A. *Pipelined Datapath*

We map the proposed two-level encoding scheme to a pipelined datapath. The structure of the datapath is shown in Fig. 2. The design parameter $\aleph$ is the maximum query order allowed. The time series data flows form the data source to a first-in-first-out (FIFO) buffer for reuse. The data in the FIFO buffer is then encoded as an unsigned integer.

The major part of the architecture consists of four components namely the data buffer, the increment decider, the Lehmer code reviser and the compressor. The first three components correspond to the first level of encoding where the time series subsequence in the data buffer is encoded as a Lehmer code. The fourth component corresponds to the second level of encoding where the Lehmer code is compressed to an unsigned integer. Structures and functions of the four components are as follows:

1) The data buffer is a first-in-first-out buffer that caches a sequence of $\aleph$ consecutive data elements of the time series data. One may customise on-chip block RAMs in the reconfigurable hardware to implement this buffer. At the end of each cycle, the content of the leftmost unit is discarded. Each storage unit except the rightmost one in the FIFO buffer takes the data from its right neighbour. The rightmost unit reads one entry from the data stream.
2) The increment decider is a group of $(\aleph - 1)$ comparators. These comparators correspond to the comparing operation in Equation 2. Each comparator takes two inputs. Every unit of the data buffer except the rightmost one contributes one input of the comparator. The rightmost unit in the data buffer serves as the other input of each comparator. A comparator is expected to produce 1 if the input from the rightmost buffer is smaller than the other input.
3) The Lehmer code reviser consists of $(\aleph - 1)$ adders and $(\aleph - 1)$ buffer units. The buffer units caches a Lehmer code of the latest encoded sequence. Each unit of the buffer is an input of an adder, while the output of the increment decider provides another

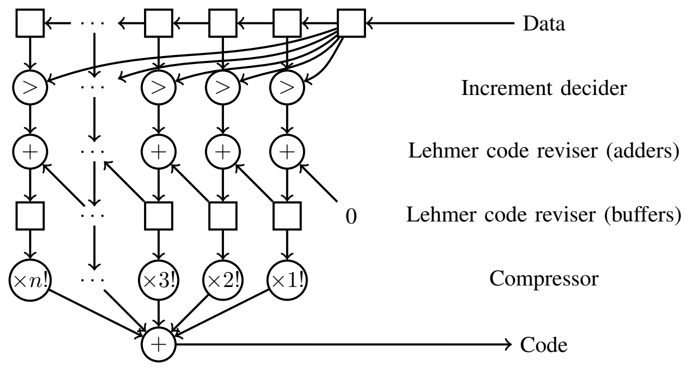


Fig. 2: Structure of the datapath

input. This component is in charge of updating the Lehmer code using Equation 2.

4) The compressor is composed of $(\aleph - 1)$ multipliers and a single adder. This structure corresponds to the compressing operation in Equation 3. Each multiplier multiplies its input by the factorial of a constant integer n in $[1..\aleph]$. The adder then sums up the products of all constant integers.

Before using the pipeline to encode sequences, we allow the user to disable multipliers in the compressor by forcing their output to be zero. When the query order n is less than $\aleph$, one needs to disable the first $(\aleph - n)$ constant multipliers before streaming data into the pipeline.

Suppose the Lehmer code cache is storing the Lehmer code of the current sequence in the data cache. When the data source feeds a new data entry to the data cache, the encoding process for the sequence in the data cache begins. The Lehmer code reviser obtains the sequence from the data cache and revise the code using Equation 2. Then the compressor fetches the code from the Lehmer code buffer and compress it to an unsigned integer using the factorial number system. Finally the reconfigurable accelerator sends the compressed code to the host computer.

When implementing the proposed architecture, an engineer may conduct a series of optimisations to maximise the performance. For instance, the engineer may enable fully pipelined hardware execution by inserting buffers along the datapaths and by rescheduling the data stream. In addition, we note that it is unnecessary to use standard multipliers in the datapath. One input of each adder is the output of a comparator. This signal must be either zero or one. Therefore, an adder is satisfactory as long as it can add an integer by one. Furthermore, one input of the multiplier must be the factorial of a constant integer. As a result, a constant integer multiplier [13], [14], an input of whom must be a predefined constant, is competent for the multiplication operation. These constant multipliers consume less hardware resources than fully functional ones.

