

Imperial College London
Department of Computing

Reconfigurable Predictive Systems for Event Streams

Ce Guo

September, 2016

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in
Computing of Imperial College London and the Diploma of Imperial College London

Declaration of Originality

I hereby declare that this thesis is my own original work. All sources used for the thesis have been fully and properly cited.

Copyright Declaration

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.

Acknowledgements

Firstly, I would like to thank my supervisor Professor Wayne Luk. It is a great honour for me to be one of his PhD students. I am very grateful that he granted me unconditional trust and freedom all the time, especially during the tough periods. I appreciate all his contributions of time, ideas, opportunities and funding to make my PhD experience productive and stimulating.

Besides my supervisor, I would like to thank Andreea-Ingrid Funie, Thomas Chau, David Thomas, Maciej Kruke, Tobias Becker, and Timothy Todman. They not only provided insightful advice and generous help but also are a great source of collaboration.

My sincere thanks also goes to the others in the Department of Computing at Imperial College London: James Arram, Pavel Burovskiy, Bridgette Cooper, Kit Cheung, Gabriel De Figueiredo Coutinho, Stewart Denholm, Paul Grigoras, Maciej Kurek, Liucheng Guo, Eddie Hung, Gordon Inggs, Adrien Le Masle, Nicholas Ng, Xinyu Niu, Shengjia Shao and Jinzhe Yang. All of them made my time at Imperial College London enjoyable and productive.

Abstract

The study of event streams involves modelling arrival times of discrete random events. Predictive systems for event streams infer information about future occurrences of random events. These systems are useful for various applications such as stock trading modelling and earthquake analysis, but the computational burdens limit their predictive power. This thesis addresses the design and optimisation of reconfigurable predictive systems for event streams.

The first contribution of this thesis is the reconfigurable acceleration solution for the calculations in the Hawkes point process models. We propose reconfigurable dataflow engines for intensity evaluation and likelihood evaluation for univariate and multivariate Hawkes point process models. We also establish an efficient collaboration scheme between the CPU platform and the reconfigurable accelerator to speed up parameter estimation.

The second contribution of this thesis is a novel predictive model for event streams. As the Hawkes point process models have limited predictive accuracy and low computational efficiency, we propose a predictive model for event streams using regression techniques. We derive the model from the intensity function of the Hawkes point process, but the final form of the proposed model works without point process models. A software system based on the proposed model reduces the prediction error by 3%–7% compared to a system based on the Hawkes process. A hardware-accelerated system based on the proposed model is 5–66 times faster in model fitting compared to the accelerated system based on the Hawkes process, while the two systems produce similar predictive accuracy.

The third contribution of this thesis is the design of two reconfigurable accelerators for time series analysis. One accelerator is for the estimation of correlation for multivariate time series. The other accelerator is for the ordinal pattern encoding for univariate time series. We design the two accelerators by transforming the calculations of the corresponding statistical metrics into pipeline-friendly forms. Compared to multicore CPUs, both accelerators show high efficiency for large time series data.

Contents

1	Introduction	12
1.1	Overview	12
1.2	Contributions	13
1.2.1	Reconfigurable Computing for Hawkes Process Models	14
1.2.2	Novel Predictive Model for Event Streams with Regression	14
1.2.3	Pipelined Dataflow Engines for Time Series Analysis	15
1.3	Publications	15
1.4	Thesis Organisation	16
2	Background	17
2.1	Predictive Systems for Event Streams	17
2.1.1	Event Streams	17
2.1.2	Predictive Systems for Event Streams	19
2.2	Reconfigurable Computing	21
2.2.1	Reconfigurable Computing with FPGAs	21
2.2.2	Dataflow Computing Platform	23
2.3	Intensity-Based Point Process Models	24
2.3.1	Point Processes	24
2.3.2	Self-Exciting Point Processes	26
2.3.3	Prediction with Point Process Models	27
2.4	Regression	29
2.4.1	Gradient Descent for Linear Regression	30
2.4.2	Reconfigurable Training Engines for Regression	31
2.5	Time Series Analysis	34
2.5.1	Time Series	34
2.5.2	Measure of Correlation	35
2.5.3	Ordinal Analysis	36
2.6	Summary	37

3	Reconfigurable Accelerators for Hawkes Process Models	39
3.1	Intensity Evaluation	40
3.1.1	Strategy of Acceleration	40
3.1.2	One-Pass Computation of Multiple Intensities	41
3.1.3	Hardware Design	43
3.1.4	Data Parallelism	44
3.1.5	Experimental Evaluation	44
3.2	Likelihood Evaluation for Univariate Data	47
3.2.1	Recursive Likelihood Evaluation	47
3.2.2	Hardware Design and Optimisation	51
3.2.3	Experimental Evaluation	52
3.2.4	Results and Discussion	52
3.3	Likelihood Evaluation for Multivariate Data	53
3.3.1	Straightforward Likelihood Evaluation	54
3.3.2	Straightforward Intensity Evaluation	56
3.3.3	Recursive Intensity Evaluation	57
3.3.4	Hyper-Recursive Intensity Evaluation	60
3.3.5	Hardware Design	62
3.3.6	Experimental Evaluation	64
3.4	Host-Accelerator Collaboration for Parameter Estimation	66
3.4.1	Straightforward Simulated Annealing	66
3.4.2	Eager Preparation	67
3.4.3	Experimental Evaluation	68
3.5	Summary	69
4	Event Stream Prediction with Regression	70
4.1	Approximate Intensity Evaluation	70
4.1.1	Step Function Approximation	74
4.1.2	Separators and Step Heights	77
4.1.3	Algorithms for Approximate Intensity Functions	79
4.2	Event Stream Prediction with Regression	83
4.2.1	Elimination of Point Process Models	83
4.2.2	Efficient Computation of Count Vectors	86
4.2.3	Data Preparation for Regression	90
4.2.4	Gradient Descent Engine for Linear Regression	91
4.3	Experimental Evaluation	93
4.3.1	Benchmarks	93
4.3.2	Experimental Configuration	94

4.3.3	Results and Discussion	96
4.4	Summary	98
5	Pipelined Data Flow Engines for Time Series Analysis	100
5.1	Long-Run Covariance Matrix Estimation	101
5.1.1	Analysis of the Straightforward Algorithm	101
5.1.2	Pipeline-Friendly Expression for Long-Run Covariance	102
5.1.3	Pipeline-Friendly HAC Estimation Algorithm	106
5.1.4	Hardware Design	107
5.1.5	Experimental Evaluation	110
5.2	Ordinal Pattern Encoding	113
5.2.1	Sorting-Free Ordinal Pattern Encoding	113
5.2.2	Compressed Ordinal Pattern Encoding	115
5.2.3	Pipelined Reconfigurable Accelerator	118
5.2.4	Experimental Evaluation	121
5.2.5	Case Study: Calculating Permutation Entropy	122
5.3	Summary	127
6	Conclusion	128
6.1	Summary of Achievements	128
6.1.1	Custom Computing for Hawkes Process	128
6.1.2	Novel Predictive Model for Event Streams	129
6.1.3	Reconfigurable Acceleration for Time Series	130
6.2	Future Work	130
6.2.1	Point Process Models with Non-Temporal Information	130
6.2.2	Training Engines for Supervised Learning	131
6.2.3	Acceleration of Time Series Analysis Techniques	132
6.2.4	Improving Reconfigurable Implementations and Their Evaluation	132
6.3	Reflection	133
A	General Settings of Experiments	148

List of Tables

2.1	Parameter estimation methods for Hawkes process models	29
3.1	Test data for intensity evaluation	45
4.1	Comparison of three intensity evaluation methods	76
4.2	Configuration of experimental predictive systems	96
4.3	Prediction error (mean absolute error)	97
4.4	Computational efficiency	98
5.1	Lehmer code for ordinal patterns with order 3	116
5.2	Example of encoding process	117
5.3	Computational efficiency for permutation entropy	127

List of Figures

2.1	An event sequence with three types of events	18
2.2	Framework of dataflow computing	23
2.3	A two dimensional Hawkes process	28
3.1	Data dependency of straightforward intensity evaluation	41
3.2	Data dependence pattern of proposed evaluation strategy	43
3.3	Basic architecture with a single glove	44
3.4	Exploiting data parallelism by combining g gloves	45
3.5	Experimental results on intensity evaluation	46
3.6	Basic modules	51
3.7	Q Module	52
3.8	Experimental results	53
3.9	Data dependency pattern in different intensity evaluation methods	56
3.10	Intensity evaluation cell	63
3.11	Likelihood evaluation accelerator	63
3.12	Computational efficiency for likelihood evaluation	65
3.13	Gantt chart of two host-accelerator collaboration schemes	67
3.14	Computational efficiency in parameter estimation	69
4.1	Exact and approximate intensities	82
4.2	Prediction by simulation	83
4.3	Elimination of simulation facilities	85
4.4	Completeness of history	90
4.5	Completeness of future	91
4.6	Data path for one iteration of SGD training	92
4.7	Segments taken from benchmarks	95
5.1	Structure of a bead	108
5.2	Proposed architecture for HAC estimation	108
5.3	Results on computational efficiency	111

5.4	Results on scalability	112
5.5	Number of bits required by each encoding scheme	118
5.6	Structure of the data path	120
5.7	Example execution of the data path	120
5.8	Computational efficiency results	123
5.9	Time series with different complexity	124

Chapter 1

Introduction

1.1 Overview

Event streams record occurrences of discrete random events in chronological order. Dynamic systems around the world produce a considerable volume of event stream data every day. These data contain rich information about the underlying mechanism of the data producer. Statistical patterns extracted from these data can help decision making. For instance, event stream modelling is attracting the attention of researchers and practitioners involved in seismology [1, 2], algorithmic trading [3, 4, 5] and neuroscience [6].

The analysis of event stream data includes a series of calculations that are computationally demanding, particularly for multivariate data. In this case, computational acceleration is in great need. In recent years, the efficiency problem has become increasingly severe, because the length and dimensionality of real-world data have been growing swiftly.

The objective of this research is to develop and optimise reconfigurable hardware designs to accelerate the computations in event stream prediction and the related statistical methods. In particular, the problems and techniques we study in this research are as follows:

- Among the problems in event stream analysis, we mainly focus on predictive modelling problems. Predictive modelling for event streams offers the opportunity to predict future situations based on historical data. For instance, a segment of the log of a website is an event stream data set containing information about behaviours and preferences of visitors. Such data set enables the development of models to predict the amount of web traffic and pre-arrange server resources accordingly.
- Among the application acceleration techniques, we focus on the design of dataflow engines based on reconfigurable hardware. Specifically, we describe an algorithm as a spatial layout of computational units on a data path. During the computation, data streams flow through the data path, so that the computational units alongside the data

path may apply operations on the segment of data flowing through the units.

The acceleration of calculations in event stream modelling is useful as the conventional techniques for event stream modelling have high complexity. For instance, when we use the Hawkes point process [7] to model data, it is necessary to estimate the parameters to fit a model to a data set. In the estimation process, an optimisation algorithm maximises the likelihood function by searching in the parameter space. The optimisation algorithm needs to evaluate the likelihood function frequently. Each evaluation of the likelihood function takes $O(N^2)$ time in general [8] where N is the number of events in the data. Therefore, even a single evaluation of the likelihood function may take a considerable amount of time when the number of events is large. Moreover, the number of evaluations of the likelihood function depends on the distribution of data and the settings of the search algorithm. There exists no upper bound for the number of likelihood evaluations to the best of our knowledge. As a result, even if the data volume is small, the total execution time for parameter estimation may still be long. For instance, the maximum likelihood evaluation in [9] with only 2308 events takes ten minutes on a desktop computer. When the data volume is large, it may take days or months to fit the model with sufficient accuracy using a desktop computer.

The biggest challenge in this research is that conventional techniques for event stream modelling involve operations that are difficult to accelerate using reconfigurable hardware, such as matrix inversion, sequence sorting, and random memory access. We address this challenge using two approaches. The first approach is to derive mathematical methods to transform the algorithms for conventional models to facilitate hardware design. The second approach is to develop new predictive models with efficient algorithms.

While data analysis systems can often benefit from the speed, simplicity and power efficiency of hardware implementations, hardware acceleration of event stream modelling has not received attention. Our work is the first to cover hardware acceleration of event stream modelling to the best of our knowledge.

1.2 Contributions

In this thesis, we present three contributions about hardware acceleration of event stream prediction. The first contribution is the hardware acceleration solutions for the Hawkes point process models. The second contribution is a novel predictive model for event streams. The third contribution is the design of reconfigurable accelerators for two techniques in time series analysis.

1.2.1 Reconfigurable Computing for Hawkes Process Models

Our first contribution focuses on the Hawkes self-exciting point process. Hawkes processes are point processes which capture the statistical patterns of random events. Data modelling with the Hawkes point process involves three major operations namely *intensity evaluation*, *likelihood evaluation* and *parameter estimation*. Intensity evaluation is the operation to evaluate the intensity function which quantifies the instantaneous occurrence rate of random events. Likelihood evaluation is the operation to calculate the likelihood value that measures how well a set of parameters fits a data set. Parameter estimation is the operation to find the best parameter set that fits a model to a data set.

Chapter 3 describes our hardware acceleration solutions for the three operations. We first develop dataflow engines to accelerate intensity evaluation, likelihood evaluation for univariate data, and likelihood evaluation for multivariate data respectively. In the design of the three accelerators, we simplify data dependency with mathematical transformation, rearrange the computation, and exploit data parallelism. We then accelerate parameter estimation by developing an efficient collaboration scheme between the host computer and the accelerator for likelihood evaluation.

1.2.2 Novel Predictive Model for Event Streams with Regression

One may use point process models to make predictions for event streams. However, we are still interested in alternative predictive models for two reasons:

- Conventional point process models may limit the predictive accuracy. For instance, a predictive system based on a Hawkes process model may give inaccurate predictions when the data set does not follow a self-exciting pattern.
- The parameter estimation for Hawkes process models has no analytical solutions. As a result, it may be difficult to find a satisfactory parameter set in reasonable time even if we can accelerate the estimation process using reconfigurable hardware.

Chapter 4 proposes a novel predictive model to address the accuracy and efficiency problems. We start by proposing an approximate intensity function for the Hawkes process with a counting technique. By analysing the approximate intensity function, we manage to design a predictive model for event streams based on the counting technique and regression analysis. The proposed predictive model works without point process models, which improves the computational efficiency and facilitates reconfigurable computing. We conduct an empirical study to compare the proposed model with point process models on the CPU and FPGA platforms. The proposed predictive model is comparable to the Hawkes point process models regarding predictive accuracy and computational efficiency.

1.2.3 Pipelined Dataflow Engines for Time Series Analysis

Time series analysis is a topic closely related to event streams. Similar to event stream data, time series are temporal data collected from random processes. However, unlike event streams, time series analysis is relatively well-studied. There exist a large number of classic techniques for time series. One may take advantage of these time series techniques to study event streams by converting event stream data to time series. Time series analysis techniques for long time series are computationally expensive.

Chapter 5 provides two dataflow engines for time series analysis. Each dataflow engine accelerates a time-consuming calculation. The first engine accelerates a correlation measurement technique for multivariate time series, namely the heteroskedasticity and autocorrelation consistent covariance matrix estimation, or the HAC estimation in short. HAC estimation can be used to verify the quality of predictive models. We design the HAC estimation engine by developing a hardware-friendly expression of the HAC estimator, by proposing algorithmic transformations to eliminate conditionals, and by promoting data reuse. The second dataflow engine accelerates a complexity analysis technique for univariate time series, namely ordinal pattern encoding. Ordinal pattern encoding can be used to quantify the statistical complexity and predictability of temporal data. We design the engine by eliminating sorting operations and by compressing the result to save memory bandwidth.

1.3 Publications

Parts of the work presented in this thesis have been published. The following publication contributes to the intensity evaluation engine for the Hawkes process in Chapter 3:

- **C. Guo**, W. Luk, E. Vinkovskaya, R. Cont, ‘Customisable pipelined engine for intensity evaluation in multivariate Hawkes point processes’. *ACM SIGARCH Computer Architecture News*, 41(5), 59-64.

The following two publications contribute to the likelihood evaluation engine for the Hawkes process in Chapter 3:

- **C. Guo**, W. Luk, ‘Accelerating parameter estimation for multivariate self-exciting point processes’. *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2014
- **C. Guo**, W. Luk, ‘Accelerating maximum likelihood estimation for Hawkes point processes’. *International Conference on Field Programmable Logic and Applications (FPL)*, 2013

The following publications contribute to the Heteroskedasticity and autocorrelation consistent covariance matrix estimation engine in Chapter 5:

- **C. Guo**, W. Luk, ‘Accelerating HAC estimation for multivariate time series’. *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2013, **Best Paper Award**
- **C. Guo**, W. Luk, ‘Pipelined HAC estimation engines for multivariate time series’. *Journal of Signal Processing Systems*, 77(1-2): 117-129, 2014

The following publication contributes to the ordinal pattern encoding engine in Chapter 5:

- **C. Guo**, W. Luk, S. Weston: ‘Pipelined reconfigurable accelerator for ordinal pattern encoding’. *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2014

The following publications are also published during my PhD studies, but they are not discussed in detail in this thesis:

- L. Guo, **C. Guo**, D. B. Thomas, W. Luk, ‘Pipelined genetic propagation’. *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2015, **Best Paper Nominee**
- J. Yang, **C. Guo**, W. Luk, T. Nahar, ‘Collaborative processing of least-square Monte Carlo for American options’, *International Conference on Field-Programmable Technology (FPT)*, 2014
- S. Shao, **C. Guo**, W. Luk, S. Weston, ‘Accelerating transfer entropy computation’, *International Conference on Field-Programmable Technology (FPT)*, 2014
- **C. Guo**, H. Fu, W. Luk, ‘A fully-pipelined expectation-maximization engine for Gaussian mixture models’. *International Conference on Field-Programmable Technology (FPT)*, 2012

1.4 Thesis Organisation

The rest of the thesis is organised as follows. Chapter 2 provides the background on event stream modelling, reconfigurable computing, intensity-based point processes, regression analysis and time series. Chapter 3 describes the first contribution of this thesis where we use reconfigurable hardware to accelerate intensity evaluation, likelihood evaluation and parameter estimation for Hawkes point process models. Chapter 4 presents the second contribution where we derive a new predictive model for event streams. Chapter 5 details the third contribution where we speed up two calculations in time series analysis. Chapter 6 concludes this thesis by recapping the achievements and discussing future work.

Chapter 2

Background

This chapter provides the background of this thesis, including the definitions of important concepts and discussions about existing techniques. Section 2.1 introduces predictive systems for event streams. Section 2.2 discusses reconfigurable computing techniques and our experimental platform. Section 2.3 covers intensity-based point process models, which are dominating modelling techniques for event streams. Section 2.4 illustrates the modelling and computation of regression models. Section 2.5 gives an overview of time series analysis.

2.1 Predictive Systems for Event Streams

The problems and techniques we study in this thesis are related to event streams. In this section, we provide a brief introduction to event streams and the related prediction problems.

2.1.1 Event Streams

Event streams are streams of discrete events [10, 11]. For ease of discussion, we assign an index to each event in chronological order [12]. In particular, we denote an event stream with M types of events by

$$\mathfrak{E} = ((t_0, l_0), (t_1, l_1), (t_2, l_2) \dots) \quad (2.1)$$

Each entry of the event stream is the record of an event in the form (t_i, l_i) where

- $i \in \mathbb{N}$ is the index of the event.
- $t_i \in \mathbb{R}$ is the occurrence time of the event. In this research, we assume that $t_i < t_{i+1}$ for all $i \in \mathbb{N}$.
- $l_i \in [0..(M-1)]$ is the type label of the event.

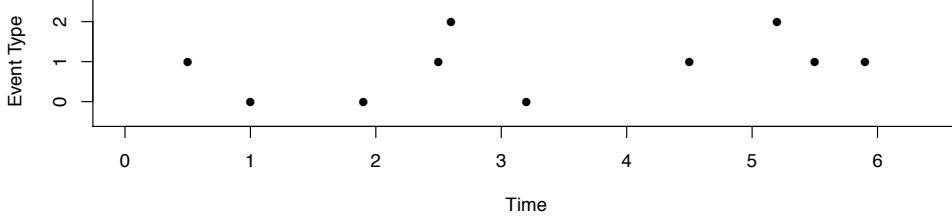


Figure 2.1: An event sequence with three types of events

An event stream has an infinite length, but an algorithm can only process a finite number of events in finite time. As a result, we analyse finite data segments taken from event streams. Specifically, we analyse *event sequences* [13, 14] which are finite continuous sub-sequences intercepted from event streams. For ease of discussion, let $\mathcal{E}(t_{\perp}, t_{\top})$ be an event sequence containing all the events collected from an event stream in the time interval $\{t' : t_{\perp} \leq t' < t_{\top}\}$. For instance, an event sequence taken in the time interval $\{t' : 0 \leq t' < 6.1\}$ is as follows:

$$\begin{aligned} \mathcal{E}(0, 6.1) = & ((0.5, 1), (1.0, 0), (1.9, 0), (2.5, 1), (2.6, 2), \\ & (3.2, 0), (4.5, 1), (5.2, 2), (5.5, 1), (5.9, 1)) \end{aligned} \quad (2.2)$$

We may visualise an event sequence by plotting the events in a two-dimensional Cartesian coordinate system [14]. The horizontal axis of the plot corresponds to the time, while the vertical axis describes the event type. Each point on the plane marks an event. For instance, a plot for the event sequence in Equation 2.2 is in Figure 2.1.

For ease of discussion, let $|\mathcal{E}(t_{\perp}, t_{\top})|$ be the number of events in the event sequence. Let $\mathcal{E}_m(t_{\perp}, t_{\top})$ be the sequence of all events in $\mathcal{E}(t_{\perp}, t_{\top})$ with type label m . For instance, given the event sequence in Equation 2.2, we have

$$|\mathcal{E}(0, 6.1)| = 10 \quad (2.3)$$

$$\mathcal{E}_0(0, 6.1) = ((1.0, 0), (1.9, 0), (3.2, 0)) \quad (2.4)$$

$$\mathcal{E}_1(0, 6.1) = ((0.5, 1), (2.5, 1), (4.5, 1), (5.5, 1), (5.9, 1)) \quad (2.5)$$

$$\mathcal{E}_2(0, 6.1) = ((2.6, 2), (5.2, 2)) \quad (2.6)$$

Models for event streams serve as the core components in a wide range of applications. For instance, Cowpertwait *et al.* [15, 16] propose an event stream model to analyse rainfall data. The model fits well with 23 years of historical daily data taken from the Thames catchment in the United Kingdom. Truccolo *et al.* [17] analyse the spiking behaviours of a neurone using

an event stream model. Other applications of event stream models include public security [18, 19], financial computing [4], seismology [1] and cash machine management [20].