B. An Experimental Implementation

We build an FPGA implementation of the proposed accelerator. The hardware platform we use is a Maxeler MAX3 acceleration card. This card is equipped with a Xilinx Virtex-6 SX475T FPGA and 48GB of DRAM. The card is installed in a host computer with four Intel i7-870 CPU cores running at 2.93GHz and 16GB DDR3 memory. Each CPU core contains two logical cores with Intel hyper-threading technology. The acceleration card communicates with the host computer via an 8-lane PCI Express 2.0 interface. The clock frequency of the FPGA is set to 100MHz. The hardware design is described in the MaxJ language and compiled to VHDL with Maxeler MaxCompiler. The CPU code in the host computer is written in the C programming language with the OpenMP library.

The maximum query order for our implementation is $\aleph = 12$. One consideration behind this setting is that appropriate query orders for real-world problems are typically small. The order 7 is sufficiently high for many real-world problems [4], [15]. Moreover, we are unable to find any problem that requires a query order larger than 12 in terms of statistics. The other consideration is that 12 is the highest order such that an encoding result fits a 64-bit unsigned integer. Therefore, with this setting, we may keep the simplicity and efficiency of the implementation by avoiding arbitrary-precision arithmetic.

It is possible to promote data parallelism by running multiple datapaths in parallel. A designer may use fewer datapaths to fit a smaller FPGA, or more datapaths to exploit available resources. In our experimental implementation, we deploy 6 datapaths in the FPGA chip to use up the I/O bandwidth.

The time series data are recorded in the IEEE single precision floating point number format. In real-world applications, the ordinal encoder receives the data stream from a data source. To simulate this situation, we load the data to the DRAM of the acceleration card and treat the DRAM as the data source. The encoding results are transmitted from the acceleration card to the main memory of the host computer. The data bandwidth consumption from the on-board DRAM to the FPGA chip is around 2.24GB/s. The encoding results are represented in 32-bit unsigned integers. They are transmitted to the host computer via the PCI Express interface. The memory bandwidth consumption for result transmission is also around 2.24GB/s. The whole design only consumes 20.36% of fine-grained logic, 4.17% of DSPs and 10.01% of block RAM in the FPGA chip.

V. Experimental Evaluation

We conduct an experimental evaluation to test the performance of the proposed architecture. In this section, we first describe the experimental configuration, and then discuss the performance results.

A. Experimental Configuration

To make a fair comparison, we only evaluate the systems with identical post-processing efforts. Therefore, we do not compare the proposed architecture with those based on the permutation representation or the Lehmer code representation. In particular, we focus on the hardware implementation described in section IV-B and a software implantation on a CPU platform. To make a fair comparison, we implement the CPU version on the host computer of the FPGA accelerator. We manually optimise the code and compile it using the Intel C/C++ compiler with the highest compilation optimisation. Since the performance of an encoding engine is not related to the distribution of the data, we use randomly generated time series data to test our proposed architecture.

We validate the correctness of our implementation using the same data for performance evaluation. We compare the histogram generated by our system with the one produced by the MATLAB implementation described in [16]. We observed no difference between the two histograms in all tests.

B. Performance of Acceleration

The performance results are shown in Fig. 3. The first two plots record the execution times in ordinal pattern encoding for query orders $n = 6, 12$. To observe the performance for different data size scales, we generate data with lengths from 10^5 to 10^9. Lengths of time series and the execution times are plotted in log scale with base 10 in these figures. A small difference along the vertical axis means a huge difference in execution time. The time of loading data to the DRAM is not considered because in real-world problems, the encoding system may obtain data directly from a data source without loading the data into memory.

The last two plots in Fig. 3 show the speedups of the FPGA system over the CPU system with one thread to eight threads. The speedup of the FPGA system is higher for larger time series data because the execution time of the FPGA system includes an initialisation time. This time becomes insignificant when the data set is large. It is expected that a speedup reaches its maximum value when the data is sufficiently large and the initialisation time becomes negligible.

For the same data size, the speedups with $n = 12$ are higher than those with $n = 6$. We set $\aleph = 12$ for each datapath. Therefore, when $n = \aleph = 12$, all computational resources in the architecture participate in the computation. In contrast, when $n = 6$, the system does not work at its full capacity because $\aleph - n = 6$ multipliers are disabled in each datapath.

The maximum speedup is observed when the query order n is 12 and the data size reaches 10^9. In this case, the FPGA implementation is respectively 62 times, 33 times, 19 times and 12 times faster than the CPU implementation running on one thread, two threads, four threads, and eight threads. We estimate the energy consumption of the workstation using an external energy meter, the FPGA implementation is also 22 times and 4 times more energy-efficient than the CPU implementation with one thread and eight threads.

VI. Case Study: Calculating Permutation Entropy

Permutation entropy [15] is a real number that measures the statistical complexity for a time series. This quantity is similar to the Shannon entropy [17] for non-temporal data. However, they are very different in a statistical sense since the permutation entropy contains information about unique features of time series data such as self-coherence. The permutation entropy is widely used to detect dynamic changes [18] and to estimate the predictability [19] for time series. However, due to the lack of efficient ordinal pattern encoders, the calculation of this entropy, particularly for high-order queries, is computationally demanding.

A. Definition and Applications

One may estimate the probability distribution of all ordinal patterns in a time series using their relative frequencies. For each unique ordinal pattern π_m, the relative frequency is

$$p(\pi_m) = \frac{\#\{t : \mathfrak{F}(x_{t+1}, \ldots, x_{t+n}) = \pi_m\}}{T - n + 1} \tag{7}$$

where $\mathfrak{F}(b_1, \ldots, b_n)$ is the encoding result of the ordinal pattern of $(b_1, \ldots, b_n)$. The permutation entropy of query order n, denoted by $H(n)$, is the information entropy of the distribution of ordinal patterns expressed by

$$H(n) = -\sum_{m=1}^{M} p(\pi_m) \log_2 p(\pi_m) \tag{8}$$

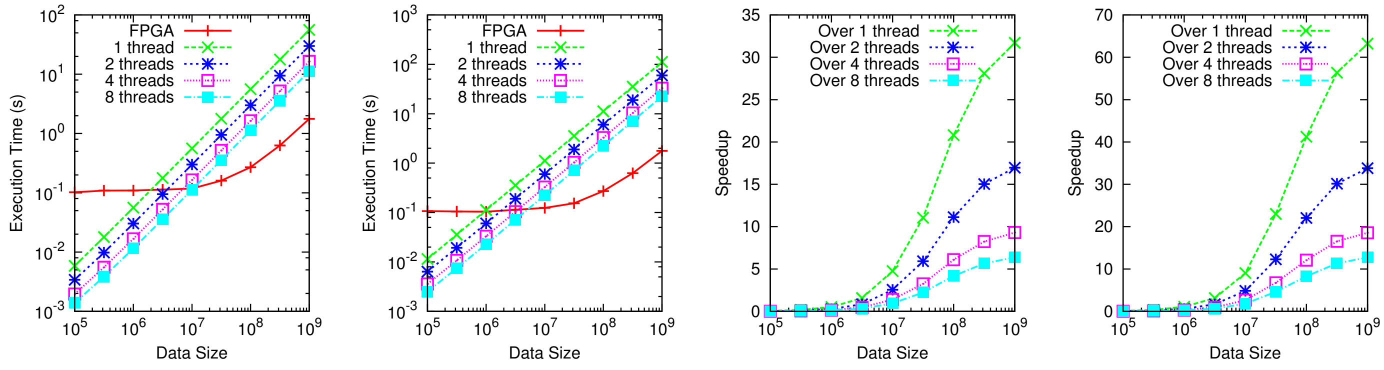
For instance, the time series $x = (6, 8, 2, 4, 7, 3)$ contains four contiguous sequences with length 3, namely $(6, 8, 2)$, $(8, 2, 4)$, $(2, 4, 7)$ and $(4, 7, 3)$. The ordinal patterns for the four sequence are respectively $(3, 1, 2)$, $(2, 3, 1)$, $(1, 2, 3)$ and $(3, 1, 2)$. There are three unique ordinal patterns namely $\pi_1 = (3, 1, 2)$, $\pi_2 = (2, 3, 1)$ and $\pi_3 = (1, 2, 3)$. The ordinal pattern π_1 appears twice while each of π_2 and π_3 appears only once. Therefore, the relative frequencies for the three unique patterns are $p(\pi_1) = 0.5$, $p(\pi_2) = 0.25$ and $p(\pi_3) = 0.25$. The permutation entropy with query order 3 is $H(3) = -(0.5 \log_2 0.5 + 0.25 \log_2 0.25 + 0.25 \log_2 0.25) = 1.5$.