2.1.2 Predictive Systems for Event Streams

The prediction problem of event streams is the problem of inferring information about the occurrences of events in the future. Let $\mathbf{ptgt}$ be a target function defined on an event sequence $\mathcal{E}(t_{\perp}, t_{\top})$. Let q be a query time and δt be the length of a prediction period [21]. The prediction problem is the problem where we use a predictive system to infer the value of the function $\mathbf{ptgt}(\mathcal{E}(q, q + \delta t))$ using the events occurred in the history $\mathcal{E}(0, q) = ((t_0, l_0), (t_1, l_1), (t_2, l_2), \dots, (t_{N-1}, l_{N-1}))$.

The type of a prediction problem depends on the definition of the target function $\mathbf{ptgt}(X)$. In particular, if the objective of the prediction problem is to predict the exact occurrence times and type labels for all events in the time interval $\{t' : q \leq t' < q + \delta t\}$, the target function is $\mathbf{ptgt}(X) = X$. We call this prediction problem the *ultimate prediction problem*. It is attractive to solve the ultimate prediction problem, but we do not study this problem in this thesis for two reasons.

1. It is impossible to solve the ultimate prediction problem for a random system. Event sequences are observable, but the state of a system is not observable. Real-world dynamic systems are highly stochastic. As a result, the system may produce different future scenarios even if the observable initial condition of the system stays unchanged.
2. It is unnecessary to solve the ultimate prediction problem. A practical prediction target in decision making and knowledge discovery is often a statistic or a simple fact about the future events rather than the exact occurrence times of the events. Therefore, a predictive system is useful as long as it predicts the required information accurately and efficiently.

Rather than solving the ultimate prediction problem, we focus on the *arrival prediction problem* in this thesis. *Arrival prediction* is the problem where we infer the occurrence time of the next event. At time q , we predict the occurrence time of the next event of type m in $\{t' : q \leq t' < +\infty\}$. In other words, the target function for event type m takes the form

$$\mathbf{ptgt}(X) = \min\{t : (t, m) \in X\} \quad (2.7)$$

We abstract the arrival prediction problem from many practical situations. The problem is of particular importance for the situations where a single event causes a huge impact. For instance, a patient with a heart disease may suffer sudden heart attacks. A predictive model that predicts the potential occurrence time of the next heart attack may help the patient to

take precautions. Moreover, the arrival times of events also suggest the occurrence frequency. In other words, a solution to the arrival prediction is also useful in the situation where it is necessary to allocate resources according to the number of events in the near future. For instance, a manager of a telephone call centre may be interested in predicting the number of calls. The predictions may help the manager to arrange a proper amount of staff to prevent customers from waiting while controlling the cost of human resources [22].

Like other prediction problems in statistical inference and knowledge discovery, the predictions about event streams usually contribute information to decision making. As a result, a slight difference in predictive accuracy may result in a significant profit or loss. Moreover, it is necessary to finish computation before the predictive target becomes a history, and before it is too late to take actions.

We study the predictive systems for event streams from the perspective of modelling and computing. The modelling issues include the representation of the predictive model and algorithmic routines. The computational issues include the hardware and software supporting the computation. Both perspectives involve practical challenges. In particular, we identify two challenges from the perspective of model accuracy.

- The first challenge is the non-repeatable property of stochastic systems. An ideal method to capture the statistical relationship between history and future is to reset the system to an identical condition repetitively and observe different versions of futures. For instance, we may model the changes of a stock index over time as an event stream. It is ideal that we could reverse the time to make the market activities repeatable. By reversing the time for multiple times, it is possible to observe different versions of future scenarios with an identical initial condition. In this case, we may predict the future in a statistically reasonable way. Nevertheless, the reset of a real-world stochastic system usually requires the reversion of time, which is currently impossible.
- The second challenge is the weak descriptive power of data points. A stochastic system may have unobservable and highly complicated internal structures. Therefore, an event may be the outcome of thousands of hidden factors. As a result, a long event sequence may still contain insufficient information about the mechanism which generates it. For instance, the changes of an index in the stock market is the outcome of the activities of numerous market participants. Practically, it is only possible to obtain simple observations about the market such as stock prices and trading volumes while the exact activities of the market participants are unavailable. Different activities of participants may result in the same observation at one time and different observations in the future.

On the other hand, the following two challenges hamper computational efficiency.

- The first challenge is the high time complexity of model fitting algorithms. The structure of event sequences makes it difficult to sample the data to reduce the time complexity.

Without sampling, it is impossible to design a model fitting algorithm with a lower time complexity than $O(N)$, where N is the number of events. Moreover, the time complexity of algorithms for fitting a sophisticated point process model to data is far higher than $O(N)$ or even unbounded [8].

- The second challenge is the difficulty in parallelisation. Parallelised and pipelined hardware platforms like graphics processing units (GPUs) and field programmable gate arrays (FPGAs) significantly improve the computational efficiency of various predictive systems [23]. Successful applications of hardware accelerators rely on appropriate partitioning of the task. In a traditional prediction problem with independent data points, it is often possible to divide the task by applying data operations to multiple data points in parallel. Nevertheless, the dependency between the events in an event stream makes it difficult to partition the task in a similar way.

In Chapters 3, we address the challenges using the Hawkes point process models. These models deal with the challenges of predictive accuracy by capturing the distribution of random events in the time domain. We design reconfigurable accelerators to improve the computational efficiency. In Chapter 4, we address the challenges by proposing a new predictive model for event streams. The proposed model copes with the challenges of predictive accuracy using regression techniques. The proposed model also improves the computational efficiency by eliminating intensity-based point process models and by facilitating reconfigurable computing. In Chapter 5, we address the challenges by developing accelerated analytical tools to extract information about serial correlation and statistical complexity from data. The extracted information helps people to design and tailor models, algorithms and tools for higher accuracy and efficiency.

2.2 Reconfigurable Computing

Reconfigurable computing is an approach to solve computational problems using reconfigurable hardware. The reconfigurable hardware that we use in this research is based on field-programmable gate array (FPGA) technology. In this section, we provide a short introduction to FPGA devices and the acceleration platform that we use in this thesis.

2.2.1 Reconfigurable Computing with FPGAs

The microprocessor is the major part for computation in desktop computers and servers. They provide the ability of general-purpose computing, and they are relatively easier to program. However, the price for the generality is the relatively lower computational efficiency and higher energy consumption compared with application-specific integrated circuits (ASICs)

[24]. However, ASICs are unattractive for many applications due to the long design cycles and high upfront non-recurring engineering cost associated with chip fabrication. FPGAs provide circuit-level hardware acceleration solutions like ASICs, while FPGAs do not have non-recurring engineering cost.

A typical FPGA device provides fine-grained and coarse-grained computational resources [25]. Fine-grained resources include look-up tables (LUTs), which are used in implementing logic operations. Coarse-grained resources include multipliers and on-chip memory. One may arrange and connect resources together to solve problems efficiently. In addition to computational efficiency, a major strength of FPGA implementation compared with central processing unit (CPU) or graphics processing unit (GPU) implementation is reduced power consumption. However, this thesis focuses on computational efficiency while power and energy consumption will be considered in future work.

A common design platform for FPGAs includes a chain of tools to compile a design to a configuration for the FPGA device [26]. First, the *synthesis* tool converts the design captured at the register-transfer level into a description of logic gates. Second, the *technology mapping* tool deploys the logic gates to fine-grained and coarse-grained resources on a specific FPGA chip. Finally, the *place-and-route* tool decides the physical locations of the technology-mapped components and wire the connections. The output of the place-and-route tool is a bit-stream that configures the FPGA device.

Design productivity is an important problem in FPGA design. In particular, the traditional way to program an FPGA device is to use hardware description languages, such as VHDL [27] or Verilog [28], at the circuit level or the register-transfer level. The hardware programmed in this way may exploit the computational efficiency with a full control of all the available resources. However, the detailed control of the system limits the productivity of designers.

The research on high-level synthesis addresses the productivity problem. Using a high-level synthesis tool, a designer may describe a design using high-level programming languages, such as C and C++. The high-level synthesis tool converts the code in a high-level programming language to low-level description of circuits. There are an increasing number of commercial and open-source tools for high-level synthesis. In particular, commercial tools include the Vivado HLS [29] supported by Xilinx, OpenCL tools [30] supported by both Altera and Xilinx, and OpenSPL [31] supported by Maxeler Technologies. Open-source tools, including LegUp [32] and ROCCC [32], are also gaining popularity. Review papers about high-level synthesis techniques and tools include [33] and [34].

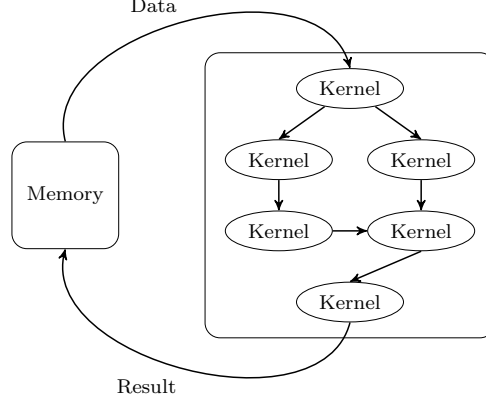


Figure 2.2: Framework of dataflow computing

2.2.2 Dataflow Computing Platform

The main reconfigurable programming paradigm studied in this research is *dataflow computing* [35]. We show the framework of dataflow computing in Figures 2.2. In the traditional software computing, a programmer implements an algorithm as a temporal sequence of operations. In contrast, in dataflow computing, the designer describes an algorithm as a spatial layout of computational kernels on a data path. During the computation, the data flows from the memory and pass through the data path, so that the kernels deployed on the data path may apply operations to the data.

In all the experiments in the thesis, we use the Maxeler MAX3 acceleration platform to design dataflow engines to accelerate the operations in event stream modelling. The hardware of the acceleration platform contains a Xilinx Virtex-6 SX475T FPGA and 48GB of dynamic random-access memory (DRAM). The acceleration platform communicates with a host computer via a PCI-Express X8 interface. The software platform includes a high-level synthesis tool, MaxCompiler, which compiles the hardware design written in the MaxJ hardware description language to a configuration for the FPGA.

The typical workflow to accelerate an application using the Maxeler MAX3 platform include the following steps:

1. Identification of efficiency bottleneck: the designer finds the computationally expensive parts of the application by time complexity analysis and profiling.
2. Design of kernels and connections: the designer describes the calculations as kernels and connections using the MaxJ language.
3. Simulation: the MaxCompiler generates simulated hardware, and the designer may

use the simulated hardware to verify the correctness of the design against software implementations.

4. Hardware compilation: the MaxCompiler converts the design described in MaxJ into VHDL and calls the Xilinx FPGA toolchain to generate a bit-stream for the FPGA.
5. Hardware optimisation: the designer may optimise the efficiency of the hardware by analysing the reports from the tools and by recompiling the hardware.

We follow the above steps to create all the hardware designs in the experiments on the MAX3 platform in this thesis. Although we use the Maxeler MAX3 platform to provides experimental results on computational efficiency and predictive accuracy, the methods proposed in this thesis can target other FPGA systems or even ASIC systems.

2.3 Intensity-Based Point Process Models

A straightforward way to build a predictive system for a prediction problem is to use intensity-based point process models. Section 2.3.1 introduces the intensity-based point processes. Section 2.3.2 discusses a family of point process models namely the self-exciting point process models. Section 2.3.3 presents a method to make predictions with point process models.

2.3.1 Point Processes

Point processes [36] are random processes describing occurrence times of random events. A univariate point process $\{t_i\}$ is a random process whose realisations are discrete time points in chronological order. A multivariate point process with M dimensions is a collection of M correlated univariate point processes. In other words, a multivariate point process takes the form

$$\mathcal{P} = \{\{t_i\}_0, \{t_i\}_1, \dots, \{t_i\}_{M-1}\} \quad (2.8)$$

Each univariate point process is a component of the multivariate point process. A realisation of an M -dimensional point process is a description of occurrence times of M types of events.

The dominating models in the study of event streams are intensity-based point process models. The idea behind intensity-based point process models is to model a statistical quantity called intensity. The intensity measures the instantaneous occurrence rate of events. Given a point process, the intensity of event type m at a time t is

$$\lambda_m(t) = \lim_{\tau \downarrow 0} E\left(\frac{C_m(t+\tau) - C_m(t)}{\tau}\right) \quad (2.9)$$

where $\lim_{\tau \downarrow 0}(u(\tau))$ takes the limit of the function value $u(\tau)$ when τ decreases to approach zero; $C_m(t)$ is the counting process of event type m which counts the number of events of type m during $\{t' : 0 < t' \leq t\}$. In other words, the *counting process* $C(t)$ is defined by

$$C(t) = \sum_{t_i \leq t} 1_{i \in \mathbb{N}} \quad (2.10)$$

The intensity function defines the occurrence patterns of random events. We may model the data by modelling the intensity function. Conventional methods to model the intensity function include the following:

1. The intensity may simply be a constant number. A point process with a constant intensity function is a *homogeneous Poisson process* [37]. The constant intensity limits the descriptive power of homogeneous Poisson process models. However, the simplicity of the intensity function makes it possible to derive fast algorithms for parameter estimation and inference in various applications. In this research, we use the homogeneous Poisson process as the baseline in the experiments about predictive accuracy in Section 4.3.
2. The intensity may depend on non-temporal information. In particular, a large number of studies build spatial-temporal point processes [38, 39] to use spatial information to model intensities. An spatial-temporal point process considers the location of the event in addition to the occurrence time and the event type. However, the objective of this research is to model the behaviours of random events in the time domain. We do not assume the availability of non-temporal information such as physical locations. Therefore, we do not discuss the intensity models in this category.
3. The intensity may also be a stochastic process itself [40, 41]. In particular, using self-exciting intensity functions is a popular method to model the intensity in a stochastic manner. Intuitively, in a point process with a self-exciting intensity function, an event in the recent history implies that more events are likely to occur in the near future. In this research, we study the self-exciting point processes from two perspectives. First, we accelerate the likelihood evaluation and parameter estimation routines in Chapter 3. Second, we use the self-exciting point processes as a start point to derive a new predictive model for event streams in Chapter 4.

In addition to the above three types of methods, it is possible to model the intensity using Gaussian processes [42], network models [43], decision trees [10] and decision forests [32].

2.3.2 Self-Exciting Point Processes

A point process $\{t_i\}$ is a self-exciting process [7] if the intensity function λ satisfies

$$\lambda(t) = \lambda^\perp + \int_0^t g(t-w) dC(w) \quad (2.11)$$

where $\lambda^\perp$ is a non-negative constant number; C is the counting process of $\{t_i\}$ and g is a response function [44] which needs to be integrable and positive [5].

Before using the point process to model the data, it is necessary to specify a response function g . We focus on the Hawkes exponential response function [7] defined by

$$g(x) = \alpha e^{-\beta x} \quad (2.12)$$

where α and β are constant parameters. With this response function, the intensity function can be written as

$$\lambda(t) = \lambda^\perp + \int_0^t \alpha e^{-\beta(t-w)} dC(w) \quad (2.13)$$

The feature that makes the Hawkes process useful for making predictions and modelling data is that the intensity (a) is stochastic, (b) changes in time and (c) depends on the history of the process.

Hawkes processe models are parametric. A parameter set for a Hawkes process with an exponential response function is a triplet in the form $\theta = (\lambda^\perp, \alpha, \beta)$. Given the parameter set and an event sequence, it is possible to compute the intensities at any time.

A multivariate point process $\{\{t_i\}_0, \{t_i\}_1, \dots, \{t_i\}_{M-1}\}$ is a multivariate self-exciting point process if the intensity function λ_m satisfies

$$\lambda_m(t) = \lambda_m^\perp + \sum_{m'=0}^{M-1} \int_0^t g_{m,m'}(t-w) dC_{m'}(w) \quad (2.14)$$

for all dimensions $m \in [0..M-1]$ where $\lambda_m^\perp$ is a constant parameter; $C_{m'}$ is the counting process of $\{t_i\}_{m'}$ and $g_{m,m'}$ is a response function which needs to be integrable and positive [5]. In this research, we focus on the multivariate Hawkes exponential response function defined by

$$g_{m,m'}(u) = \alpha_{m,m'} e^{-\beta_{m,m'} u} \quad (2.15)$$

where $\alpha_{m,m'}$ and $\beta_{m,m'}$ for all $(m, m') \in [0..M-1]^2$ are constant parameters. In other words, the intensity of an M dimensional self-exciting point process with the multivariate Hawkes

exponential response function is in the form

$$\lambda_m(t) = \lambda_m^\perp + \sum_{m'=0}^{M-1} \int_0^t \alpha_{m,m'} e^{-\beta_{m,m'}(t-w)} dC_{m'}(w) \quad (2.16)$$

The property of such a point process can be uniquely given by a parameter set

$$\theta = (\lambda^\perp, A, B) \quad (2.17)$$

where

$$\lambda^\perp = \left(\lambda_0^\perp \quad \lambda_1^\perp \quad \cdots \quad \lambda_{M-1}^\perp \right)^T \quad (2.18)$$

$$A = \begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,M-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,M-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{M-1,0} & \alpha_{M-1,1} & \cdots & \alpha_{M-1,M-1} \end{pmatrix} \quad (2.19)$$

$$B = \begin{pmatrix} \beta_{0,0} & \beta_{0,1} & \cdots & \beta_{0,M-1} \\ \beta_{1,0} & \beta_{1,1} & \cdots & \beta_{1,M-1} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{M-1,0} & \beta_{M-1,1} & \cdots & \beta_{M-1,M-1} \end{pmatrix} \quad (2.20)$$

where the parameter matrix B may reduce to an M dimensional vector if we assume that all rows are identical [5]. For instance, Figure 2.3 shows an event sequence produced by a two-dimensional Hawkes process.

2.3.3 Prediction with Point Process Models

Point process models provide a way to build predictive systems for event streams. As we focus on parametric point process models in this research, it is necessary to estimate the parameters of the point process model using data before making predictions. A well-studied approach for parameter estimation is maximum likelihood estimation (MLE) [45, 46, 47]. The core calculation of MLE is the evaluation of the following estimator:

$$\theta_{\text{MLE}} = \operatorname{argmax}_\theta L(\theta) \quad (2.21)$$

where $\operatorname{argmax}_x(f(x))$ is the argument x that maximises the function value $f(x)$; L is the likelihood function of the point process.

The computational efficiency of parameter estimation depends on the form of the likelihood function. In an intensity-based point process model, the form of the likelihood function

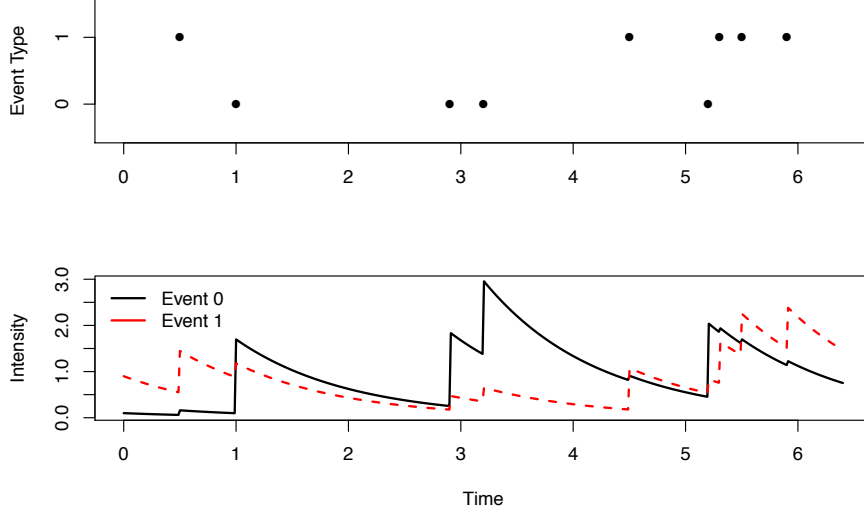


Figure 2.3: A two dimensional Hawkes process

depends on the definition of the intensity function. In particular, if the estimator for a point process model has a closed-form expression, one may compute θ_{MLE} directly by evaluating the expression. Otherwise, it is necessary to search in the parameter space to find the parameter set that maximises the likelihood function.

The predictive accuracy of a homogeneous Poisson process model may be limited because it assumes a constant intensity for each type of event. Nevertheless, the algorithms for parameter estimation and prediction are highly efficient. The parameter estimation is computable in $O(1)$ time if the numbers of events for each event type are available, and the count of events takes at most $O(N)$ time where N is the number of events in the data set. Therefore, the parameter estimation is computable in at most $O(N)$ time.

Hawkes processes models are mathematically more general than homogeneous Poisson process models. The generality makes the Hawkes process models more accurate than homogeneous Poisson process models in data interpretation [48] and prediction [5, 49] for some practical problems. However, the complexity of the Hawkes process models makes it impossible to compute the maximum likelihood estimator in polynomial time. In particular, it is necessary to search in the parameter space for a parameter set that maximises the likelihood. To the best of our knowledge, there are no theoretical upper bounds for the execution time for any search algorithm with the likelihood function. As a result, the MLE for Hawkes process models is usually time-consuming [8].

One may use a general-purpose optimisation algorithm to search for a parameter set. Common methods for the MLE for Hawkes process models include the Nelder–Mead (NM) downhill simplex algorithm [55, 56], the expectation-maximisation (EM) algorithm [50], the

Table 2.1: Parameter estimation methods for Hawkes process models

Reference	Field of Study	M	Optimisation Tech.
Olson <i>et al.</i> [50]	General-purpose	1	EM
Da Fonseca <i>et al.</i> [51]	Trading analytics	1	GMM
Brentnall <i>et al.</i> [20]	ATM management	1	NM
Vinkovskaya [5]	Limited order books	4	NM
Halpin <i>et al.</i> [52]	Email interactions	2	NM
Liniger <i>et al.</i> [53]	Asset returns	1	NM, EM, DEA
Zadeh <i>et al.</i> [54]	Online social networks	2	NM, SA
Kobayashi <i>et al.</i> [19]	Online social networks	1	NM

generalised method of moments (GMM) [57], the differential evolution algorithm (DEA) [58] and the simulated annealing algorithm (SA) [59]. Table 2.1 summaries the optimisation algorithms used in a collection of studies.

An optimisation algorithm for maximum likelihood evaluation needs to evaluate log-likelihood frequently during the optimisation process. The evaluation process becomes computationally expensive when the number of data points is large. Practically, it is necessary to use large data sets to provide satisfactory estimation accuracy. As a result, the evaluation of the likelihood function usually dominates the execution time of the parameter search [51, 53, 60].

Reconfigurable computing is a promising method to accelerate the MLE operation for Hawkes process models. There have been studies applying pipelined dataflow engines to accelerate MLE for other statistical models. For instance, Guo *et al.* [61] design an expectation-maximisation engine to accelerate the MLE for Gaussian mixture models. However, the likelihood functions for different statistical models are substantially different. As a result, it is impossible to directly apply the acceleration techniques in existing work to our problem.

The acceleration of MLE for Hawkes point process models requires efficient calculation of the likelihood function. However, it is challenging to design hardware to accelerate the calculation due to the complex data dependency in the likelihood function. In this thesis, we develop mathematical methods to remove the complex data dependency and design reconfigurable accelerators to improve the efficiency of MLE for Hawkes process models in Chapter 3. As it is impossible to avoid the parameter search, the MLE process may still be inefficient even if we use hardware accelerators. We address the efficiency problem by proposing a novel predictive model in Chapter 4 which works without point process models.

2.4 Regression

Regression [62, 63] is a topic in statistical inference and supervised machine learning. A regression model maps an observation in the form of a vector to a numeric label. One may use a regression model to make predictions after training it with data. It is usually necessary to use sophisticated model representations to secure satisfactory predictive accuracy. Sophisticated models require training algorithms with high time complexity. Moreover, an effective way to improve the predictive accuracy is to increase the amount of data rather than designing smarter algorithms [64]. Therefore, the training process of regression models may become computationally expensive for practical problems.

In this section, we briefly introduce the gradient descent method for training linear regression models. We then review existing reconfigurable accelerators for training regression models.

2.4.1 Gradient Descent for Linear Regression

The goal of training a regression model is to find a function ϕ that maps a D -dimensional vector $\mathbf{x}_p = (x_{p,0}, x_{p,1} \dots, x_{p,D-1})$ to a numeric label y_p using a training data set. A training data set is a collection of data points $\mathcal{O} = \{o_0, o_1 \dots o_{N-1}\}$. Each data point is in the form $o_i = (\mathbf{x}_i, y_i)$ where $\mathbf{x}_i$ is a feature vector; y_i is the expected label for $\mathbf{x}_i$. In this study, we address the regression problem where all the feature vectors are real vectors and all labels are real numbers.

We focus on linear regression models in this research. In other words, the regression model produces a prediction $\phi_{\mathbf{w}}(\mathbf{x}_p)$ for the input $\mathbf{x}_p$ by

$$\phi_{\mathbf{w}}(\mathbf{x}_p) = \mathbf{x}_p \cdot \mathbf{w} = \sum_{d=0}^{D-1} w_d x_{p,d} \quad (2.22)$$

where the parameter $\mathbf{w} = (w_0, w_1, \dots, w_{D-1})$ is a D -dimensional vector of linear combination weights. Practically, it is possible to enhance the predictive power by applying non-linear basis expansion to the input domain.

The vector $\mathbf{w}$ uniquely determines the regression function with respect to the training data. Therefore, the training process of the model is to find out the best parameter $\mathbf{w}$. Before training the model, it is necessary to find a quantity to tell good parameters from bad ones. A conventional way to measure the goodness-of-fit for a parameter $\mathbf{w}$ is to define a loss function. In this study, we use the squared error on the training data as the loss function. In particular, the loss function on a single data point $o_i = (\mathbf{x}_i, y_i)$ for the regression model is

$$L(\mathbf{w}, \mathbf{x}_i, y_i) = \frac{1}{2} (\phi_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2 \quad (2.23)$$

The overall loss function on the whole data set is

$$L(\mathbf{w}) = \sum_{i=0}^{N-1} L(\mathbf{w}, \mathbf{x}_i, y_i) \quad (2.24)$$

The training algorithm of the regression model is the optimisation process of the model with respect to parameter $\mathbf{w}$. In particular, the training process of a linear regression model is the procedure to find the parameter $\mathbf{w}$ that minimises the overall loss function in Equation 2.24. Practically, we may use iterative optimisation algorithms, such as gradient descent algorithms, to train regression models when the data volume is large [65].

Gradient decent algorithms [66] are optimisation techniques that minimises error functions under the guidance of the gradient information. In particular, there exist two typical methods, namely *batch gradient descent* [67, 68] and *stochastic gradient descent* [69, 70]. The batch gradient descent algorithm directly uses the gradients of the overall error function on the whole data set in Equation 2.24 to guide the minimisation process. In contrast, the stochastic gradient descent algorithm uses the gradient information of the error function for individual data points in Equation 2.23. In other words, it interactively updates the parameters to minimise the overall loss function in Equation 2.24 in an indirect manner.

The stochastic gradient descent algorithm starts the minimisation process with an initial guess for the parameter $\mathbf{w}$ and update the parameter iteratively with data using the update rule

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla \mathbf{w} \quad (2.25)$$

where α is a positive real number indicating the learning rate that may change over time [71]; $\nabla \mathbf{w} = (\nabla w_d \dots \nabla w_{D-1})$ is the gradient of the loss function in Equation 2.23 for a data point with respect to $\mathbf{w}$. In the linear regression model, each entry in $\nabla \mathbf{w}$ takes the form

$$\nabla w_d = \partial_{w_d} (L_i(\mathbf{w}, \mathbf{x}_i, y_i)) = \left(\sum_{d=0}^{D-1} w_d x_{i,d} - y_i \right) x_{i,d} \quad (2.26)$$

Algorithm 1 shows the computational routine of the stochastic gradient descent method. Discussions on the correctness and complexity of the method are available in [63].

2.4.2 Reconfigurable Training Engines for Regression

There have been a number of studies on the acceleration of supervised learning methods. We briefly review existing acceleration solutions for training regression models. In particular, we focus on two well-explored types of regression models namely neural networks and support vector machines in the rest of this subsection. As we focus on the regression model, we do

Algorithm 1: Stochastic gradient descent for linear regression

Input: Training data set $\mathcal{O}$

Output: Parameter $\mathbf{w}$

```
1  $\mathbf{w} \leftarrow \text{make-initial-guess}()$ 
2 while termination condition not met do
3    $(\mathbf{x}_i, y_i) \leftarrow \text{load-data-point}(\mathcal{O})$ 
4    $\hat{y}_i \leftarrow \phi_{\mathbf{w}}(\mathbf{x}_i)$ 
5    $\delta_i \leftarrow \hat{y}_i - y_i$ 
6   for  $d \in [0..(D-1)]$  do
7      $\nabla w_d \leftarrow \delta_i x_{i,d}$ 
8      $w_d \leftarrow w_d - \alpha \nabla w_d$ 
9   end
10 end
```

not cover the hardware solutions for unsupervised learning methods such as frequent pattern discovery [72], clustering [73] and probability density modelling [61]. As we focus on the training process, we do not cover the hardware engines that only accelerate the prediction process of regression models. Reviews that cover prediction engines include [74, 75, 76].

Training Neural Networks

Among the statistical models that are capable for regression, neural networks are arguably the most popular candidate for reconfigurable acceleration [77]. There are three types reconfigurable acceleration solutions for neural networks.

The first type of solutions accelerate the training process of general-purpose neural network architectures. The studies in this category use existing neural network structures and achieve high efficiency from the perspective of model optimisation. The dominating method to train neural networks is the back-propagation algorithm [78]. Eldredge and Hutchings [79] divide the back-propagation algorithm into three stages and design hardware for each stage separately. During the training process, the hardware performs a run-time reconfiguration at the beginning of each stage. Paul and Rajopadhye [80] propose a systolic back-propagation engine that avoids the run-time reconfiguration. Their design maps all calculations in a complete back-propagation procedure to hardware. Maeda and Tada [81] propose a training engine for neural networks using the simultaneous perturbation rule [82] to avoid the gradient computation in the training process.

The second type of solutions are hardware-oriented neural network structures. A popular network structure in this category is the *block-based neural network*. Moon and Kong [83] first propose the block-based neural network (BbNN) and implement the prediction facility on the FPGA platform. The structure of the network is different from a conventional feed-forward neural network. Specifically, a block-based neural network connects a collection of neuron

blocks. A neutron block carries four numeric I/O ports. Each I/O port may serve as either an input port or an output port. An output port provides an activation signal computed from the input signals on the same neutron block. Jiang *et al.* [84, 85, 86] study the training process of the BbNN using evolutionary algorithms on the CPU platform. The idea behind their training approach is to encode the topology of the network and the configuration of the neural blocks into a vector so that an evolutionary algorithm may improve the network by manipulating the vector. Merchant and Peterson [77] make it possible to train the block-based neural networks using FPGAs.

The third type of solutions are domain-specific or problem-specific neural networks. A representative neural network structure in this category is the *convolutional neural network*. The convolutional neural network (CNN) [87] is a feed-forward neural network structure inspired by the visual cortex of animals. The major application of CNNs is image recognition [87]. The reconfigurable acceleration solutions for CNNs usually take advantages of the unique properties of image data. Farabet *et al.* proposes an FPGA-based RISC processor that maps basic operations in the CNN. The processor takes a description of a pre-trained CNN in the form of a sequence of instructions to make predictions. A useful observation in this study is that a proper reduction of the precision of image operations only results in a little negative impact on the predictive accuracy. However, the precision reduction may save a considerable amount of hardware resources. Further optimisations [88, 89, 90] have made FPGA devices faster and more energy-efficient than CPUs and GPUs for inference. Zhao *et al.* [91] propose the first training engine for CNNs using a streaming data path. The data path contains a collection of parameterised modules. The organisation of these modules changes over time with run-time reconfiguration, which enables the data path to train different layers of a network.

Training Support Vector Machines

Support vector machines (SVMs) [92, 93] are supervised learning models originally designed for binary classification problems. A variant of the original SVM model, support vector regression (SVR) [94], solves regression problems. The majority of existing research on the acceleration of SVM models focus on the original SVM for classification. However, as the optimisation methods behind the original SVM and the SVR are similar, it is possible to apply the acceleration techniques for the original SVM to SVR with slight modifications. We divide the studies about the acceleration of SVM training into two categories.

The first type of methods evaluate the kernel function of SVMs using a reconfigurable accelerator and leave the optimisation framework to the CPU platform. Cadambi *et al.* [95] discovers that dot product operations in the computation of kernel function may consume 90%-95% of the total execution time. They design a co-processor to compute the dot product. An

optimised implementation of the proposed architecture on a Virtex-5 LX330T FPGA achieves 18 times speed up in training compared to a dual-core AMD Opteron CPU running at 2.2GHz.

The second type of methods map the whole training process to a reconfigurable engine. Anguita *et al.* [96] propose a training algorithm for SVMs namely *Flex b Svm* for the ease of hardware mapping. This algorithm solves the optimisation problem in an iterative way and allows efficient mapping to reconfigurable hardware. An optimised implementation of the architecture on a Xilinx Virtex-II XC2V8000 FPGA achieves a predictive accuracy of around 95% in two practical problems. An improvement in on-chip SVM training is the the *1-variable sequential minimal optimisation (SMO) algorithm* proposed in [97]. The 1-variable SMO algorithm trains SVM models with integer parameters using an optimisation method similar to the standard SMO [98]. The use of integer parameters enables the architecture to target reconfigurable hardware with limited resources for floating-point arithmetic. In their experiments with 13 benchmarks and a people-tracking application, the SVM models trained by the proposed architecture incur similar error compared with conventional SVM systems. However, as the study focuses on the consumption of hardware resources, the computational efficiency was not compared to other architectures such as CPUs or GPUs.

2.5 Time Series Analysis

Time series analysis [99] is a subject closely related to event streams. In this section, we provide a brief introduction to time series analysis and explain two computational problems namely correlation measurement and ordinal analysis.

2.5.1 Time Series

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

$$y = (y_0, y_1, y_2, \dots, y_{N-1}) \quad (2.27)$$

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

$$y_i = (y_{i,0} \ y_{i,1} \ y_{i,2} \ \dots \ y_{i,D-1})^T \quad (2.28)$$

where $y_{i,0} \dots y_{i,D-1}$ are *components* of the data point y_i . In particular, a univariate time series is a special case of the multivariate time series where each data point contains a single component.

There is a close relationship between time series and event streams. Also, time series

analysis provides indirect approaches to analyse event streams. A widely-used technique is to convert an event sequence to a time series and analyse the converted time series. For instance, Smith *et al.* [100] propose a state-space method to model univariate event sequences. This model divides the timeline into intervals with equal widths so that each interval contains at most one event. Yu *et al.* [101] also divide the timeline into equally-sized intervals, but the model in this work allows multiple events to occur in one interval. The model counts the number of events in each interval to form time series data.

A practical problem of time series analysis is the lack of fast computing systems to calculate statistics from data. In recent years, data analysts began to appreciate the value of microscopic statistical patterns in large time series data. Therefore, the efficiency problem arise. Moreover, the value of data decays with time. It is important to extract useful information as fast as possible before the data lose its value. As a result, it is desirable to design faster computing facilities for time series analysis.

Hardware acceleration for time series data processing has not been well-studied. Existing work mainly focuses on pattern matching and aligning problems for time series. For instance, Sart *et al.* [102] propose acceleration solutions to speed up dynamic time wrapping (DTW) for time series using GPUs and FPGAs. Wang *et al.* [103] develop an FPGA-based hardware engine for DTW-based sequence searching in time series. 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.

2.5.2 Measure of Correlation

An important problem in time series analysis is the measure of correlation between components. Assume that we have a random process that satisfies

$$\mathbb{E}[y_i] = \mu \quad (2.29)$$

$$\mathbb{E}[(y_i - \mu)(y_{i-h} - \mu)^T] = \Omega_h \quad (2.30)$$

where μ is the expectation vector of the time series; Ω_h is the autocovariance matrix of lag h . Let y be a time series of length N sampled from this random process. One may estimate μ by taking the sample mean over N time steps

$$\hat{\mu} = \bar{y} = \frac{1}{N} \sum_{i=0}^{N-1} y_i \quad (2.31)$$

In addition to the mean vector, it is also useful to know the correlation between components. However, measuring such a correlation is non-trivial for time series because a data point may depend on historical states. As a consequence, commonly used correlation measures for

non-temporal data, like the ordinary covariance matrix, may become invalid [99].

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

$$S = \sum_{h=-\infty}^{\infty} \Omega_h \quad (2.32)$$

Unfortunately, it is impossible to compute the matrix S using Equation 2.32 because the length of a time series must be finite.

Heteroskedasticity and autocorrelation consistent (HAC) estimation is a technique that approximates S using a finite-length time series. This estimation can be achieved by computing the *Newey-West estimator* [104, 105] defined by

$$\hat{S} = \hat{\Omega}_0 + \sum_{h=1}^H \kappa\left(\frac{h}{H+1}\right)(\hat{\Omega}_h + \hat{\Omega}_h^\top) \quad (2.33)$$

where

- H is a user-defined lag truncation parameter.
- κ is a real-valued kernel function. In this research, we study the HAC estimation with pre-computed kernel function values. Discussions about kernel functions are available in [104] and [106].
- $\hat{\Omega}_h$ is the estimate of the autocovariance matrix with lag h defined by

$$\hat{\Omega}_h = \frac{1}{N} \sum_{i=h}^{N-1} (y_i - \hat{\mu})(y_{i-h} - \hat{\mu})^\top \quad (2.34)$$

The estimate of long-run covariance is a weighted sum of a collection of estimated autocovariance matrices where the kernel function controls the weights. As the time to compute the estimator in Equation 2.34 grows with the size of the data, HAC estimation for long and multivariate time series data is computationally demanding when the data volume is large. We aim to improve the computational efficiency of HAC estimation by designing reconfigurable accelerators. However, it is challenging to design such an accelerator, because the calculation of autocovariance matrices may incur high bandwidth consumption.

2.5.3 Ordinal Analysis

In addition to the measure of correlation, another important technique for time series analysis is to quantify the complexity of a time series. A complexity measure reflects the regularity

and predictability of the time series. Ordinal analysis [107] is an approach to analyse the complexity of time series by considering the order of small segments in the time series. This approach does not depend on a particular time series model, and it is robust against various types of noise. It has been used in characterising dynamic changes in time series, with various applications such as financial risk modelling and biomedical signal processing.

Given a sequence of n distinct values $b = (b_0, \dots, b_{n-1})$, the ordinal pattern of b is mathematically described by a permutation $\pi = (k_0, \dots, k_{n-1})$, such that $b' = (b_{k_0}, \dots, b_{k_{n-1}})$ is in ascending order. For instance, the sequences $b = (b_0, b_1, b_2) = (5, 2, 4)$ satisfies $b_1 < b_2 < b_0$. Therefore, the ordinal pattern of this sequence is $(1, 2, 0)$. A univariate time series $x = (x_0, \dots, x_{N-1})$ and an integer $n \in [1..N]$ contains $(N - n + 1)$ contiguous sub-sequences with length n . Ordinal patterns of these sub-sequences contain rich information about chaotic properties of the time series [108].

Ordinal analysis refers to the study of the distribution of these ordinal patterns. The first step for ordinal analysis is to extract all ordinal patterns in the time series and encode them in a predefined format. The length of sub-sequences we consider is the query order of the encoding problem. Ordinal pattern encoding is time-consuming for long time series or high query orders.

The encoding procedure is computationally intensive regarding arithmetic operations, but it only requires sequential access to the time series data. We consider it beneficial to encode ordinal patterns with a pipelined accelerator using reconfigurable hardware. The abundant logical resources in modern reconfigurable devices enable the arithmetic operations to be efficiently parallelised and pipelined. However, encoding ordinal patterns using reconfigurable hardware is challenging, as the straightforward encoding method requires sorting, which lacks efficient hardware implementation.

2.6 Summary

In this chapter, we review five topics related to this thesis, including predictive modelling for event streams, reconfigurable computing, point process models, regression analysis and time series analysis.

We mention that the conventional method for event stream modelling is to use point process models. The calculations for model fitting involve complex data dependency. As a result, it is challenging to design dataflow engines for acceleration. In Chapter 3 of the thesis, we address the challenges by deriving new mathematical methods for the calculations to simplify the data dependency. In Chapter 4 of the thesis, we address the challenges by proposing a novel predictive model for event streams which avoids the complex data dependency in point process models.

We mention that the techniques for time series analysis require high bandwidth consump-

tion, and they involve operations that lack efficient hardware implementations such as sorting. In Chapter 5, we address the challenges by designing new algorithms to promote data reuse, and by developing time series encoding schemes to avoid sorting.

Chapter 3

Reconfigurable Accelerators for Hawkes Process Models

This chapter covers reconfigurable acceleration solutions for three operations for Hawkes point process models namely intensity evaluation, likelihood evaluation and parameter estimation. Intensity evaluation is the operation to calculate the intensities for a time point t using a data set $\mathcal{E}(0, T)$ and a parameter set θ . Likelihood evaluation is the operation to evaluate the data likelihood using a parameter set θ and an event sequence $\mathcal{E}(0, T)$. Parameter estimation is the operation to find a parameter set θ that best fits a event sequence $\mathcal{E}(0, T)$. The main novel aspects presented in this chapter include the following:

- An intensity evaluation engine for the Hawkes point process models. (Section 3.1)
- Novel mathematical expressions of the likelihood functions that simplify data dependency for dataflow computing. (Section 3.2.1 and 3.3.4)
- Reconfigurable accelerators for likelihood evaluation based on the proposed mathematical expressions. (Section 3.2.2 and 3.3.5)
- An empirical study where we apply the likelihood evaluation accelerator to improve the efficiency of parameter estimation. (Section 3.4)

The rest of this chapter is organised as follows. Section 3.1 presents an approach for accelerating intensity evaluation for multivariate Hawkes point processes. Section 3.2 and 3.3 respectively discuss the likelihood evaluation for univariate and multivariate Hawkes process models. Section 3.4 presents an efficient collaboration scheme between the host computer and the accelerator for parameter estimation. Section 3.5 gives a brief summary of this chapter.

3.1 Intensity Evaluation

Intensity evaluation for a Hawkes process model is the evaluation of the intensities at a given time t for all dimensions, $\lambda_0(t), \lambda_1(t), \lambda_2(t), \dots, \lambda_{M-1}(t)$, using a parameter set $\theta = (A, B, \Lambda)$. The major challenge for reconfigurable acceleration is the bottleneck of memory bandwidth. We address the challenge by proposing an intensity evaluation strategy shown in Equation 3.4, 3.5 and 3.6. With the proposed strategy, we may save the memory bandwidth by calculating multiple intensities in parallel with a single data pass.

In general, the intensity of the Hawkes process is a function of the whole history of the event stream. In real-time applications, such as the modelling and prediction of high-frequency financial data involving thousands of data points, the data dependency makes the repeated evaluation of the intensity computationally challenging.

As we have discussed in Section 2.1, we focus on the Hawkes exponential response function this research. This response function has a Markov property allowing the computation of intensities with sequential updates. In our discussions about intensity evaluation, we do not take advantage of this property so that the methods proposed in this section can be adapted to work with other response functions regardless whether the function has the Markov property. We take advantage of the Markov property when we study the likelihood evaluation operation in Section 3.2 and 3.3.

3.1.1 Strategy of Acceleration

In this section, we first analyse the data dependency in the intensity evaluation problem. We then develop a computing strategy to calculate multiple intensities in parallel with a single data pass.