Permutation entropy is a widely used complexity measure for time series. A major application area of this measure is bioscience signal processing. For instance, Olofsen et al. [20] calculate the permutation entropy of the electroencephalogram (EEG) in order to quantify the effect of anaesthetic drug. Frank et al. [21] use permutation entropy to analyse heartbeat sequences. They conclude that permutation entropy significantly improves the classification accuracy of fatal behavioural states. Nicolaou et al. [22] use permutation entropy of the EEG as a feature to characterise different stages of sleeping. Another application area of permutation entropy is financial computing. For instance, Zunino et al. [23] propose to use this measure to describe the degree of stock market inefficiency. They also successfully use this measure to predict the stage of stock market development. Ruiz et al. [24] suggest the use of a modified version of permutation entropy to estimate uncertainty and volatility in markets. Ortiz-Cruz et al. [25] use permutation entropy along with other entropy measures to analyse the dynamics of crude oil market and discuss the relationship between such measures and economic conditions.

B. System Design

We divide the calculation of permutation entropy into three major operations: (i) ordinal pattern encoding where ordinal patterns of all sequences are encoded; (ii) histogram construction where a histogram is built to store the count of all unique ordinal patterns following Equation 7; and (iii) entropy computation where the distribution of ordinal patterns is extracted from the histogram and the permutation entropy is calculated by Equation 8.

The ordinal pattern encoding is the most time-consuming procedure. We accelerate this part using the architecture proposed in Section IV. The other two operations are unsuitable



(a) Execution Time, n=6 (b) Execution Time, n=12 (c) Speedup, n=6 (d) Speedup, n=12

Fig. 3: Performance results

for reconfigurable architectures. Both calculations rely on a histogram, the maintenance of which needs random memory access. It is impossible to avoid the randomness because one may never predict the encoding result of the next ordinal pattern.

Fast memory resources such as on-chip block RAM in FPGAs are able to support efficient random memory access. However, the storage capacity of the fast memory resources in a commercial reconfigurable device is usually far too small even for a moderate-sized histogram. The number of entries in the histogram H is no more than the number of distinct ordinal patterns of order n in the time series. Specifically, the number of entries must be fewer than the minimum of (i) the number of subsequences in the time series and (ii) the number of permutations of order n. In other words, if we denote the maximum number of entries in the histogram by $|h|$, then we have

$$|h| \leq \min\{(T-n+1), n!\} \tag{9}$$

For instance, assume that a histogram entry only stores the count of the corresponding ordinal pattern, and each count fits a 32-bit unsigned integer. To evaluate a permutation entropy with query order 5, the fast memory resources required by the histogram far exceed the available resources in a Xilinx V6-SX475T FPGA device.

To avoid random memory access to off-chip memory from the reconfigurable device, we propose to encode ordinal patterns in the reconfigurable accelerator but finalise the remaining two calculations in the general-purpose host computer. The accelerator receives the time series data stream and encodes the ordinal pattern for each sequence. Once the accelerator finishes producing the encoding result for a subsequence, it transmits the resulting code to the host computer. The computer updates the histogram when it receives a code from the reconfigurable accelerator. At all times, the histogram maintains the counts for each possible code. Once the host computer receives a code from the accelerator, it increases the count of this code by one.

When the host computer has received and stored the codes of all sequences in the time series data, it is then possible to calculate the permutation entropy. We extract the distribution of ordinal patterns by calculating the relative frequency for each unique code in the histogram. We then compute the permutation entropy by taking the standard information entropy following Equation 8.