A computable form of the intensity is as follows:

$$\begin{aligned}\lambda_m(t) &= \lambda_m^\perp + \sum_{m'=0}^{M-1} \sum_{k=0}^{K_m(t)-1} g_{m,m'}(t - t_{m',k}) \\ &= \lambda_m^\perp + \sum_{m'=0}^{M-1} \sum_{k=0}^{K_m(t)-1} \alpha_{m,m'} e^{-\beta_{m,m'}(t-t')} \end{aligned} \quad (3.1)$$

where $K_m(t)$ is the number of events with type label m before t . It is straightforward to compute the intensities $\lambda_0(t), \lambda_1(t), \lambda_2(t), \dots, \lambda_{M-1}(t)$ individually using Equation 3.1. However, we consider such a strategy inefficient from the perspective of reconfigurable computing.

We investigate this efficiency problem by analysing the data dependency which is shown in Figure 3.1. In this figure, an edge from a node x to node y describes the fact that x is a necessary element to compute y . If y is computable using a single access to x , then the label of the edge is '1'. Otherwise the label is an asterisk. The data dependency suggests that if

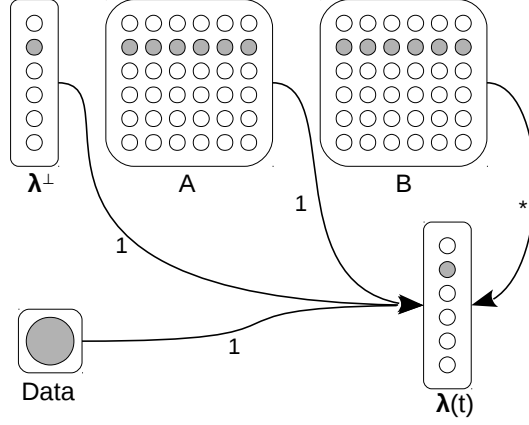


Figure 3.1: Data dependency of straightforward intensity evaluation

we directly evaluate the intensities following the definition, we may only obtain one intensive value after accessing the whole data set. As a result, it is unavoidable to access the whole data set for M times to compute all the intensities for all the M event types.

In this study, we aim to design a hardware architecture to accelerate intensity evaluation. The bandwidth between the memory and the acceleration platform is limited. Therefore, it is inefficient in terms of memory bandwidth consumption if we obtain only one intensity with a data pass.

3.1.2 One-Pass Computation of Multiple Intensities

Regarding the problem on the memory bandwidth, we propose a method to calculate multiple intensities in parallel with a single data pass. Let $\lambda_{m_0}(t), \lambda_{m_1}(t), \dots, \lambda_{m_{c-1}}(t)$ be a group of c intensities at time t . By Equation 3.1,

$$\begin{pmatrix} \lambda_{m_0}(t) \\ \lambda_{m_1}(t) \\ \vdots \\ \lambda_{m_{c-1}}(t) \end{pmatrix} = \begin{pmatrix} \lambda_{m_0}^\perp \\ \lambda_{m_1}^\perp \\ \vdots \\ \lambda_{m_{c-1}}^\perp \end{pmatrix} + \mathbf{s}(t) \quad (3.2)$$

where

$$\mathbf{s}(t) = \sum_{m'=0}^{M-1} \begin{pmatrix} \sum_{k=0}^{K_m(t)-1} \alpha_{m_0,m'} e^{-\beta_{m_0,m'} \cdot (t-t_{m',k})} \\ \sum_{k=0}^{K_m(t)-1} \alpha_{m_1,m'} e^{-\beta_{m_1,m'} \cdot (t-t_{m',k})} \\ \vdots \\ \sum_{k=0}^{K_m(t)-1} \alpha_{m_{c-1},m'} e^{-\beta_{m_{c-1},m'} \cdot (t-t_{m',k})} \end{pmatrix} \quad (3.3)$$

$$= \sum_{m'=0}^{M-1} U_{m'} \mathbf{v}_{m'}(t) \quad (3.4)$$

where $U_{m'}$ is a $c \times c$ diagonal matrix defined by

$$U_{m'} = \begin{pmatrix} \alpha_{m_0,m'} & 0 & 0 & 0 \\ 0 & \alpha_{m_1,m'} & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \alpha_{m_{c-1},m'} \end{pmatrix} \quad (3.5)$$

and $\mathbf{v}_{m'}(t)$ is a c -dimensional column vector defined by

$$\mathbf{v}_{m'}(t) = \sum_{k=0}^{K_m(t)-1} \begin{pmatrix} e^{-\beta_{m_0,m'} \cdot (t-t_{m',k})} \\ e^{-\beta_{m_1,m'} \cdot (t-t_{m',k})} \\ \vdots \\ e^{-\beta_{m_{c-1},m'} \cdot (t-t_{m',k})} \end{pmatrix} \quad (3.6)$$

The diagonal matrix $U_{m'}$ is independent of the data. All the elements needed to construct $U_{m'}$ are available in the m' -th column of the model parameter A in Equation 2.19. The vector $\mathbf{v}_{m'}(t)$ is the crucial part of the computation. Once $\mathbf{v}_{m'}(t)$ for all $m' \in [0..(M-1)]$ are available, $\lambda_{m_0}(t)$, $\lambda_{m_1}(t)$, $\lambda_{m_2}(t)$, $\dots$, $\lambda_{m_{c-1}}(t)$ are computable by Equation 3.2, 3.4, 3.5 and 3.6. However, the vector $\mathbf{v}_{m'}(t)$ depends on all past events. Therefore, the computation of $\mathbf{v}_{m'}(t)$ becomes the efficiency bottleneck when the number of events in the data set is large.

The data dependency underlying our proposed method shown in Figure 3.2 is different from that of the straightforward method shown in Figure 3.1. In particular, the proposed method only needs to access the data set once to compute c intensities. We consider this property valuable for developing hardware acceleration engines. In reconfigurable computing systems, it is usually the memory access time, rather than the logical resources, that limits the computational efficiency and scalability of the hardware. Obtaining multiple results with one data pass improves the memory efficiency.

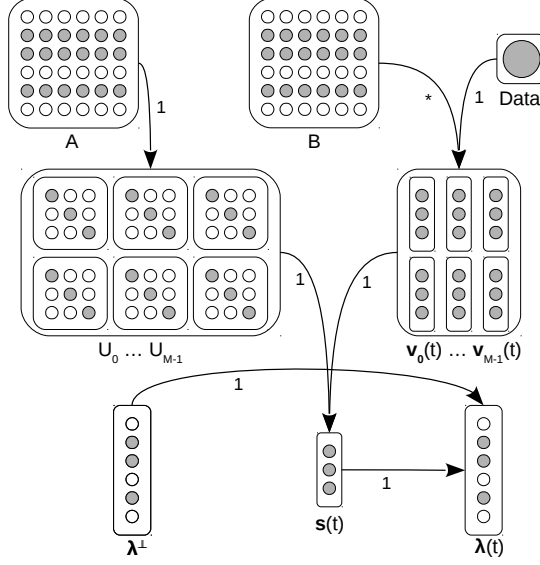


Figure 3.2: Data dependence pattern of proposed evaluation strategy

3.1.3 Hardware Design

The calculation of $\mathbf{v}_{m'}(t)$ is the most computationally expensive procedure in intensity evaluation. We accelerate this calculation by designing and optimising a pipelined hardware architecture. We first propose a basic architecture and explain its core component, the *glove*. We then propose a method to combine multiple gloves to promote data parallelism.

We propose a hardware architecture, which is shown in Figure 3.3, to compute $\mathbf{v}_{m'}(t)$ following Equation 3.6. The major computational part of the architecture, which we call a *glove*, is enclosed in the large rounded square in the figure. For ease of discussion, we refer to a component in charge of accumulating the statistical information for a single event type as a *finger*. It is necessary to deploy c fingers in a glove to compute c intensities in a single data pass.

The output of the i -th finger is accumulated using an accumulator. When an event in the m' -th dimension passes through the engine, the engine calculates and accumulates the contribution of the event towards all the c components in $\mathbf{v}_{m'}(t)$ in parallel. When all the data instances in the m' -th dimension have been streamed into the engine, the entries in vector $\mathbf{v}_{m'}(t)$ can be harvested from the accumulators. When the whole data set has passed through the engine, we may obtain $\mathbf{v}_{m'}(t)$ for all $m' \in [0..(M-1)]$. The host computer then calculates $\lambda_{m_0}(t)$, $\lambda_{m_2}(t)$, $\dots$, $\lambda_{m_{c-1}}(t)$ using Equation 3.2, 3.4, 3.5 and 3.6.

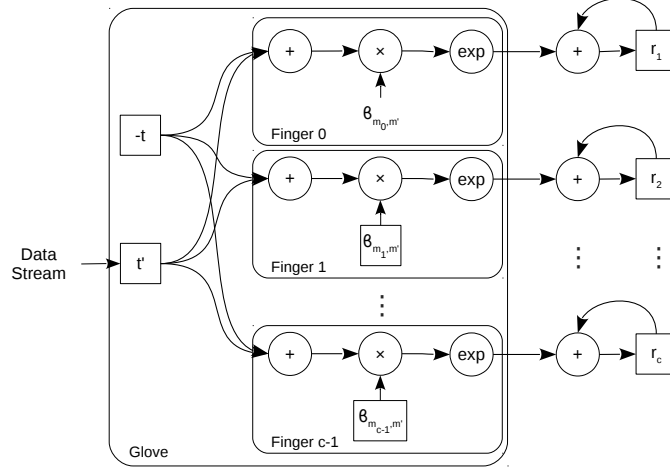


Figure 3.3: Basic architecture with a single glove

3.1.4 Data Parallelism

The single-glove architecture reaches its maximum computational efficiency when the number of fingers c is a factor of the number of dimensions M . In this situation, all the M intensities can be computed with exactly M/c data passes. In contrast, if c is not a factor of M , we need $\lceil M/c \rceil$ data passes to finish the computation of all intensities. In some data passes, certain fingers of the glove are inactive. Practically, we may maximise c with respect to the limitation of hardware resources on a particular platform to maximise computational efficiency. However, we also expect to reduce c to avoid or reduce the idleness of hardware resources.

The memory bandwidth requirement of the single-glove architecture is both small and constant. Moreover, there is no particular computational order in the summation operation in Equation 3.6. As a result, we may take advantage of data parallelism to maximise resource usage without increasing the number of fingers c .

We propose to deploy a group of g gloves in the way shown in Figure 3.4 to exploit the data parallelism. Specifically, we break the data stream into g segments with identical or similar length and stream them into g gloves in parallel.

3.1.5 Experimental Evaluation

The general information of the test data is listed in Table 3.1. For each problem, we generate test data sets with different sizes L from 10^4 to $10^{9.5}$ using the thinning procedure [109]. We produce the data to simulate the situations in [5, 2, 9, 6]. The distribution of the simulated data may be different from the real-world data. However, our results are unaffected as the execution time only depends on the size of data.

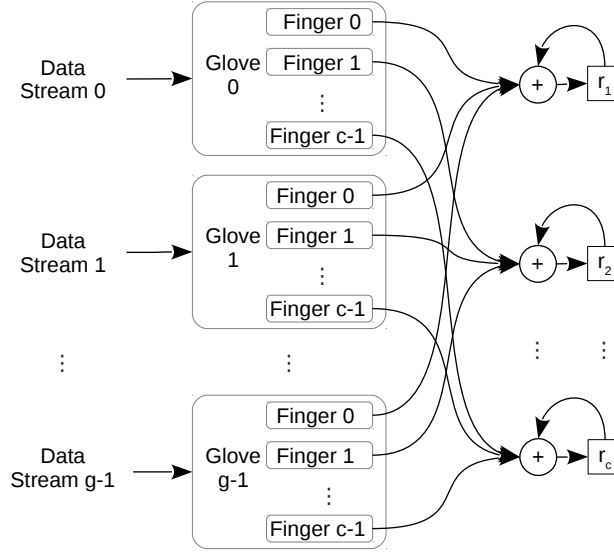


Figure 3.4: Exploiting data parallelism by combining g gloves

Table 3.1: Test data for intensity evaluation

Case	Area of study	No. of dimensions M
1	Order book reconstruction (1 stock) [5]	4
2	Earthquake analysis (10 geological areas) [2]	10
3	Stock trading behaviour modelling (10 stocks) [9]	20
4	Neurone activity modelling (32 neurones) [6]	32

A data set is loaded into the main memory of the host computer before the computation begins. The data are then transmitted to the acceleration system via the PCI-Express interface in real time during the computation. The computational efficiency is measured by the total execution time for all the M intensities. The execution time also includes the data transmission overhead from the memory to the acceleration platform.

Experimental results are shown in Figure 3.5. Each column of the figures corresponds to a data set in Table 3.1. The first two rows of figures record the computational efficiency of the FPGA and CPU implementations respectively. We record the data size in log scale with base 10 in these figures. We also record the execution time in log scale to show the trends. A small difference along the vertical axis in each figure means a huge difference in execution time as we use the log space. The third row of the figures shows the speedup of the FPGA implementation over one core and eight cores. We do not take the logarithms of these numbers.

The CPU implementation works well on eight cores. It is around 7 times faster than a single core in all tested cases. The speedup of the FPGA implementation over the CPU imple-

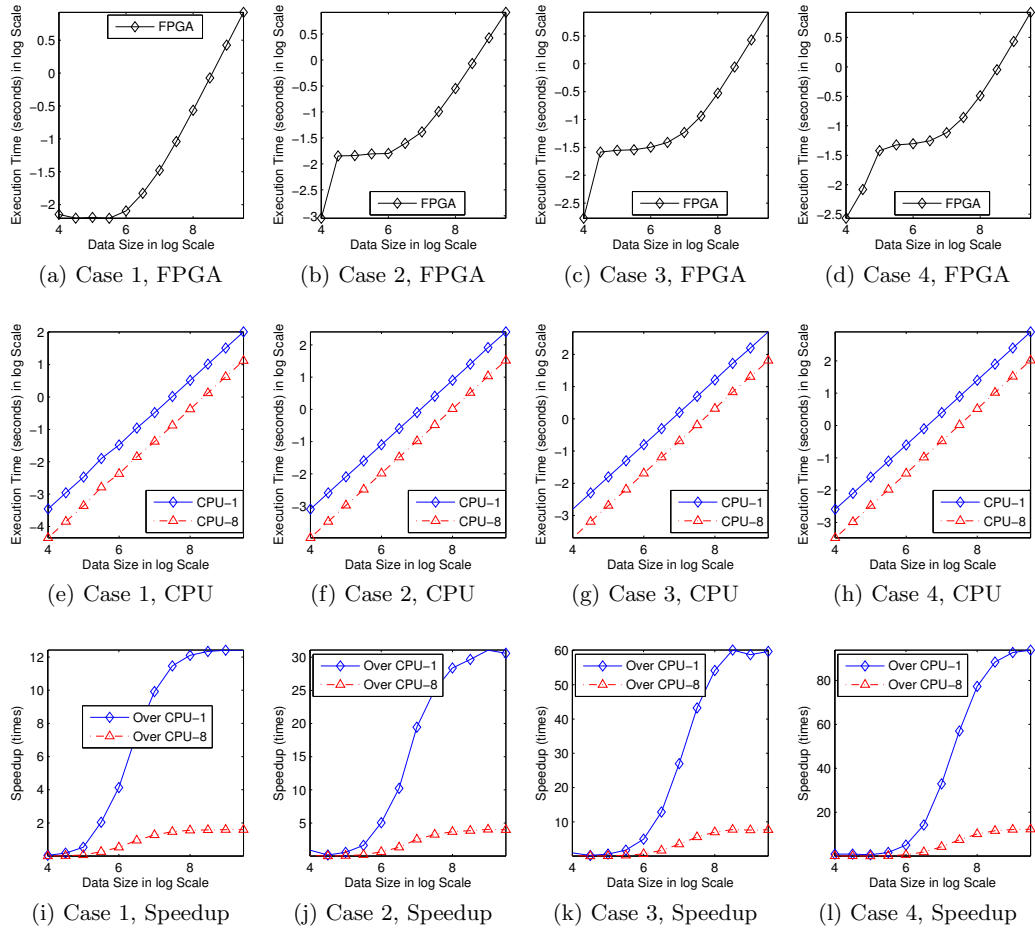


Figure 3.5: Experimental results on intensity evaluation

mentation increases as the size of data set grows. The maximum speedup we observe across all the case studies is in Case 4 when the data size reaches $10^{9.5}$. The FPGA implementation is up to 94 times, and 12 times faster than the CPU implementation on a single core and eight cores respectively.

The execution time of the CPU implementation increases linearly with the data size. However, the growth pattern of the FPGA implementation appears irregular. In Case 2, 3 and 4, there are huge leaps when the data size increases from 10^4 to 10^5 . We are unsure about the reason behind these leaps, but they are likely to be caused by the hardware platform rather than the design. After the leaps, the incremental trends follow convex functions. The convex increment pattern suggests a better computational efficiency than a linear one when the data size is large. Specifically, when the size of the data grows by h times, the execution time grows less than h times. The reason for the observation is that the overhead including hardware initialisation time and post-processing time of the host computer does not increase, while the execution time spent on pipeline processing grows linearly with the size of the data set.

The execution time for a fixed data size is similar across the four case studies regardless the number of dimensions because the numbers of dimensions in all cases are less than or equal to the number of fingers c in each glove. Therefore, the engine is capable of processing a data set with one pass through the data. As a result, the time spent on pipeline processing for different data sets are identical, and the small difference in the total execution times is due to the difference in post-processing time in the host computer.

3.2 Likelihood Evaluation for Univariate Data

Likelihood evaluation is a core calculation in parameter estimation. It is a computationally expensive procedure and requires acceleration. In this section, we present a reconfigurable computing solution to accelerate likelihood evaluation for univariate Hawkes processes using reconfigurable hardware. The main challenge for acceleration is to simplify the data dependency in the likelihood function. In this section, we propose a mathematical transformation for the likelihood function to simplify the data dependency.

3.2.1 Recursive Likelihood Evaluation

During the evaluation of the likelihood function, it is necessary to collect a piece of statistical information from each event in the data. By the definition of the likelihood function, when we collect information from an event, we need to revisit all events before it. In this section, we derive an expression for the likelihood function. When we collect information from an event using the proposed expression, we only need to revisit one event before it.

When the data only has a single type of events, we only need to record the occurrence times of the events and omit the event labels. In other words, we may write an event sequence of K events as

$$\mathcal{E}(0, T) = (t_0, t_1, t_2, \dots, t_{K-1}) \quad (3.7)$$

where T is the termination time of data collection. If this time is unavailable, we assume that the arrival time of last event is the termination time following [47]. In other words, we take $T = t_{K-1}$. With a single event type, the parameter set is in the form $\theta = (\lambda^\perp, \alpha, \beta)$. The partial likelihood [110] of the Hawkes process is

$$L(\theta) = \int_0^T \ln \lambda(t) dC(t) - \int_0^T \lambda(t) dt \quad (3.8)$$

The expression of the likelihood is the difference of two terms. The first term equals the sum of the logarithms of intensities at the occurrence times of all events, i.e.

$$\int_0^T \ln \lambda(t) dC(t) = \sum_{k=0}^{K-1} \ln \lambda(t_k) \quad (3.9)$$

The second term of the likelihood expression is the integration of the intensity function over the time interval of data collection. We may rewrite this term in a computable form as follows:

$$\int_0^T \lambda(t) dt = \int_0^T (\lambda^\perp + \alpha \sum_{k=0}^{K(t)-1} e^{-\beta(t-t_k)}) dt \quad (3.10)$$

$$= \lambda^\perp T + \int_0^T (\alpha \sum_{k=0}^{K(t)-1} e^{-\beta(t-t_k)}) dt \quad (3.11)$$

$$= \lambda^\perp T - \sum_{k=0}^{K-1} \frac{\alpha}{\beta} (e^{-\beta(T-t_k)} - 1) \quad (3.12)$$

We then rewrite the expression of the partial likelihood as

$$L(\theta) = \sum_{k=0}^{K-1} \ln \lambda(t_k) - (\lambda^\perp T - \sum_{k=0}^{K-1} h(k)) \quad (3.13)$$

$$= \sum_{k=0}^{K-1} (\ln \lambda(t_k) + h(k)) - \lambda^\perp T \quad (3.14)$$

where $h(k)$ is a term depending on the termination time of sampling and the time of the k -th

event defined by

$$h(k) = \frac{\alpha}{\beta}(e^{-\beta(T-t_k)} - 1) \quad (3.15)$$

One needs the values of $h(k)$ and $\lambda(t_k)$ for all $k \in [0..(K-1)]$ in the computation of $L(\theta)$. It is relatively easy to calculate $h(k)$ using its definition in Equation 3.15. This calculation is unlikely to cause problems in reconfigurable computing as the data dependency pattern is simple. Specifically, the value of $h(k)$ only depends on the parameter set θ and the occurrence time of the k -th event t_k . However, the data dependency pattern of $\lambda(t_k)$ is complex. Specifically, by the definition of the intensity function in Equation 2.13, we have

$$\lambda(t_k) = \lambda^\perp + \alpha \sum_{k'=0}^{k-1} e^{-\beta(t_k-t_{k'})} \quad (3.16)$$

It is difficult to calculate $\lambda(t_k)$ by Equation 3.16 on the reconfigurable computing platform, because $\lambda(t_k)$ depends on $(t_0, t_1, t_2, \dots, t_{k-1})$.

We present an alternative way to compute $\lambda(t_k)$ to simplify the data dependency pattern. The exponential response function provides the opportunity to eliminate the complex data dependency in the computation of $\lambda(t_k)$. We achieve this by applying the likelihood decomposition technique in [45] to our exponential response function. Let

$$\phi(k) = \sum_{k'=0}^{k-1} e^{-\beta(t_k-t_{k'})} \quad (3.17)$$

Then the intensity for any k are in the form

$$\lambda(t_k) = \lambda^\perp + \alpha\phi(k) \quad (3.18)$$

If $\phi(k)$ is available, it is easy to calculate the intensity $\lambda(t_k)$ as well as the logarithm of the intensity $\lambda(t_k)$. Therefore we find a way to compute $\phi(k)$ in a recursive manner.

- When $k = 0$, there exist no points before t_k . The summation operator in Equation 3.17 always equals zero regardless the parameter β . As a result,

$$\phi(0) = 0 \quad (3.19)$$

- When $k > 0$, we may decompose $\phi(k)$ as follows:

$$\phi(k) = \sum_{k'=0}^{k-2} e^{-\beta(t_k - t_{k'})} + e^{-\beta(t_k - t_{k-1})} \quad (3.20)$$

$$= \sum_{k'=0}^{k-2} e^{-\beta[(t_k - t_{k-1}) + (t_{k-1} - t_{k'})]} + e^{-\beta(t_k - t_{k-1})} \quad (3.21)$$

$$= e^{-\beta(t_k - t_{k-1})} \sum_{k'=0}^{k-2} e^{-\beta(t_{k-1} - t_{k'})} + e^{-\beta(t_k - t_{k-1})} \quad (3.22)$$

$$= e^{-\beta(t_k - t_{k-1})} \left(\sum_{k'=0}^{k-2} e^{-\beta(t_{k-1} - t_{k'})} + 1 \right) \quad (3.23)$$

On the other hand, by Equation 3.17,

$$\sum_{k'=0}^{k-2} e^{-\beta(t_{k-1} - t_{k'})} = \phi(k-1) \quad (3.24)$$

Substitute Equation 3.24 into Equation 3.23, we obtain:

$$\phi(k) = e^{-\beta(t_k - t_{k-1})} (\phi(k-1) + 1) \quad (3.25)$$

Unifying the above two cases, we have

$$\phi(k) = \begin{cases} 0 & \text{if } k = 0 \\ e^{-\beta(t_k - t_{k-1})} (\phi(k-1) + 1) & \text{if } k > 0 \end{cases} \quad (3.26)$$

Equation 3.26 suggests that one may compute $\phi(k)$ using $\phi(k-1)$ without knowing $t_{k'}$ for any $k' \in [0..(k-2)]$. The transformation is particularly useful for hardware design as it eliminates the data dependency in Equation 3.17.

For hardware efficiency, we compute the likelihood function with sequential data access and simple data dependency. We define

$$Q(k) = \sum_{k'=0}^{k-1} (\ln \lambda(t_{k'}) + h(k')) \quad (3.27)$$

The likelihood function in terms of Q is as follows:

$$L(\theta) = Q(K) - \lambda^\perp T \quad (3.28)$$

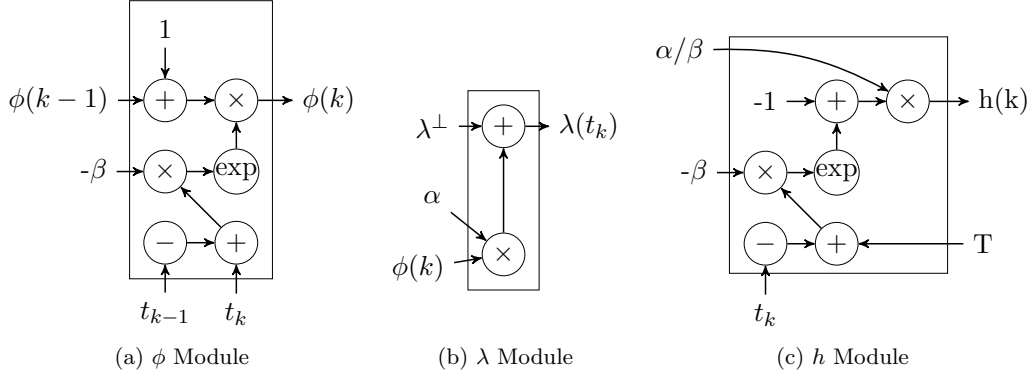


Figure 3.6: Basic modules

On the other hand, $Q(k)$ is recursively computable by

$$Q(k) = \begin{cases} 0 & \text{if } k = 0 \\ Q(k-1) + \ln \lambda(t_k) + h(k) & \text{if } k > 0 \end{cases} \quad (3.29)$$

3.2.2 Hardware Design and Optimisation

We map the likelihood evaluation method derived in the previous section to a pipelined hardware architecture. We first describe three hardware modules and discuss a method to combine them to compute $Q(k)$ efficiently. We propose three arithmetic modules: the ϕ module, λ module and h module. Their structures are shown in Figure 3.6.

1. The ϕ module is designed based on Equation 3.26. The base case of the function when $k = 0$ is not reflected in the hardware. By supplying $\phi(-1) = -1$ to the ϕ module, we can obtain $\phi(0) = 0$.
2. The λ module is designed based on Equation 3.18.
3. The h module is based on Equation 3.15. One input to the h module, α/β , does not change over time. We precompute this value in the host computer to avoid unnecessary resource consumption for the reconfigurable device.

A combination of the basic modules, the Q module, is shown in Figure 3.6. This module is designed based on Equation 3.29. It computes $Q(k)$ and $\phi(k)$ by taking $Q(k-1)$, $\phi(k-1)$ and model parameters as the inputs.

Due to the latency of the module, it is impossible to use an output of $Q(k)$ as the input immediately for the next iteration. Therefore, evaluating the likelihood for a single parameter set will result in pipeline stalls. We apply the c-slow retiming [111] to process a batch of N_θ parameter sets to keep the pipeline busy. Specifically, the Q module loads an event t_k every

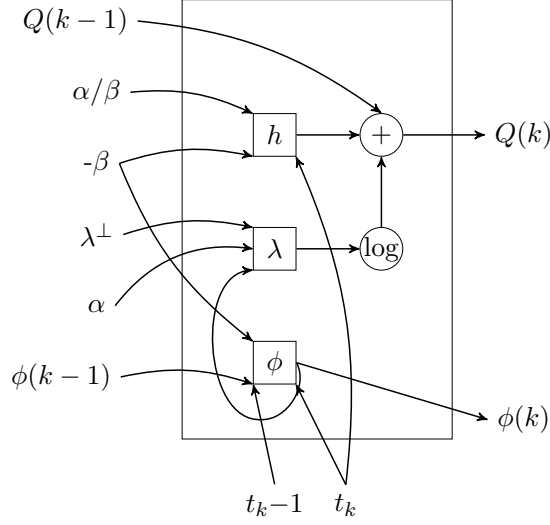


Figure 3.7: Q Module

N_θ cycles. During the next N_θ cycles, the Q module starts the computation of $Q(k)$ for a parameter set at the beginning of each cycle. We also deploy multiple Q modules to exploit parallelism.

3.2.3 Experimental Evaluation

We implement the proposed architecture with all the discussed optimisations in an FPGA-based acceleration platform and test its computational efficiency using data sets with various sizes. We develop a Hawkes process simulator based on the thinning procedure [45] to produce the test data. Using the simulator, we generate a sequence of 10^9 events for our experiments. We deploy twelve Q modules in parallel to maximise hardware resource usage for better computational efficiency. We evaluate 1024 parameter sets in a batch to avoid pipeline stalls. We use the IEEE single precision floating point numbers throughout the FPGA and CPU implementations except for the inputs of the logarithm evaluation unit in FPGA, where we use double precision floating point numbers for better accuracy. A likelihood value computed by the FPGA implementation may slightly differ from the one computed by the CPU implementation, but the relative difference is below 0.1% in all the experiments.

3.2.4 Results and Discussion

The experimental results are plotted in Figure 3.8. The execution times for a query are recorded in Figure 3.8a. The speedup of the FPGA implementation over the CPU implementation on a single core and eight cores are presented in Figure 3.8b. We record the data size in

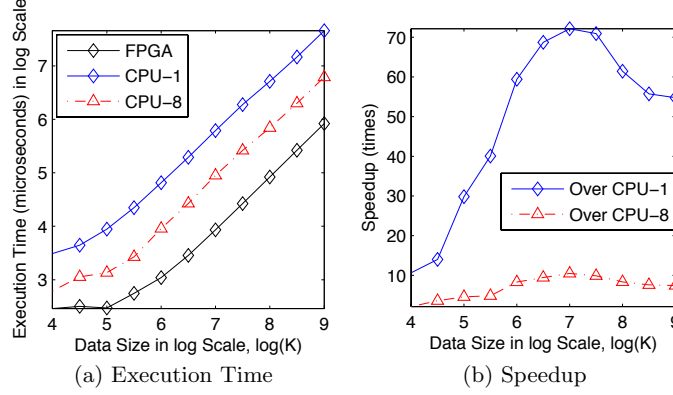


Figure 3.8: Experimental results

log scale with base 10 in both figures. To reflect the trend of the increment of the execution time, we also plot the results in log scale in Figure 3.8a.

The execution times of both the CPU and FPGA implementation increase approximately linearly as the data size grows. The CPU implementation works well on eight cores. It is around 7.1 times faster than a single core.

When data size $K \leq 10^7$, the speedup of the FPGA implementation over the CPU implementation increases as the size of the data set grows. This is because, while the execution time spent on the pipeline grows linearly with the size of the data set, the overhead including the hardware initialisation time and post-processing time does not increase. The peak speedup is 72 times and 10 times over the CPU implementation on a single core and eight cores respectively.

When $K > 10^7$, the speedup of the FPGA implementation decreases slightly and stabilises at around 55 times over a single core. We find that the increment of execution time of the FPGA implementation follows a linear pattern but that of the CPU implementation is slightly below a linear trend when $K > 10^7$. We are unsure about the reason behind this observation, but a possible explanation is that the compiler optimised the code for large data volume.

3.3 Likelihood Evaluation for Multivariate Data

In this section, we discuss a way to build a reconfigurable accelerator for likelihood evaluation for multivariate Hawkes process models. Similar to the univariate case, the major challenge of the likelihood evaluation for multivariate Hawkes process is the complex data dependency. However, it is impossible to simplify the data dependency using the method proposed in the previous section, because the likelihood function of the multivariate Hawkes process is more general than that of the univariate case. We address the challenge by deriving a mathematical

transformation for the likelihood function of the multivariate Hawkes process.

3.3.1 Straightforward Likelihood Evaluation

In this subsection, we analyse the data dependency problem in the straightforward method of likelihood evaluation. The likelihood of a multivariate self-exciting point process is the sum of likelihoods of all dimensions. The likelihood of a single dimension [112] is

$$L_m(\theta) = \int_0^T \ln \lambda_m(t) dC_m(t) - \int_0^T \lambda_m(t) dt \quad (3.30)$$

$$= \sum_{k=0}^{K_m-1} \ln \lambda_m(t_{m,k}) - \lambda_m^\perp T + \sum_{k=0}^{K_m-1} h_m(k) \quad (3.31)$$

where

$$h_m(k) = \sum_{m'=0}^{M-1} \frac{\alpha_{m,m'}}{\beta_{m,m'}} (e^{-\beta_{m,m'}(T-t_{m',k})} - 1) \quad (3.32)$$

Therefore the likelihood of the whole process is

$$L(\theta) = \sum_{m=0}^{M-1} L_m(\theta) \quad (3.33)$$

$$= \sum_{m=0}^{M-1} \left(\sum_{k=0}^{K_m-1} \ln \lambda_m(t_{m,k}) - \lambda_m^\perp T + \sum_{k=0}^{K_m-1} h_m(k) \right) \quad (3.34)$$

$$= Q + R + S \quad (3.35)$$

where

$$Q = \sum_{m=0}^{M-1} \sum_{k=0}^{K_m-1} \ln \lambda_m(t_{m,k}) \quad (3.36)$$

$$R = -T \sum_{m=0}^{M-1} \lambda_m^\perp \quad (3.37)$$

$$S = \sum_{m=0}^{M-1} \sum_{k=0}^{K_m-1} h_m(k) \quad (3.38)$$

We may compute the likelihood by adding up the three components Q, R and S. Among the three components, R is easy to compute with a single multiplication. We focus on the

computation of Q and S . In other words, we have

$$Q = \sum_{i=0}^{N-1} \ln \lambda(o_i) \quad (3.39)$$

where

$$\lambda(o_i) = \lambda_{t_i}(t_i) \quad (3.40)$$

Q is computable if the intensities for all observations, i.e. $\lambda(o_i)$ for $i \in [0..(N-1)]$, are available. S is in a complex and data-dependent form, but we may calculate the contribution of each event towards S independently. By Equation 3.38,

$$S = \sum_{m=0}^{M-1} \sum_{k=0}^{K_m-1} \sum_{m'=0}^{M-1} \frac{\alpha_{m,m'}}{\beta_{m,m'}} (e^{-\beta_{m,m'}(T-t_{m',k})} - 1) \quad (3.41)$$

An equivalent form of S is

$$S = \sum_{m=0}^{M-1} \sum_{i=0}^{N-1} \frac{\alpha_{m,l_i}}{\beta_{m,l_i}} (e^{-\beta_{m,l_i}(T-t_i)} - 1) \quad (3.42)$$

The value of S stays unchanged if the two summation operators are swapped. Therefore

$$S = \sum_{i=0}^{N-1} \sum_{m=0}^{M-1} \frac{\alpha_{m,l_i}}{\beta_{m,l_i}} (e^{-\beta_{m,l_i}(T-t_i)} - 1) \quad (3.43)$$

Assume that we can compute the following two quantities for each data point o_i :

$$s_i = \sum_{m=0}^{M-1} \frac{\alpha_{m,l_i}}{\beta_{m,l_i}} (e^{-\beta_{m,l_i}(T-t_i)} - 1) \quad (3.44)$$

$$q_i = \ln \lambda(o_i) \quad (3.45)$$

Then we have

$$S + Q = \sum_{i=0}^{N-1} (s_i + q_i) \quad (3.46)$$

In other words, one may obtain $(S + Q)$ by accumulating $(s_i + q_i)$ for all $i \in [0..(N-1)]$. The effort to compute $(S + Q)$ depends on the efficiency of intensity evaluation, as we need to evaluate $\lambda(o_i)$ for all observations o_i . Therefore, given a data set with N data points, one needs to evaluate N intensity values.

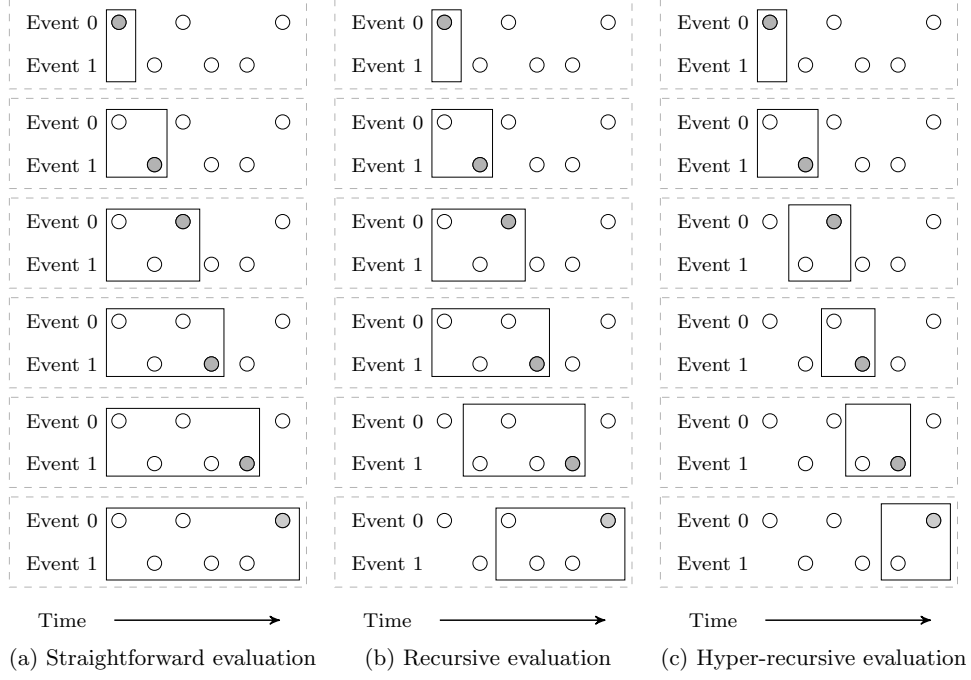


Figure 3.9: Data dependency pattern in different intensity evaluation methods

3.3.2 Straightforward Intensity Evaluation

The computable form of the intensity is as follows:

$$\lambda_m(t) = \lambda_m^\perp + \sum_{m'=0}^{M-1} \sum_{k'=0}^{K_m(t)-1} \alpha_{m,m'} e^{-\beta_{m,m'}(t-t_{m',k'})} \quad (3.47)$$

where

$$K_m(t) = |\mathcal{E}_m(0, t)| \quad (3.48)$$

This equation suggests that, when computing the intensity at time t , it is necessary to collect a piece of statistical information from each point appearing before time t .

The computation of likelihood only requires the intensity values when an event occurs. Therefore, rather than considering intensities for arbitrary time points, we only pay attention to the intensities in the form

$$\lambda(o_i) = \lambda_{l_i}^\perp + \sum_{j=0}^{i-1} \alpha_{l_i, l_j} e^{-\beta_{l_i, l_j}(t_i - t_j)} \quad (3.49)$$

This expression is computable, but the data dependency cause problems in computational efficiency.

The data dependency pattern underlying this expression is Figure 3.9a. In this figure, each dotted square is a scenario when the intensity of the data point in gray is being evaluated. The points enclosed in the solid box are accessed and used in the evaluation. Given an observation o_i , the data points needed for $\lambda(o_i)$ are $o_0, o_1, o_2, \dots, o_{i-1}$. Admittedly, this is a predictable pattern where the locations of the depended data points are only related to the index i rather than the distribution of the data. However, this strategy is not efficient because it takes $O(N^2)$ time to compute all intensities for a data set with N points. The time complexity suggests that the likelihood evaluation time increases quadratically as the number of data point grows. In this case, even if we develop a hardware accelerator for likelihood evaluation, the computational efficiency may not scale well with the data size.

3.3.3 Recursive Intensity Evaluation

As the straightforward intensity evaluation method incurs high time complexity, we need to reduce the time complexity before eliminating the complex data dependency. In this subsection, we derive a recursive likelihood evaluation strategy to reduce the time complexity.

The intensity for an event o_i is in the form

$$\lambda(o_i) = \lambda_{l_i}^\perp + \sum_{m'=0}^{M-1} \sum_{j=0}^{i-1} z_{l_j, m'} \alpha_{l_i, m'} e^{-\beta_{l_i, m'}(x_i - x_j)} \quad (3.50)$$

where

$$z_{a,b} = \begin{cases} 0, & a = b \\ 1, & a \neq b \end{cases} \quad (3.51)$$

The value of $\alpha_{l_i, m'}$ are independent of j . Therefore, we may move it out of the inner summation operator. The intensity at o_i becomes

$$\lambda(o_i) = \lambda_{l_i}^\perp + \sum_{m'=0}^{M-1} \alpha_{l_i, m'} \sum_{j=0}^{i-1} z_{l_j, m'} e^{-\beta_{l_i, m'}(x_i - x_j)} \quad (3.52)$$

Let

$$r_{i, m'} = \sum_{j=0}^{i-1} z_{l_j, m'} e^{-\beta_{l_i, m'}(x_i - x_j)} \quad (3.53)$$

Then the expression of intensity can be transformed into

$$\lambda(o_i) = \lambda_{l_i}^\perp + \sum_{m'=0}^{M-1} \alpha_{l_i, m'} r_{i, m'} \quad (3.54)$$

This equation suggests that $\lambda(o_i)$ is computable in $O(M)$ time if $r_{i, m'}$ for all $m' \in [0..(M-1)]$ are available. Therefore, the efficiency of intensity evaluation depends on the efficiency of computing $r_{i, m}$.

We then focus on the computation of $r_{i, m}$ in hardware pipelines. For ease of discussion, we define a quantity

$$\iota(i, m^*) = \max\{i' : l_{i'} = m^* \text{ and } i' < i\} \quad (3.55)$$

In other words, $\iota(i, m^*)$ is the maximum index of the points with the label m^* appearing before o_i . The historical points appearing before o_i can be divided into two types: (i) those occur before $o_{\iota(i, l_i)}$, namely $o_0, o_1, o_2, \dots, o_{\iota(i, l_i)-1}$; and (ii) those do not occur before $o_{\iota(i, l_i)}$, namely $o_{\iota(i, l_i)}, o_{\iota(i, l_i)+1}, \dots, o_{i-1}$. Then $r_{i, m}$ can be decomposed as

$$r_{i, m'} = u_{i, m'} + v_{i, m'} \quad (3.56)$$

where

$$u_{i, m'} = \sum_{j=0}^{\iota(i, l_i)-1} z_{m', l_j} e^{-\beta_{l_i, m'}(x_i - x_j)} \quad (3.57)$$

$$v_{i, m'} = \sum_{j=\iota(i, l_i)}^{i-1} z_{m', l_j} e^{-\beta_{l_i, m'}(x_i - x_j)} \quad (3.58)$$

The two quantities, $u_{i, m'}$ and $v_{i, m'}$ contains the statistical information collected from points of type (i) and type (ii) respectively. One may compute $u_{i, m}$ efficiently based on $r_{\iota(i, l_i), m'}$. Specifically, the definition of $r_{i, m'}$ in Equation 3.53 can be rewritten as

$$r_{\iota(i, l_i), m'} = \sum_{j=0}^{\iota(i, l_i)-1} z_{m', l_j} e^{-\beta_{m_{\iota(i, l_i)}, m'}(x_{\iota(i, l_i)} - x_j)} \quad (3.59)$$

From the definition of $\iota(i, m^*)$ in Equation 3.55,

$$m_{\iota(i, l_i)} = l_i \quad (3.60)$$

Substituting Equation 3.60 into Equation 3.59,

$$r_{\iota(i,l_i),m'} = \sum_{j=0}^{\iota(i,l_i)-1} z_{m',l_j} e^{-\beta_{l_i,m'}(x_{\iota(i,l_i)}-x_j)} \quad (3.61)$$

On the other hand, we rewrite $u_{i,m'}$ in terms of $r_{\iota(i,l_i),m'}$ by introducing the term $x_{\iota(i,l_i)}$ into Equation 3.57,

$$u_{i,m'} = \sum_{j=0}^{\iota(i,l_i)-1} z_{m',l_j} e^{-\beta_{l_i,m'}[(x_i-x_{\iota(i,l_i)})+(x_{\iota(i,l_i)}-x_j)]} \quad (3.62)$$

$$= \sum_{j=0}^{\iota(i,l_i)-1} z_{m',l_j} e^{-\beta_{l_i,m'}(x_i-x_{\iota(i,l_i)})} e^{-\beta_{l_i,m'}(x_{\iota(i,l_i)}-x_j)} \quad (3.63)$$

$$= e^{-\beta_{l_i,m'}(x_i-x_{\iota(i,l_i)})} \left[\sum_{j=0}^{\iota(i,l_i)-1} z_{m',l_j} e^{-\beta_{l_i,m'}(x_{\iota(i,l_i)}-x_j)} \right] \quad (3.64)$$

$$= e^{-\beta_{l_i,m'}(x_i-x_{\iota(i,l_i)})} r_{\iota(i,l_i),m'} \quad (3.65)$$

By Equation 3.65, one may calculate $u_{i,m'}$ using $x_{\iota(i,l_i)}$ without accessing $o_0, o_1, o_2, \dots, o_{\iota(i,l_i)-1}$.