C. Evaluation with Real-World Data

We use two large data sets to test the performance of the proposed hybrid permutation entropy calculator. One data set is an EEG data set with 3750 records, each containing two time series. Each time series has 10240 entries. Another data set is a single time series taken from the foreign exchange market. This time series records the exchange rate between the United States dollar and British pound sterling in January 2014 with 100Hz sample rate. The length of this time series is 267840000.

We calculate the permutation entropy for all the time series in each data set. We then record the total execution time for the two experimental systems. All operations in the CPU platform are executed on a single core. The performance results are shown in Table I. The execution times for the CPU-only system and the CPU+FPGA system are respectively recorded in the third and fourth column in the table. The fifth column, titled ‘Speedup(L)’ shows the limited speedups of the CPU+FPGA system over the CPU-only system considering both the histogram construction time and the entropy calculation time. These speedups are limited by the memory bandwidth of our experimental device. The sixth column with the title ‘Speedup(I)’ records the ideal speedups by assuming that the encoding operation takes negligible time. A value in this column is the theoretical upper bounds of the corresponding experimental configuration if the performance of the CPU remains unchanged. The last column records the gaps between the limited speedups to the ideal speedups in percentage.

The CPU+FPGA system achieves 5-11 times speedup over the CPU-only system in the tests. We observe lower speedups in permutation entropy calculation than ordinal pattern encoding. This is because the histogram update procedure in the host computer takes a considerable amount of time. We also observe that the best speedup is not obtained with the highest query order. This is different from the observation concerning ordinal pattern encoding. This is because, when the query order is higher, the construction of histogram takes longer

TABLE I: Performance for permutation entropy calculation

Data	Order	CPU	C+F	Speedup(L)	Speedup(I)	Gap
EEG	3	1.5405	0.2591s	5.9 times	13.5 times	56%
EEG	6	2.8846	0.2594s	11.1 times	24.8 times	55%
EEG	9	4.2769	0.3902s	10.9 times	17.3 times	36%
EEG	12	6.1952	1.1484s	5.3 times	6.1 times	12%
$/£	3	5.3878s	0.8904s	6.0 times	13.5 times	55%
$/£	6	10.0763s	0.9029s	11.1 times	24.9 times	55%
$/£	9	15.0903s	1.3626s	11.0 times	17.2 times	36%
$/£	12	21.4257s	3.9963s	5.3 times	6.1 times	12%

time than small query orders. Therefore, the overall speedup is negatively affected even if we have high speedup in ordinal pattern encoding.

Although the performance of the experimental system is limited by the host computer, it is still significantly faster than existing open-source software solutions. For example, Unakafova et al. [4] discuss an acceleration solution for permutation entropy using MATLAB. This software is the only implementation we can find that addresses the computational efficiency problem. In the experiment with EEG data set, the solution in [4] fails to finish running in 4 hours when the query order reaches 12. In this case, our system terminates in 1.1484 seconds. In other words, our solution is conservatively 12539 times faster than the one proposed in [4].

VII. Conclusion and Future Work

This paper presents the first reconfigurable acceleration solution for ordinal pattern encoding. We begin by proposing a two-level permutation encoding scheme to avoid sequence sorting operations in the accelerator, and to reduce the amount of data transmission. We then develop a hardware mapping method by promoting data reuse, by parallelising arithmetic operations, and by pipelining the data path. We also build an experimental system and conduct an evaluation to compare its performance with software solutions. In addition, we conduct a case study applying our proposed architecture to the calculation of permutation entropy.

Future work includes extending the architecture to cover other ordinal analysis applications besides permutation entropy calculation. One may further customise the architecture when integrating it with particular applications.

Acknowledgement

This work is supported in part by the China Scholarship Council, by Maxeler University Programme, by the European Union Seventh Framework Programme under grant agreement number 257906, 287804 and 318521, by UK EPSRC, and by Xilinx.

References

[1] C. Guo and W. Luk, "Accelerating HAC estimation for multivariate time series," in *Proceedings of International Conference on Application-specific Systems, Architectures and Processors*, 2013.

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

[3] D. Gembris, M. Neeb, M. Gipp, A. Kugel, and R. Männer, "Correlation analysis on GPU systems using NVIDIA's CUDA," *Journal of real-time image processing*, vol. 6, no. 4, pp. 275–280, 2011.