On the other hand, one may directly compute $v_{i,m'}$ by Equation 3.58. After computing $u_{i,m'}$ and $v_{i,m'}$ for a dimension m' , one may easily obtain $r_{i,m'}$ by Equation 3.56. When $r_{i,m'}$ for all $m \in [0..(M-1)]$ are available, one may compute $\lambda(o_i)$ by Equation 3.54.

We call this method *recursive intensity evaluation* because the two crucial intermediate variable $r_{i,m'}$ and $u_{i,m'}$ can be recursively computed based on each other. Using this method, one may compute an intensity $\lambda(o_i)$ based on $o_{\iota(i,l_i)}, o_{\iota(i,l_i)+1}, \dots, o_i$. In other words, the intensity at o_i is calculated based on the points occurring in all dimensions since the last event of type l_i . This data dependency pattern is shown graphically in Figure 3.9b.

In terms of hardware design, the recursive intensity evaluation method enables us to compute all intensities by a one-off visit through the data. As a result, it is possible to design a fully pipelined hardware accelerator based on the proposed intensity evaluation method if the following two assumptions holds:

- The on-chip memory is sufficiently large to cache $o_{\iota(i,l_i)}, o_{\iota(i,l_i)+1}, \dots, o_i$ at any time for all $i \in [0..(N-1)]$.
- The arithmetic resources are available to extract information from all the cached points in parallel.

However, neither of the two assumptions holds practically. One may encounter the worst case where there are N data points but $\iota(i, m_N) = 0$. To fully prepare for such a worst case, we

need to cache and collect statistical information from $(N-1)$ points using the on-chip memory, but the on-chip memory on a typical reconfigurable device is too small to store $(N-1)$ events.

3.3.4 Hyper-Recursive Intensity Evaluation

The recursive intensity evaluation strategy reduces the time complexity of intensity evaluation, but the data dependency remains complex. In this subsection, we propose the *hyper-recursive intensity evaluation* strategy which eliminates the data dependency. The key insight of this method is that we introduce a group of intermediate variables that (i) contain all necessary statistical information of all past points for intensity computation and (ii) can be computed with a hardware resource requirement that does not scale up with data size N .

Finding such a group of intermediate variables is a very difficult task from the perspective of statistics, information theory and algorithm design, because this variable set needs to be compact, informative and easy to compute. In this study, we manage to find a set of variables that meets all these requirements. Let

$$\omega_{i,m',m^*} = r_{\iota(i,m^*),m'} \quad (3.66)$$

$$\psi_{i,m',m^*} = \sum_{j=\iota(i,m^*)}^{i-1} z_{m',l_j} e^{-\beta_{m^*,m'}(x_{\iota(i,m^*)}-x_j)} \quad (3.67)$$

Then ω_{i,m',m^*} and ψ_{i,m',m^*} for all $(m', m^*) \in [0..(M-1)]^2$ constitute a set of variables meeting all the requirements. When $l_i = m^*$, Equation 3.66 becomes

$$\omega_{i,m',l_i} = r_{\iota(i,l_i),m'} \quad (3.68)$$

$$\psi_{i,m',l_i} = \sum_{j=\iota(i,l_i)}^{i-1} z_{m',l_j} e^{-\beta_{l_i,m'}(x_{\iota(i,l_i)}-x_j)} \quad (3.69)$$

Therefore, by Equation 3.65

$$u_{i,m'} = e^{-\beta_{l_i,m'}(x_i-x_{\iota(i,l_i)})} \omega_{i,m',l_i} \quad (3.70)$$

By Equation 3.58, $v_{i,m'}$ can be expressed in terms of ψ_{i,m',m^*}

$$v_{i,m'} = \sum_{j=\iota(i,l_i)}^{i-1} z_{m',l_j} e^{-\beta_{l_i,m'}(x_i-x_j)} \quad (3.71)$$

$$= \sum_{j=\iota(i,l_i)}^{i-1} z_{m',l_j} e^{-\beta_{l_i,m'}(x_i-x_{\iota(i,l_i)})} e^{-\beta_{l_i,m'}(x_{\iota(i,l_i)}-x_j)} \quad (3.72)$$

$$= e^{-\beta_{l_i,m'}(x_i-x_{\iota(i,l_i)})} \left[\sum_{j=\iota(i,l_i)}^{i-1} z_{m',l_j} e^{-\beta_{l_i,m'}(x_{\iota(i,l_i)}-x_j)} \right] \quad (3.73)$$

$$= e^{-\beta_{l_i,m'}(x_i-x_{\iota(i,l_i)})} \psi_{i,m',l_i} \quad (3.74)$$

Both ω_{i,m',m^*} and ψ_{i,m',m^*} depend on the value of $\iota(i, m^*)$, and $\iota(i, m^*)$ is closely related to l_{i-1} .

- When $l_{i-1} = m^*$, we have $\iota(i, m^*) = i - 1$. Therefore,

$$\omega_{i,m',m^*} = r_{i-1,m'} \quad (3.75)$$

$$\psi_{i,m',m^*} = z_{m',l_{i-1}} e^{-\beta_{m^*,m'}(x_{\iota(i,m^*)}-x_{i-1})} \quad (3.76)$$

- When $l_{i-1} \neq m^*$, we have $\iota(i, m^*) = \iota(i - 1, m^*)$. Therefore,

$$\omega_{i,m',m^*} = \omega_{i-1,m',m^*} \quad (3.77)$$

$$\psi_{i,m',m^*} = \psi_{i-1,m',m^*} + z_{m',l_{i-1}} e^{-\beta_{m^*,m'}(x_{\iota(i,m^*)}-x_{i-1})} \quad (3.78)$$

Unifying the expression of ω_{i,m',m^*} and ψ_{i,m',m^*} and in both cases, we have

$$\omega_{i,m',m^*} = \begin{cases} r_{i-1,m'} & \text{if } l_{i-1} = m^* \\ \omega_{i-1,m',m^*} & \text{if } l_{i-1} \neq m^* \end{cases} \quad (3.79)$$

$$\psi_{i,m',m^*} = \begin{cases} \tilde{\psi}_{i-1,m',m^*} & \text{if } l_{i-1} = m^* \\ \psi_{i-1,m',m^*} + \tilde{\psi}_{i-1,m',m^*} & \text{if } l_{i-1} \neq m^* \end{cases} \quad (3.80)$$

where

$$\tilde{\psi}_{i-1,m',m^*} = \begin{cases} e^{-\beta_{m^*,m'}(x_{\iota(i,m^*)}-x_{i-1})} & \text{if } l_{i-1} = m' \\ 0 & \text{if } l_{i-1} \neq m' \end{cases} \quad (3.81)$$

Equation 3.70 to 3.81 suggest that the proposed variables, ω_{i,m',m^*} and ϕ_{i,m',m^*} for all $(m', m^*) \in [0..(M-1)]^2$, provide sufficient information for intensity computation without

involving complicated data dependences.

When $i = 0$, ω_{1,m',m^*} and ψ_{1,m',m^*} for all $(m', m^*) \in [0..(M-1)]^2$ can easily be calculated using their corresponding definitions in Equation 3.66 and 3.67. When $i > 1$, assuming that $r_{i-1,m'}$, ω_{i-1,m',m^*} and ψ_{i-1,m',m^*} for all $(m', m^*) \in [0..(M-1)] \times [0..(M-1)]$ have been obtained by the time we take the data point o_i . Then $r_{i,m'}$, ω_{i,m',m^*} and ψ_{i,m',m^*} for all $(m', m^*) \in [0..(M-1)]^2$ can be computed by Equation 3.79 and 3.80. Then one may compute $u_{i,m'}$ and $v_{i,m'}$ by Equation 3.70 and 3.74. Finally, $\lambda(o_i)$ can be obtained by evaluating Equation 3.56.

We name this method *hyper-recursive intensity evaluation* because the intensity is obtained from a two-layer recursive computation. The recursively computed variables ω_{i,m',m^*} and ϕ_{i,m',m^*} are required for the recursive computation of $r_{i,m'}$ and $u_{i,m'}$. We consider this method useful for hardware implementation for the following three reasons.

1. The requirement for arithmetic resources scales linearly with the dimensionality of the problem M . The major arithmetic computation in this strategy is the update operation in Equation 3.81 when $l_{i-1} = m'$. For each data point o_i , only the entries in the form ψ_{i,l_{i-1},m^*} satisfy this condition. Therefore, one only needs to deploy M evaluation units for the update operation.
2. The computation of the intensity $\lambda(o_i)$ only requires the data point o_{i-1} . The data dependency pattern of this method is shown in Figure 3.47. There is no need to cache a sequence of data points in this case. As a result, the requirement of hardware resources is independent of the data size.
3. It is easy to parallelise the update operations for ω_{i,m',m^*} and ψ_{i,m',m^*} , because the update operations for all $(m', m^*) \in [0..(M-1)]^2$ are independent of each other by Equation 3.79 and 3.80

Hyper-recursive intensity evaluation is arguably the most general hardware-oriented algorithmic optimisation for the parameter estimation operation of Hawkes process models, as there is no theoretical constraint in the number of data dimensions.

3.3.5 Hardware Design

We develop a pipelined architecture for likelihood evaluation using the hyper-recursive intensity evaluation strategy. We propose a basic hardware module called the *intensity evaluation cell* to compute $r_{i,m'}$, ψ_{i,m',m^*} and ω_{i,m',m^*} for a data point o_i and a dimension m' following the hyper-recursive intensity evaluation strategy discussed in the previous section. In other words, an intensity evaluation cell does not provide the intensity value directly, but it collects statistical information from data so that the intensity value can easily be computed.

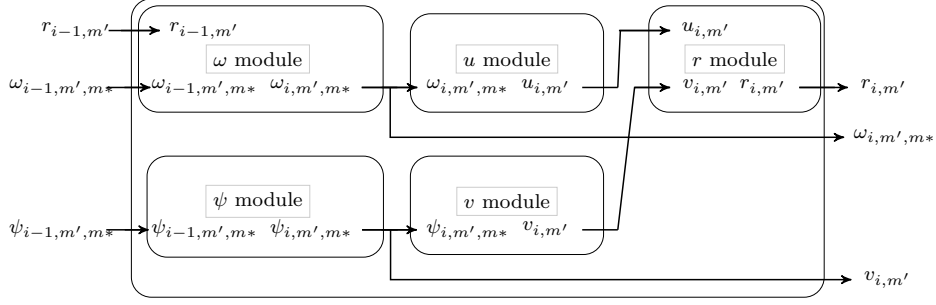


Figure 3.10: Intensity evaluation cell

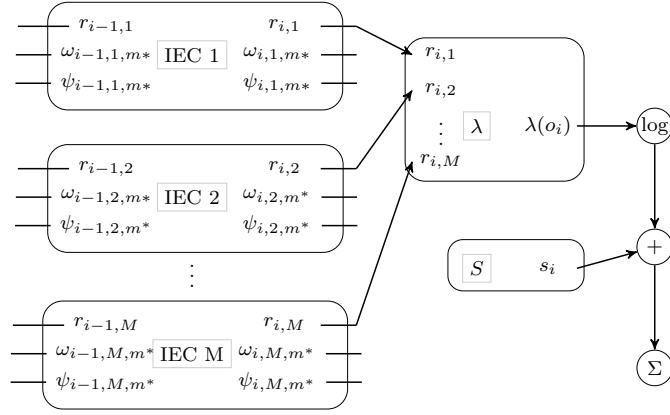


Figure 3.11: Likelihood evaluation accelerator

The structure of the intensity evaluation cell is shown in Figure 3.10. The data input and the parameter set are omitted for brevity. As shown in the figure, an intensity evaluation cell contains five modules. The ω module and ψ module compute Equation 3.79 and 3.80 respectively for $\omega_{i,m',m*}$ and $\psi_{i,m',m*}$ respectively. The u module and v module take the outputs of the $\omega_{i,m',m*}$ and $\psi_{i,m',m*}$ respectively, and then evaluate Equation 3.70 and 3.74 to produce $u_{i,m'}$ and $v_{i,m'}$. The r module then use $u_{i,m'}$ and $v_{i,m'}$ to calculate $r_{i,m'}$ by Equation 3.56.

We design a data path to finalise the likelihood evaluation by combining multiple intensity evaluation cells. The structure of the data path is shown in Figure 3.11. Each block marked with ‘IEC’ represents an intensity evaluation cell. In this architecture, we deploy M intensity evaluation cells to compute $r_{i,0}$, $r_{i,1}$, $\dots$, $r_{i,M-1}$ in parallel for each data point o_i . These results are passed to a λ module which computes the intensity $\lambda(o_i)$ by Equation 3.54. We also design the S module which computes s_i according to Equation 3.43. By accumulating the sum of $\ln(o_i)$ and s_i in an accumulator, we can finally harvest the value of $(S + Q)$ by Equation 3.46.

3.3.6 Experimental Evaluation

We evaluate the proposed likelihood evaluation accelerator with data. We develop simulators to mimic four real-world situations and create four data sets for the experiments, namely ‘rainfall’, ‘earthquake’, ‘neuron’ and ‘highfreq’. The ‘rainfall’ data set contains simulated rainfall records from three observation points; the ‘earthquake’ data set is a simulated log of earthquake detectors in eight locations; the ‘neuron’ data set describes the activity of ten simulated spiking neurons; and the ‘highfreq’ data set is the data collected from a simulated market for high-frequency currency trading. The numbers of dimensions of these four data sets are 3, 8, 10, and 14 respectively. The distribution of the simulated data may be different from real-world data, but the results are useful because the execution time only depends on the data size rather than the data distribution. We deploy 14 intensity evaluation cells in the FPGA to maximise resource usage. This implementation takes 71 percent of fine-grained logic, 38 percent of DSPs and 45 percent of block memory on the FPGA. The clock frequency of the FPGA is set to 120MHz.

The occurrence times of data points are represented in IEEE single precision floating point numbers. The event type labels are represented in 8-bit unsigned integers. We validate the correctness and precision of the system by comparing its results with a MATLAB implementation using the same data sets. The maximum relevant error is less than 1.1 percent.

In addition to the hybrid FPGA-based system, we build a CPU-only implementation that runs exclusively on the host computer of the acceleration platform. This implementation is programmed with the OpenMP library. To make a fair comparison, we apply a series of optimisations to the implementation, including avoiding redundant memory access and selecting appropriate parallelisation scheme. Both the host code for the FPGA implementation and the CPU implementation are written in the C programming language.

A data set is loaded into the main memory of the host computer before the computation begins. The data are then transmitted to the acceleration system via the PCI-Express interface in real time during the computation. The execution time includes the data transmission overhead.

Experimental results for likelihood evaluation are shown in Figure 3.12. The first row of figures show the execution times for likelihood evaluation. We record the data size in log scale with base 10 in these figures. We also record the the execution time in log scale to reflect the trend of the increment of the execution time. As we use the log axis for execution time, a small difference along the vertical direction in each figure means a huge difference in execution time. The second row of the figures shows the speedup of the FPGA-based system over one CPU core and eight CPU cores.

The maximum speedup for likelihood evaluation is achieved when $M = 14$ and the data size reaches 10^7 . In this case, the FPGA-based implementation is respectively 67 times and 11

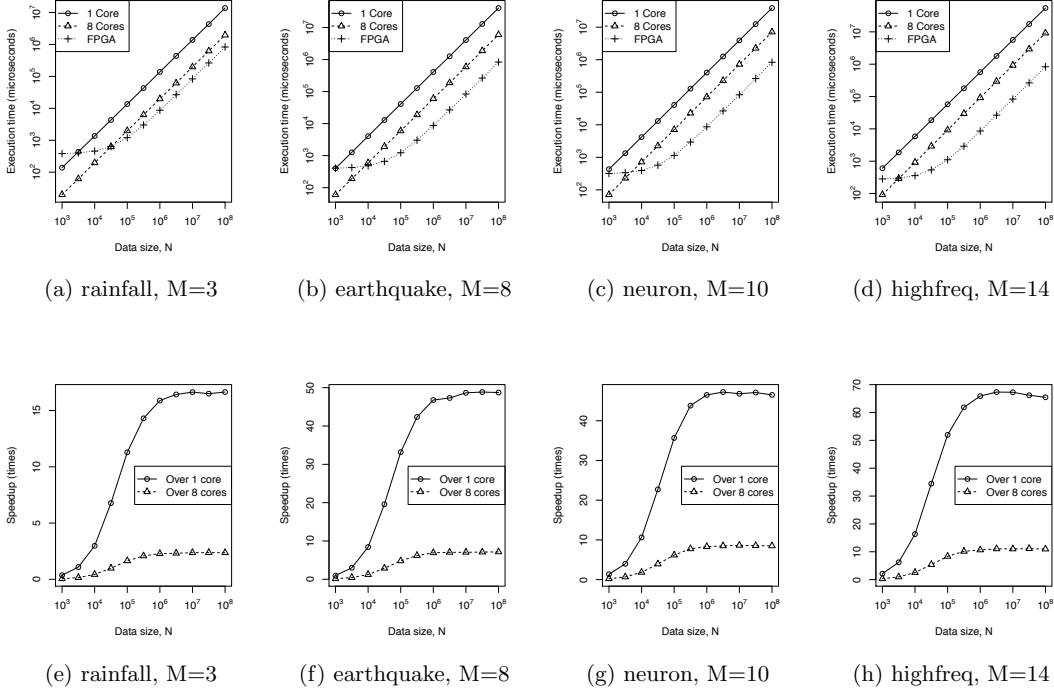


Figure 3.12: Computational efficiency for likelihood evaluation

times faster than a single CPU core and eight CPU cores for likelihood evaluation. Moreover, we obtain the following three observations from the experimental results:

- The FPGA-based system demonstrates higher speedup when the data volume is large. This is because the execution time of the FPGA-based system includes the hardware initialisation time. The overhead becomes less significant when the data volume is large.
- Although not clear in the figures, the execution time of the CPU implementation is slightly below the linear trend. As a result, the speedup of the FPGA-based implementation slightly drops when the data size reaches 10^7 . We believe that such an observation is caused by compiler optimisation of the CPU-only implementation.
- The execution time curve of the FPGA-based implementation intersects with that of the CPU-only system. The intersect points tend to shift to the left when the number of dimensions increases. This is because the execution time of the FPGA-based implementation is less affected by M than the CPU implementation, while the time of CPU implementation grows linearly with M .

3.4 Host-Accelerator Collaboration for Parameter Estimation

Simulated annealing [113, 114] is a powerful method for parameter search. We may incorporate the simulated annealing algorithm and our proposed likelihood evaluation accelerator to solve the parameter estimation problem. In this subsection, we first discuss a potential efficiency problem in parameter estimation with the proposed accelerator. We then present the *eager preparation method* which solves the efficiency problem by deliberately computing redundant information, and by rearranging the computation steps.

3.4.1 Straightforward Simulated Annealing

Assume that an implementation of the likelihood evaluation engine evaluates likelihood values for a batch of K parameters in each run, we create K searching threads accordingly. Let $\theta_{g,k}$ be the parameter set of thread k in iteration g . We first randomly pick initial parameter sets $\theta_{0,0}, \theta_{0,1}, \dots, \theta_{0,K-1}$, and then execute the following three steps repetitively.

1. Neighbour generation: the host computer generates neighbour parameter sets $\theta'_{g,0}, \theta'_{g,1}, \dots, \theta'_{g,K-1}$ for the current parameter sets $\theta_{g,0}, \theta_{g,1}, \dots, \theta_{g,K-1}$.
2. Likelihood evaluation: the accelerator reads the neighbour parameter sets $\theta'_{g,0}, \theta'_{g,1}, \dots, \theta'_{g,K-1}$ from the host computer, and then evaluate the corresponding likelihood values $L(\theta'_{g,0}), L(\theta'_{g,1}), \dots, L(\theta'_{g,K-1})$ using the data set.
3. Post-processing: for each neighbour parameter set $\theta'_{g,k}$, the host computer judges whether $L(\theta'_{g,k})$ is larger than the largest likelihood found in previous iterations. If so, the host computer updates the best solution to $\theta_{g,k}$. The host computer decides whether to accept $\theta'_{g,k}$ as the parameter set of iteration $(g+1)$. In other words, it decides whether $\theta_{g+1,k} = \theta_{g,k}$ or $\theta_{g+1,k} = \theta'_{g,k}$.

The three steps are executed until a satisfactory solution emerges or the time limit is reached. A Gantt chart of the computational steps is shown in Figure 3.13a. Typically, the execution times of the three steps are different. Likelihood estimation is the most time-consuming part in each iteration when the data volume is sufficient large. We may use the accelerator proposed in the previous section to accelerate likelihood evaluation.

The neighbour generation step and the post-processing step are not directly related to data. Mapping these steps to hardware accelerator may lead to a waste of valuable on-chip logical resources. However, when executing the two steps in the host computer, one may encounter a computational efficiency bottleneck in the neighbour generation step. The execution time of this step is considerable compared to post-processing. Specifically, the neighbour generation algorithm is a key to reduce the number of iterations. It must be

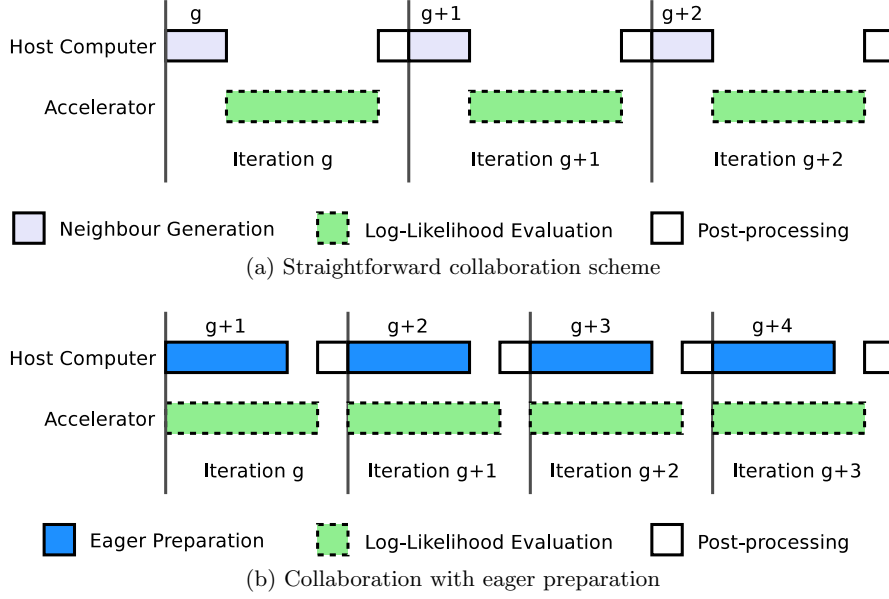


Figure 3.13: Gantt chart of two host-accelerator collaboration schemes

carefully designed with heuristics to avoid bad neighbours [115, 116]. In contrast, post-processing typically only involve the evaluation of simple expressions [117].

3.4.2 Eager Preparation

The Gantt chart in Figure 3.13a shows that when the accelerator is working, the host computer is idle. Therefore, the total execution time can be shortened if the host computer is busy while the accelerator is busy. However, it is impossible to rearrange the workflow because the three steps can only be executed in a strictly serial manner to guarantee correctness. In iteration g , the neighbour generation step requires the post-processing result of iteration $(g - 1)$. The accelerator can only be invoked when all the neighbour parameter sets of iteration g have been generated.

We propose the *eager preparation* method to parallelise neighbour generation and likelihood computation. When we begin to evaluate the likelihood using the accelerator in iteration g , if we know the exact parameter sets $\theta_{g+1,0}, \theta_{g+1,1}, \dots, \theta_{g+1,K-1}$, we can compute the neighbours $\theta'_{g+1,0}, \theta'_{g+1,1}, \dots, \theta'_{g+1,K-1}$ for all search threads while the accelerator is busy with likelihood evaluation for iterator g . Practically, it is impossible to know the new parameter sets until the post-processing step is done. However, we know that a new parameter set $\theta_{g+1,k}$ must be in one of the two cases (i) $\theta_{g+1,k} = \theta_{g,k}$ or (ii) $\theta_{g+1,k} = \theta'_{g,k}$. Therefore, rather than predicting the new parameter sets and produce their neighbours, we propose to generate neighbours for both cases for the K threads.

We name this method *eager preparation* because the host computer eagerly prepare neighbours for the next iteration regardless which parameter set will be accepted. The eager preparation operation runs on the host computer while the accelerator is busy with the evaluation of likelihood values. By adopting eager preparation in the workflow, the neighbour generation step can be removed because in an iteration g , the neighbours for both possible parameter sets for all searching threads must have been generated in the eager preparation step of the iteration $(g - 1)$. In this case, one only needs to pick the correct neighbours to invoke the accelerator. The Gantt chart of this optimised workflow is shown in Figure 3.13b.

With the eager preparation method, we deliberately compute redundant information. For K searching threads, $2K$ neighbours are generated in each iteration. Only K of them are used in the next iteration, while the remaining K neighbours are discarded. For a given parameter estimation task, the execution time of eager preparation in the optimised workflow is around twice the time of neighbour generation in the original workflow. Also, the execution time of neighbourhood generation is less than that of likelihood evaluation when the data volume is large. Therefore, the time for likelihood evaluation on the accelerator may completely cover the execution time of eager preparation on the host computer.

3.4.3 Experimental Evaluation

In this subsection, we evaluate the computational efficiency of parameter estimation. We use the same hardware platforms and data sets with the experiments in Section 3.3.6. Experimental results for parameter estimation are shown in Figure 3.14. The maximum speedup is achieved when $M = 14$ and the data size reaches $10^{7.5}$. The system is up to 78 times faster than a single-core CPU and 11 times faster than the same CPU with eight cores. We also obtain the following three observations from the results:

- The general trends of experimental results in parameter estimation is similar to the trends in likelihood evaluation in Figure 3.12, because the likelihood evaluation dominates the execution time of parameter estimation.
- The growth of the execution time of the CPU implementation does not follow a linear growth pattern when the data size is small, which is different from the results on likelihood evaluation. This observation is due to the time taken by neighbourhood generation.
- The execution time of the FPGA-based implementation stays unchanged when data size is small, because the likelihood evaluation time on the accelerator cannot cover the execution time of the eager preparation process on the host computer. In this case, the eager preparation time, which is independent of the data size, keeps the execution time unchanged.

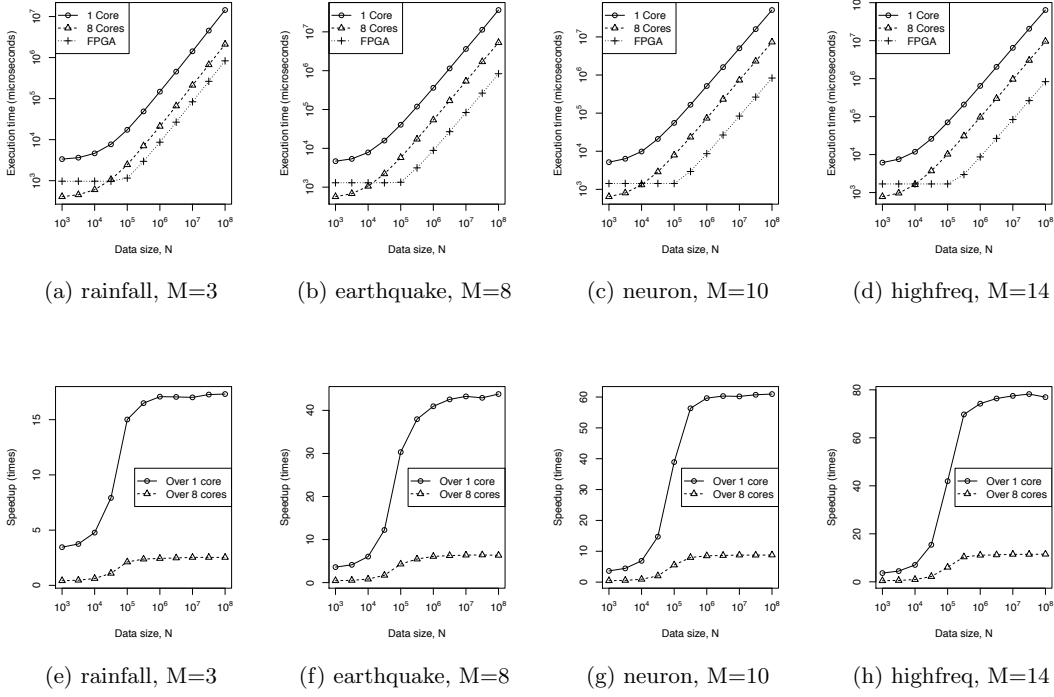


Figure 3.14: Computational efficiency in parameter estimation

3.5 Summary

This chapter discusses the acceleration of three operations for the Hawkes point process namely intensity evaluation, likelihood evaluation and parameter estimation. Section 3.1 presents a hardware architecture that accelerates intensity evaluation for multivariate Hawkes point process models. We analyse the data dependency in the computation and present a strategy to compute multiple intensities with a single pass through the data. Section 3.2 presents a hardware architecture for likelihood evaluation for the univariate Hawkes point process models. We derive a mathematical method to eliminate complex data dependency in the likelihood function. Section 3.3 generalises the likelihood evaluation strategy for the univariate Hawkes process to the multivariate case. The core technique of the generalised strategy is the hyper-recursive intensity evaluation method which eliminates complex data dependency. By mapping the proposed strategy to reconfigurable hardware, we design an accelerator for likelihood evaluation. Section 3.4 discusses an acceleration solution for parameter estimation using the likelihood evaluation accelerator. We propose the eager preparation method where we deliberately compute redundant information to gain the flexibility in workflow scheduling, which is the key technique to the reduce total execution time for parameter estimation.

Chapter 4

Event Stream Prediction with Regression

The conventional way to make predictions for event streams is to estimate a point process model using data and simulate the model. However, we are aware of the efficiency problem in the parameter estimation process of point process models. Also, the predictive accuracy of a point process model may be limited when model is unable to capture the data distribution. In this chapter, we develop an alternative predictive model for event streams without point process models. The main novel aspects in this chapter are as follows:

- An approximate intensity function with a counting technique which is inspired by the Hawkes point process. (Section 4.1)
- A novel predictive model for event streams based on the counting technique and regression analysis. (Section 4.2)
- An empirical study comparing two predictive systems based on the proposed model and a system based on the Hawkes point process model. (Section 4.3)

The organisation of this chapter is as follows. Section 4.1 presents a method to compute approximate intensity values for the Hawkes point process. Section 4.2 proposes a novel predictive model based on counting and regression. Section 4.3 describes an empirical evaluation of the proposed predictive model. Section 4.4 gives a brief summary of this chapter.

4.1 Approximate Intensity Evaluation

Intensity evaluation for Hawkes point process models is computationally demanding and difficult to accelerate. In this section, we solve this problem by proposing a method to compute approximate intensities based on a counting technique so that point process models are no

longer needed. The proposed method, shown in Algorithm 2 and Algorithm 3, provides intensities by counting the number of events in a collection of intervals in the recent history.

In Chapter 3, we have used two methods to compute the intensity of the Hawkes point process using reconfigurable hardware. The first method is a straightforward approach where we compute the intensities based on the definition of the intensity function. To compute the intensity values at time q , it is necessary to visit all events $(t', l') \in \mathcal{E}(0, q)$ and sum up the response function values. The straightforward intensity evaluation method is easy to parallelise, but it results in a huge amount of redundant computation. The second method is a sequential approach based on the relations between the current intensity and a previously computed intensity. The sequential intensity evaluation method avoids the redundant computation, but it is difficult to parallelise and pipeline due to the serial data dependency.

The redundant computation in the first method and the serial dependency pattern in the second method are two major problems for efficient reconfigurable computing. To the best of our knowledge, there exist no methods that solve the two problems together.

We believe that the root of the two problems is the pursuit of exact values of intensities. In other words, it might be possible to design efficient hardware to compute the approximate intensities rather than the exact ones. In this study, we propose to approximate the intensity function of the Hawkes point process using a collection of step functions. We start by analysing and approximating the intensity function of the univariate Hawkes process. We then generalise the method to the multivariate Hawkes process.

Let $\mathcal{E}(0, t)$ be an event sequence produced by a univariate Hawkes process $\theta = \langle \lambda^\perp, \alpha, \beta \rangle$. The intensity function at time t is

$$\lambda(t) = \sum_{i=0}^{|\mathcal{E}(0,t)|-1} g(t - t_i) + \lambda^\perp \quad (4.1)$$

$$= \sum_{i=0}^{|\mathcal{E}(0,t)|-1} \alpha e^{-\beta(t-t_i)} + \lambda^\perp \quad (4.2)$$

$$= \sum_{i=0}^{|\mathcal{E}(0,t)|-1} \alpha e^{\beta(t_i-t)} + \lambda^\perp \quad (4.3)$$

To approximate the intensity function, we build a function $\tilde{\lambda}(t)$ such that

$$\text{abs}(\tilde{\lambda}(t) - \lambda(t)) < \delta \quad (4.4)$$

where δ is a positive real number defining the error bound of the approximation. In this study,

we let $\tilde{\lambda}(t)$ take a similar form to $\lambda(t)$. In particular, we use

$$\tilde{\lambda}(t) = \sum_{i=0}^{|\mathcal{E}(0,t)|-1} \alpha \nu(t_i - t) + \lambda^\perp \quad (4.5)$$

where ν is a function that we need to design. Using Equation 4.5, we may simplify Inequality 4.4 as

$$\text{abs}\left(\sum_{i=0}^{|\mathcal{E}(0,t)|-1} \alpha \nu(t_i - t) - \sum_{i=0}^{|\mathcal{E}(0,t)|-1} \alpha e^{\beta(t_i - t)}\right) < \delta \quad (4.6)$$

The simplification removes the background intensity $\lambda^\perp$. Therefore, we may design the function ν without considering the background intensity. By simplifying Equation 4.6, we have

$$\text{abs}\left(\alpha \sum_{i=0}^{|\mathcal{E}(0,t)|-1} (\nu(t_i - t) - e^{\beta(t_i - t)})\right) < \delta \quad (4.7)$$

By the definition of the Hawkes self-exciting processes, α is a positive real number. Therefore, we divide both sides by α without changing the direction of the inequality, i.e.

$$\text{abs}\left(\sum_{i=0}^{|\mathcal{E}(0,t)|-1} (\nu(t_i - t) - e^{\beta(t_i - t)})\right) < \frac{\delta}{\alpha} \quad (4.8)$$

We may secure the validity of Inequality 4.8 by finding a function ν such that

$$\text{abs}(\nu(t_i - t) - e^{\beta(t_i - t)}) < \frac{\delta}{\alpha |\mathcal{E}(0, t)|} \quad (4.9)$$

In other words, Inequality 4.9 is a sufficient condition for error control. When Inequality 4.9 holds, the absolute difference between the exact intensity function value $\lambda(t)$ and its approximate value $\tilde{\lambda}(t)$ must be less than δ .

We generalise the above analysis to the multivariate Hawkes process and find a sufficient condition for error control. Let $\mathcal{E}(0, t)$ be an event sequence taken from an M -dimensional

Hawkes process $\theta = \langle \lambda^\perp, A, B \rangle$. The intensity function for event type m is

$$\lambda_m(t) = \sum_{m'=0}^{M-1} \sum_{i=0}^{|\mathcal{E}_{m'}(0,t)|-1} g_{m,m'}(t - t_{m',i}) + \lambda_m^\perp \quad (4.10)$$

$$= \sum_{m'=0}^{M-1} \alpha_{m,m'} \sum_{i=0}^{|\mathcal{E}_{m'}(0,t)|-1} e^{-\beta_{m,m'}(t-t_{m',i})} + \lambda_m^\perp \quad (4.11)$$

$$= \sum_{m'=0}^{M-1} \alpha_{m,m'} \sum_{i=0}^{|\mathcal{E}_{m'}(0,t)|-1} e^{\beta_{m,m'}(t_{m',i}-t)} + \lambda_m^\perp \quad (4.12)$$

Similar to the treatment for the univariate Hawkes process in Inequality 4.4, we build a function $\tilde{\lambda}_m(t)$ such that

$$\text{abs}(\tilde{\lambda}_m(t) - \lambda_m(t)) < \delta_m \quad (4.13)$$

where δ_m is the error tolerance for event type m . Let $\tilde{\lambda}(t)$ take a similar form to $\lambda_m(t)$, i.e.

$$\tilde{\lambda}_m(t) = \sum_{m'=0}^{M-1} \alpha_{m,m'} \sum_{i=0}^{|\mathcal{E}_{m'}(0,t)|-1} \nu_{m,m'}(t_{m',i} - t) + \lambda_m^\perp \quad (4.14)$$

Therefore, the error constrain simplifies to

$$\begin{aligned} & \text{abs}\left(\sum_{m'=0}^{M-1} \alpha_{m,m'} \sum_{i=0}^{|\mathcal{E}_{m'}(0,t)|-1} \nu_{m,m'}(t_{m',i} - t) \right. \\ & \quad \left. - \sum_{m'=0}^{M-1} \alpha_{m,m'} \sum_{i=0}^{|\mathcal{E}_{m'}(0,t)|-1} e^{\beta_{m,m'}(t_{m',i}-t)} \right) < \delta_m \end{aligned} \quad (4.15)$$

which is equivalent to

$$\text{abs}\left(\sum_{m'=0}^{M-1} \alpha_{m,m'} \sum_{i=0}^{|\mathcal{E}_{m'}(0,t)|-1} (\nu_{m,m'}(t_{m',i} - t) - e^{\beta_{m,m'}(t_{m',i}-t)}) \right) < \delta_m \quad (4.16)$$

This form is more complex than its counterpart for the univariate case in Inequality 4.7, because it is impossible to directly divide both sides by a parameter-dependent constant.

However, we may apply the triangle inequality of absolute values to Inequality 4.7 and obtain

$$\begin{aligned} & \text{abs}\left(\sum_{m'=0}^{M-1} \alpha_{m,m'} \sum_{i=0}^{|\mathcal{E}_{m'}(0,t)|-1} (\nu_{m,m'}(t_{m',i}-t) - e^{\beta_{m,m'}(t_{m',i}-t)})\right) \\ & \leq \sum_{m'=0}^{M-1} \alpha_{m,m'} \sum_{i=0}^{|\mathcal{E}_{m'}(0,t)|-1} \text{abs}(\nu_{m,m'}(t_{m',i}-t) - e^{\beta_{m,m'}(t_{m',i}-t)}) \end{aligned} \quad (4.17)$$

As a result, a sufficient condition to ensure the validity of Inequality 4.16 is that the right-hand side of Inequality 4.17 is less than δ_m . Let

$$\rho_m = \max_{m' \in [0..(M-1)]} \alpha_{m,m'} \quad (4.18)$$

The right hand side of the Inequality 4.17 must satisfy

$$\begin{aligned} & \sum_{m'=0}^{M-1} \alpha_{m,m'} \sum_{i=0}^{|\mathcal{E}_{m'}(0,t)|-1} \text{abs}(\nu_{m,m'}(t_{m',i}-t) - e^{\beta_{m,m'}(t_{m',i}-t)}) \\ & \leq \rho_m \sum_{m'=0}^{M-1} \sum_{i=0}^{|\mathcal{E}_{m'}(0,t)|-1} \text{abs}(\nu_{m,m'}(t_{m',i}-t) - e^{\beta_{m,m'}(t_{m',i}-t)}) \end{aligned} \quad (4.19)$$

As long as we can keep the right-hand side of Inequality 4.19 to be less than δ_m , we can guarantee the validity of Inequality 4.17 and 4.16. The nested summation in the right-hand side of Inequality 4.19 is the accumulated error of the function $\nu_{m,m'}$ for all events in the sequence. As a result, a sufficient condition to make the right-hand side less than δ_m is

$$\text{abs}(\nu_{m,m'}(t_i - t) - e^{\beta_{m,m'}(t_i - t)}) < \frac{\delta_m}{\rho_m |\mathcal{E}(0, t)|} \quad (4.20)$$

for all $(m, m') \in [0..(M-1)]^2$. The error control condition for the univariate Hawkes process in Inequality 4.9 is a special case of Inequality 4.20, because we have $\rho_m = \alpha$ and $\delta_m = \delta$ for a univariate Hawkes process.

4.1.1 Step Function Approximation

We design a function $\nu_{m,m'}$ that satisfies the error control condition in Inequality 4.20. It is expected that the function $\nu_{m,m'}$ approximates the exponential part of the response function. We propose to build $\nu_{m,m'}$ as a step function, because it is unnecessary to use adders or multipliers to evaluate the function. As the value of the exponential function $\exp(x)$ decreases towards zero as x decreases to $-\infty$, we treat the function values as zero when x is sufficiently small. Therefore, we may build $\nu_{m,m'}(x)$ as a step function with a finite number of intervals.