[4] V. A. Unakafova and K. Keller, "Efficiently measuring complexity on the basis of real-world data," *Entropy*, vol. 15, no. 10, pp. 4392–4415, 2013.

[5] J. Harkins, T. El-Ghazawi, E. El-Araby, and M. Huang, "Performance of sorting algorithms on the src 6 reconfigurable computer," in *Proceedings of International Conference on Field-Programmable Technology*, 2005, pp. 295–296.

[6] J. Martinez, R. Cumplido, and C. Feregrino, "An FPGA-based parallel sorting architecture for the burrows wheeler transform," in *International Conference on Reconfigurable Computing and FPGAs*, 2005.

[7] D. Mihhailov, V. Sklyarov, I. Skliarova, and A. Sudnitson, "Parallel FPGA-based implementation of recursive sorting algorithms," in *International Conference on Reconfigurable Computing and FPGAs*, 2010.

[8] V. Sklyarov, I. Skliarova, D. Mihhailov, and A. Sudnitson, "Implementation in FPGA of address-based data sorting," in *International Conference on Field Programmable Logic and Applications*, 2011.

[9] K. Keller and M. Sinn, "Ordinal analysis of time series," *Physica A: Statistical Mechanics and its Applications*, vol. 356, no. 1, pp. 114–120, 2005.

[10] M. Bóna, *Combinatorics of permutations*. CRC Press, 2012.

[11] D. E. Knuth, *Art of computer programming, volume 3: Sorting and searching, Second Edition*. Addison-Wesley, 1998.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT Press, 2001.

[13] T. Kean, B. New, and B. Slous, "A fast constant coefficient multiplier for the xc6200," in *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, vol. 1142, pp. 230–236.

[14] F. de Dinechin and V. Lefèvre, "Constant multipliers for FPGAs," in *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, 2000, pp. 167–173.

[15] C. Bandt and B. Pompe, "Permutation entropy: a natural complexity measure for time series," *Physical Review Letters*, vol. 88, no. 17, p. 174102, 2002.

[16] G. Ouyang, J. Li, X. Liu, and X. Li, "Dynamic characteristics of absence eeg recordings with multiscale permutation entropy analysis," *Epilepsy Research*, vol. 104, no. 3, pp. 246–252, 2013.

[17] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.

[18] Y. Cao, W.-w. Tung, J. Gao, V. A. Protopopescu, and L. Hively, "Detecting dynamical changes in time series using the permutation entropy," *Physical Review Series E*, vol. 70, pp. 046 217–046 217, 2004.

[19] X. Li, G. Ouyang, and D. A. Richards, "Predictability analysis of absence seizures with permutation entropy," *Epilepsy research*, vol. 77, no. 1, pp. 70–74, 2007.

[20] E. Olofsen, J. Sleigh, and A. Dahan, "Permutation entropy of the electroencephalogram: a measure of anaesthetic drug effect," *British Journal of Anaesthesia*, vol. 101, no. 6, pp. 810–821, 2008.

[21] B. Frank, B. Pompe, U. Schneider, and D. Hoyer, "Permutation entropy improves fetal behavioural state classification based on heart rate analysis from biomagnetic recordings in near term fetuses," *Medical and Biological Engineering and Computing*, vol. 44, no. 3, pp. 179–187, 2006.

[22] N. Nicolaou and J. Georgiou, "The use of permutation entropy to characterize sleep electroencephalograms," *Clinical EEG and Neuroscience*, vol. 42, no. 1, pp. 24–28, 2011.

[23] L. Zunino, M. Zanin, B. M. Tabak, D. G. Pérez, and O. A. Rosso, "Forbidden patterns, permutation entropy and stock market inefficiency," *Physica A: Statistical Mechanics and its Applications*, vol. 388, no. 14, pp. 2854–2864, 2009.

[24] M. d. C. Ruiz, A. Guillamón, and A. Gabaldón, "A new approach to measure volatility in energy markets," *Entropy*, vol. 14, no. 1, pp. 74–91, 2012.

[25] A. Ortiz-Cruz, E. Rodriguez, C. Ibarra-Valdez, and J. Alvarez-Ramirez, "Efficiency of crude oil markets: Evidences from informational entropy analysis," *Energy Policy*, vol. 41, pp. 365–373, 2012.