We divide the domain of the step function into K intervals separated by an ascending

sequence of $(K + 1)$ separators

$$\mathbf{b} = (b_0, b_1, b_2, \dots, b_K) \quad (4.21)$$

where $b_k < b_{k+1}$ for all $k \in [0..(K - 1)]$ and $b_0 = 0$. Given the current time t , each pair of adjacent separators defines an interval

$$\mu_k(t) = \{t' : t - b_{k+1} \leq t' < t - b_k\} \quad (4.22)$$

Given a pair of event type labels (m, m') , we use a collection of K real numbers to define the function values for the K intervals. As a result, the approximate function becomes

$$\nu_{m,m'}(x) = \sum_{k=0}^{K-1} h_{m,m',k} \chi_k(x) \quad (4.23)$$

where $\chi_k(x)$ is an indicator function defined by

$$\chi_k(x) = \begin{cases} 1 & -b_k \leq x < -b_{k+1} \\ 0 & \text{otherwise} \end{cases} \quad (4.24)$$

By Equation 4.14 and 4.23, one may write the intensity function using the approximate exponential function ν as follows:

$$\tilde{\lambda}_m(t) = \sum_{m'=0}^{M-1} \alpha_{m,m'} \sum_{i=0}^{|\mathcal{E}_{m'}(0,t)|-1} \sum_{k=0}^{K-1} h_{m,m',k} \chi_k(t_{m',i} - t) + \lambda_m^\perp \quad (4.25)$$

Therefore, it is possible to swap the two inner summation operators as they are independent of each other, i.e.

$$\tilde{\lambda}_m(t) = \sum_{m'=0}^{M-1} \alpha_{m,m'} \sum_{k=0}^{K-1} \sum_{i=0}^{|\mathcal{E}_{m'}(0,t)|-1} h_{m,m',k} \chi_k(t_{m',i} - t) + \lambda_m^\perp \quad (4.26)$$

Moreover, $h_{m,m',k}$ is independent of the summation index i is independent, it is possible to move $h_{m,m',k}$ out of the innermost summation operator, i.e.

$$\tilde{\lambda}_m(t) = \sum_{m'=0}^{M-1} \alpha_{m,m'} \sum_{k=0}^{K-1} h_{m,m',k} \sum_{i=0}^{|\mathcal{E}_{m'}(0,t)|-1} \chi_k(t_{m',i} - t) + \lambda_m^\perp \quad (4.27)$$

Table 4.1: Comparison of three intensity evaluation methods

(Column 2–4 respectively correspond to the three evaluation methods)			
Property	Straightforward	Sequential	Approximate
Exact result	Yes	Yes	No
Requirement of exact offsets	Yes	Yes	No
Requirement of full history	Yes	No	No
Requirement of extra storage space	No	Yes	Yes

By the definition of the indicator function $\chi_k(x)$, we have

$$\sum_{i=0}^{|\mathcal{E}_{m'}(0,t)|-1} \chi_k(t_{m',i} - t) = \sum_{i=0}^{|\mathcal{E}_{m'}(t-b_k, t-b_{k-1})|-1} 1 \quad (4.28)$$

$$= |\mathcal{E}_{m'}(t - b_k, t - b_{k-1})| \quad (4.29)$$

Finally, the approximate intensity function becomes

$$\tilde{\lambda}_m(t) = \sum_{m'=0}^{M-1} \alpha_{m,m'} \sum_{k=0}^{K-1} h_{m,m',k} |\mathcal{E}_{m'}(t - b_k, t - b_{k-1})| + \lambda_m^\perp \quad (4.30)$$

We may further simplify the expression using the dot product operator, i.e.

$$\tilde{\lambda}_m(t) = \sum_{m'=0}^{M-1} \alpha_{m,m'} (\mathbf{h}_{m,m'} \cdot \mathbf{c}_{m'}(t, \mathbf{b})) + \lambda_m^\perp \quad (4.31)$$

where $\mathbf{c}_{m'}(t, \mathbf{b})$ is a column vector of K numbers defined by

$$\mathbf{c}_{m'}(t, \mathbf{b}) = \begin{pmatrix} |\mathcal{E}_{m'}(t - b_1, t - b_0)| \\ |\mathcal{E}_{m'}(t - b_2, t - b_1)| \\ \vdots \\ |\mathcal{E}_{m'}(t - b_K, t - b_{K-1})| \end{pmatrix} \quad (4.32)$$

We name $\mathbf{c}_{m'}(t, \mathbf{b})$ the *count vector* of event type m' at time t with respect to $\mathbf{b}$. Moreover, we call $\{t' : t - b_{k+1} < t' \leq t - b_k\}$ be the *count interval* bounded by b_{k+1} and b_k . For ease of discussion, let $\mathbf{c}(t, \mathbf{b})$ be the concatenation of $\mathbf{c}_0(t, \mathbf{b}) \dots \mathbf{c}_{M-1}(t, \mathbf{b})$.

This approximation makes it possible to avoid the redundant computation and the sequential dependencies in the two evaluation methods for exact intensities. Table 4.1 summaries the major differences. Different from the straightforward intensity evaluation, the proposed method does not track back to visit the full history, as long as the count vectors are available.

Different from the sequential method, the proposed method stores and updates event counts instead of the information about the previous intensities.

4.1.2 Separators and Step Heights

The evaluation of the approximate intensity function in Equation 4.31 requires two parameters. The first parameter is a sequence of separators $\mathbf{b}$. The second parameter is a set of height vectors $\mathbf{h}_{m,m'}$ for all $(m, m') \in [0..(M-1)]^2$. In this section, we propose a method to compute a sequence of separators and step heights using the parameter set of a multivariate Hawkes point process.

We first make Equation 4.20 independent of the time t and the event type m by specifying a unified error bound for the maximum time T for all dimensions $m \in [0..(M-1)]$. Let

$$\Xi = \frac{\delta_m}{\rho_m |\mathcal{E}(0, T)|} \quad (4.33)$$

for all $m \in [0..(M-1)]$. A sufficient condition to secure Inequality 4.20 is

$$|\nu_{m,m'}(t_i - t) - e^{\beta_{m,m'}(t_i - t)}| < \Xi \quad (4.34)$$

Plugging Equation 4.23 into equation 4.34, we have

$$\left| \sum_{k=0}^{K-1} h_{m,m',k} \chi_k(x) - e^{\beta_{m,m'}(t_i - t)} \right| < \Xi \quad (4.35)$$

We try to find a sufficient condition about $h_{m,m',k}$ to make 4.35 constantly hold. Let $\mathbf{b}_{m,m'} = (b_{m,m',0}, b_{m,m',1}, b_{m,m',2}, \dots, b_{m,m',K})$ be a sequence of separators for a pair of event types (m, m') . We set $h_{m,m',k}$ to be a value of the exponential function in the count interval bounded by $b_{m,m',k+1}$ and $b_{m,m',k}$, i.e.

$$e^{-\beta_{m,m'} b_{m,m',k+1}} \leq h_{m,m',k} < e^{-\beta_{m,m'} b_{m,m',k}} \quad (4.36)$$

The exponential function is a monotonically increasing. Therefore, when a data point falls in the interval $\mu_k(t)$, the maximum error between the perfect exponential function value and the approximate value must be no more than the range of the exponential function in the interval $-b_{m,m',k+1} \leq x \leq -b_{m,m',k}$. In other words, the following inequality must hold:

$$\text{abs}\left(\sum_{k=0}^{K-1} h_{m,m',k} \chi_k(x) - e^{\beta_{m,m'}(t_i - t)}\right) < e^{-\beta_{m,m'} b_{m,m',k}} - e^{-\beta_{m,m'} b_{m,m',k+1}} \quad (4.37)$$

As a result, we enforce

$$e^{-\beta_{m,m'}b_{m,m',k}} - e^{-\beta_{m,m'}b_{m,m',k+1}} \leq \Xi \quad (4.38)$$

for all $k \in [0..(K-1)]$. Then

$$e^{-\beta_{m,m'}b_{m,m',k+1}} \geq e^{-\beta_{m,m'}b_{m,m',k}} - \Xi \quad (4.39)$$

Take the logarithm of both sides, we have

$$b_{m,m',k} \leq -\frac{1}{\beta_{m,m'}} \ln(e^{-\beta_{m,m'}b_{m,m',k-1}} - \Xi) \quad (4.40)$$

We want the margin to be as large as possible as long as the Inequality 4.40 holds, because larger margin results in smaller K . As a result, we take the maximum $b_{m,m',k}$, i.e.

$$b_{m,m',k+1} = -\frac{1}{\beta_{m,m'}} \ln(e^{-\beta_{m,m'}b_{m,m',k}} - \Xi) \quad (4.41)$$

We may make the expression non-recursive, i.e.

$$e^{-\beta_{m,m'}b_{m,m',k}} - e^{-\beta_{m,m'}b_{m,m',k+1}} = \Xi \quad (4.42)$$

In other words, $\{e^{-\beta_{m,m'}b_{m,m',k}}\}_k$ is an arithmetic progression with the general formula

$$e^{-\beta_{m,m'}b_{m,m',k}} = 1 - k\Xi \quad (4.43)$$

When the right-hand side is greater than or equal to zero, we take logarithm to solve $b_{m,m',k}$

$$b_{m,m',k} = -\frac{1}{\beta_{m,m'}} \ln(1 - k\Xi) \quad (4.44)$$

We use a unified collection of separators to control the complexity of the model. As smaller intervals lead to higher precision for the approximations, we take the minimum location offset for all separators. Equation 4.44 shows that $b_{m,m',k}$ decreases as $\beta_{m,m'}$ increases. Therefore, given the Hawkes process parameter $\theta = (A, B, \Lambda)$, we pick the largest entry from B to calculate b_k , i.e.

$$b_k = -\frac{1}{\beta_{\max}} \ln(1 - k\Xi) \quad (4.45)$$

where

$$\beta_{\max} = \max_{(m,m') \in [0..(M-1)]^2} \beta_{m,m'} \quad (4.46)$$

We then find the number of count vectors K . To make Equation 4.45 valid, we need to keep the input of the logarithmic function positive, i.e.

$$1 - k\Xi > 0 \quad (4.47)$$

This inequality is equivalent to

$$k < \frac{1}{\Xi} \quad (4.48)$$

In other words, when k satisfy the above inequality, it is possible to produce a separator b_k using Equation 4.45. We produce all possible separators to maximise the precision of the approximation. Therefore, we set the number of separators to the maximum integer solution of the above inequality, i.e.

$$K = \lceil \frac{1}{\Xi} \rceil - 1 \quad (4.49)$$

When the separator sequence $\mathbf{b}$ is available, we compute the step heights $\mathbf{h}_{m,m',k}$ for $(m, m', k) \in [0..(M-1)] \times [0..(M-1)] \times [1..K]$ for the approximate intensity function. Given a pair of dimensions $(m, m') \in [0..(M-1)] \times [0..(M-1)]$, we compute a vector $\mathbf{h}_{m,m'}$ containing K entries. A reasonable way to set $\mathbf{h}_{m,m',k}$ is to take a value of the function $f(x) = e^{\beta_{m,m'}x}$ in the interval $(-b_{k+1}, -b_k)$. In this study, we take the value at the middle point of the two consecutive separators, i.e.

$$\mathbf{h}_{m,m',k} = e^{-\frac{1}{2}\beta_{m,m'}(b_{k+1}+b_k)} \quad (4.50)$$

4.1.3 Algorithms for Approximate Intensity Functions

Based on the above discussion, we design an algorithm to compute the parameters for the approximate intensity function. The input of the algorithm includes (i) a Hawkes point process parameter set $\theta = (A, B, \Lambda)$ and (ii) the number of separators K . The output of the algorithm includes (i) a separator sequence $\mathbf{b}$; (ii) step height vectors $\mathbf{h}_{m,m'}$ for all $(m, m') \in [0..(M-1)]^2$.

Algorithm 2 gives the detailed steps of the initialisation process of the approximate intensity function. We show the correctness of the algorithm by specifying the correspondence between the lines and the related definitions. Line 1 computes $\beta_{\max}$ using Equation 4.46. Line 2–6 compute the separators $\mathbf{b} = (0, b_1, \dots, b_K)$ using Equation 4.45. Line 7–14 compute the heights $\mathbf{h}_{m,m'} = (\mathbf{h}_{m,m',0}, \dots, \mathbf{h}_{m,m',K-1})$ using Equation 4.50. The worst-case

Algorithm 2: Initialisation of approximate intensity function

Input: $\theta = (A, B, \Lambda); K$
Output: $\mathbf{b}; \mathbf{h}_{m,m'}$ for all $(m, m') \in [0..(M-1)]^2$

```
1  $\beta_{\max} \leftarrow \max_{(m,m') \in [1..M]^2} \beta_{m,m'}$ 
2  $b_0 \leftarrow 0$ 
3 for  $k \in [1..K]$  do
4    $b_k \leftarrow -\frac{1}{\beta_{\max}} \ln(1 - k\Xi)$ 
5 end
6  $\mathbf{b} \leftarrow (b_0, b_1, b_2, \dots, b_K)$ 
7 for  $m \in [0..(M-1)]$  do
8   for  $m' \in [0..(M-1)]$  do
9     for  $k \in [0..K-1]$  do
10       $h_{m,m',k} \leftarrow e^{-\frac{1}{2}\beta_{m,m'}(b_{k+1}+b_k)}$ 
11    end
12     $\mathbf{h}_{m,m'} \leftarrow (h_{m,m',0}, h_{m,m',1}, \dots, h_{m,m',K-1})$ 
13  end
14 end
```

Algorithm 3: Approximate intensity evaluation

Input: $\mathbf{b}; \mathbf{h}_{m,m'}$ for all $(m, m') \in [0..(M-1)]^2$; t
Output: λ_m for all $m \in [0..(M-1)]$

```
1 for  $m \in [0..(M-1)]$  do
2    $\lambda_m(t) \leftarrow \lambda_m^\perp$ 
3   for  $m' \in [0..(M-1)]$  do
4      $\lambda_m(t) \leftarrow \lambda_m(t) + \alpha_{m,m'}(\mathbf{c}_{m'}(t, \mathbf{b}) \cdot \mathbf{h}_{m,m'})$ 
5   end
6 end
```

time complexity of the algorithm is $O(M^2K)$. In particular, the calculation of $\beta_{\max}$ takes $O(M^2)$ time; the calculation of $\mathbf{b}$ also takes $O(M^2)$ time; the calculation of $\mathbf{h}_{m,m'}$ for all $(m, m') \in [0..(M-1)]^2$ takes $O(M^2K)$ time in total.

When the separators and the step height vectors become available, we may compute the intensity vector $\lambda(t)$ at any time t using Algorithm 3. The proof of correctness for the algorithm is trivial because it is a straightforward computational procedure for Equation 4.31. The worst-case time complexity for the algorithm is $O(M^2K)$.

We provide an example for the proposed approximate function. The parameter set of the

Hawkes process we use in this example is as follows:

$$A = \begin{pmatrix} 0.30 & 0.04 & 0.05 \\ 0.03 & 0.30 & 0.02 \\ 0.10 & 0.00 & 0.25 \end{pmatrix} \quad (4.51)$$

$$B = \begin{pmatrix} 0.80 & 0.80 & 0.80 \\ 0.80 & 0.75 & 0.90 \\ 0.80 & 0.80 & 0.65 \end{pmatrix} \quad (4.52)$$

$$\Lambda^\perp = \begin{pmatrix} 0.10 & 0.05 & 0.12 \end{pmatrix}^\top \quad (4.53)$$

We plot the exact intensities along with the approximate ones for $K = 4, 8, 16, 32, 64, 128$ in Figure 4.1. The three components of an intensity vector are in blue, red and green respectively. Solid lines stand for exact intensities, while dotted lines stand for approximate intensities. The approximate intensity function appears as a step function of time as we expect. We may obtain the following three other observations from Figures 4.1.

1. The step widths of the approximate function are irregular. The reason behind the irregularity is that the change rate of the count vector is data-dependent. Specifically, the value of the approximate intensity function changes with the count vector. As a result, when the count vector changes frequently in a time interval, the approximate intensity function tends to have shorter step widths.
2. A larger value of K results in more accurate approximation. In particular, a larger K corresponds to a smaller value of Ξ . The small value of Ξ enforces a better control of the error. Let $\Xi_{\text{small}} < \Xi_{\text{large}}$. The offset of the farthest separator with Ξ_{small} is larger than the offset with Ξ_{large} . Moreover, the distribution of separators with Ξ_{small} are denser than that of Ξ_{large} .
3. A small set of intervals may provide a satisfactory approximate function. In a Hawkes process, the intensity vector at any time t depends on the full history. As a result, the approximation with a finite number of separators may accumulate the error caused by the historical point whose offsets are beyond the coverage of all separators. Nevertheless, the approximate function does not show a visible sign of error accumulation, as the exponential decay of the response function makes the events in the early history far less important than the events in the recent history.

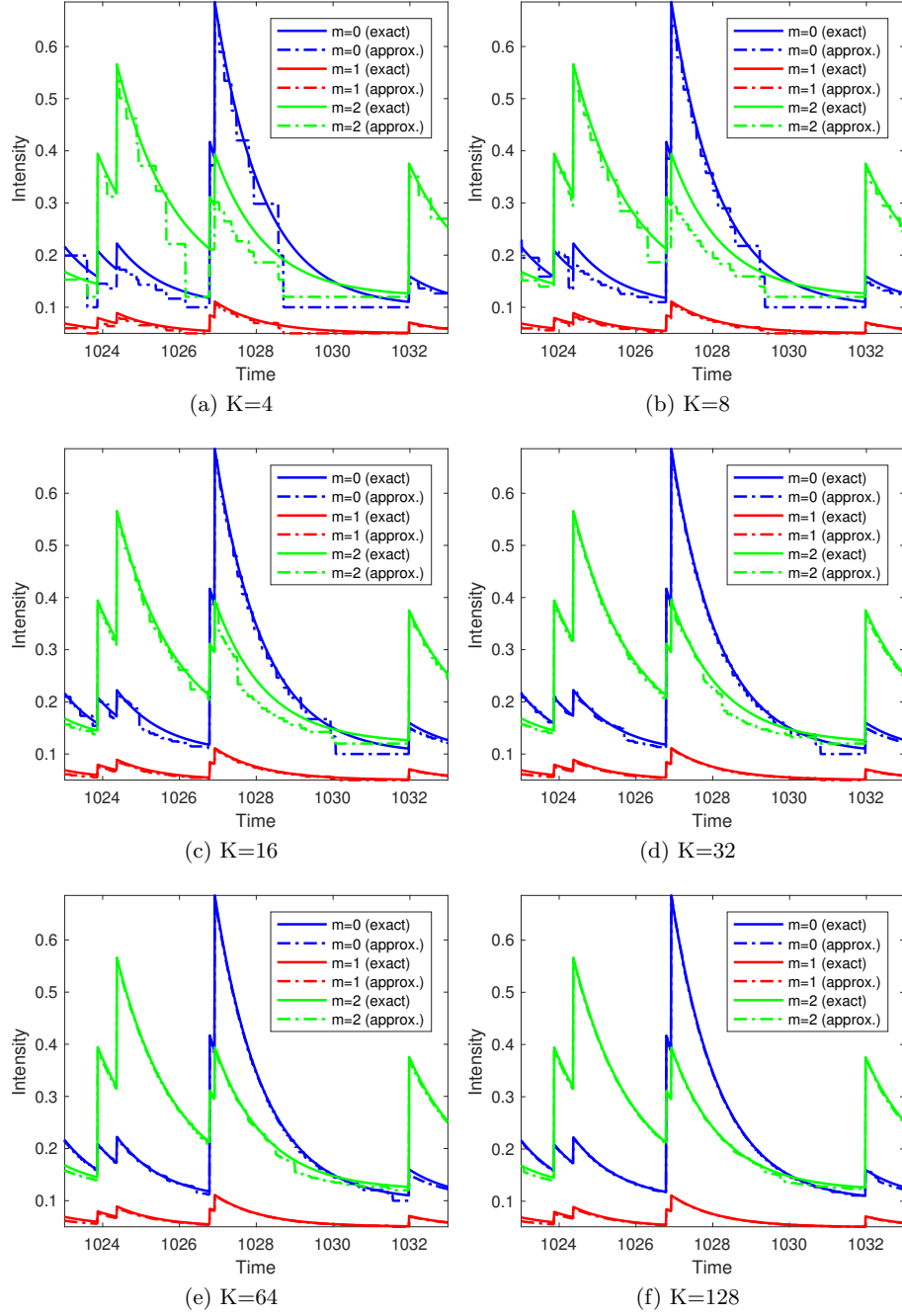


Figure 4.1: Exact and approximate intensities

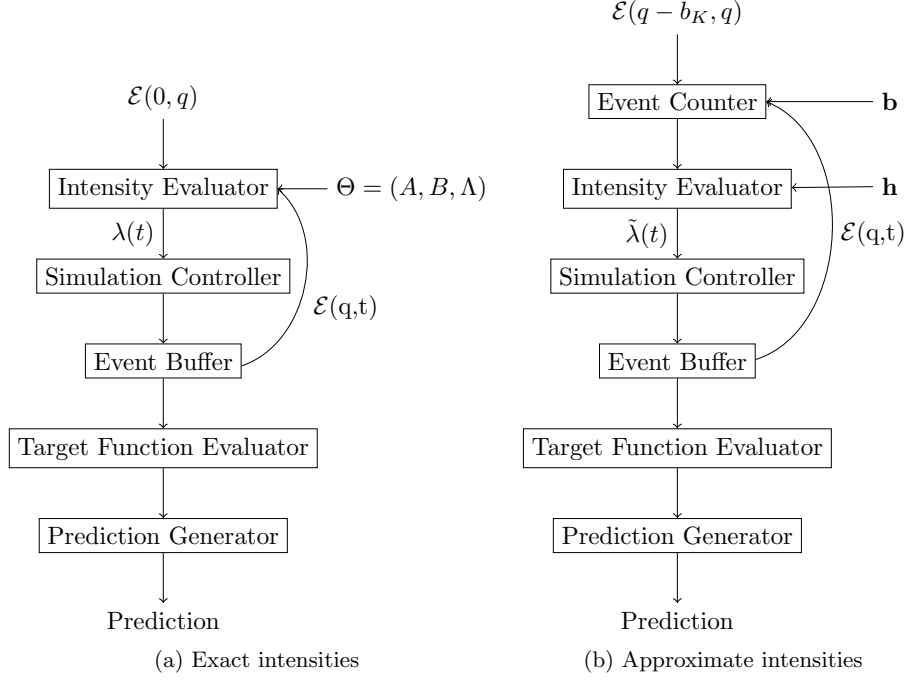


Figure 4.2: Prediction by simulation

4.2 Event Stream Prediction with Regression

The parameter estimation of the Hawkes point process models is computationally expensive. Therefore, fitting a predictive system based on the Hawkes process may be time consuming. In this section, we propose to use the count vectors and regression techniques to make predictions for event streams. The fitting algorithm for the proposed model, as shown in Algorithm 4 does not require the estimation of point process models. Moreover, it is possible to design reconfigurable hardware to further accelerate the model fitting process.

4.2.1 Elimination of Point Process Models

The conventional way to make predictions with point process models is to perform simulations. The underlying idea of the simulation-based method is straightforward. The simulator generates N_S different futures situations. It is then possible to make a prediction by averaging the target values in the N_S future situations.

We analyse the calculations involved in the prediction process to reveal the potential problems. Figure 4.2a shows the workflow of a simulation-based predictive system for event streams. The system simulates an intensity-based point process model to generate future situations. We omit the internal control signals in the simulation process for brevity. The

predictive system generates a prediction using the following four pieces of information:

1. The query time q .
2. The set of all events $\mathcal{E}(0, q)$.
3. The parameter set of the point process model θ .
4. The number of repetitions of the simulation N_S .

The internal structure of the predictive system consists of an *intensity evaluator*, a *simulator*, a *target function evaluator*, and a *prediction generator*. These parts operate as follows:

- The intensity evaluator computes the exact intensities at time t for the event sequence $\mathcal{E}(0, t)$. The two input signals of the intensity evaluator are the events in the actual history $\mathcal{E}(0, q)$ and the simulated events $\mathcal{E}(q, t)$. The evaluator obtains $\mathcal{E}(0, t)$ by concatenating $\mathcal{E}(0, q)$ and $\mathcal{E}(q, t)$.
- The simulator consists of a controller and an event buffer for simulated events. To generate a new event, the simulation controller passes the query time t and the event stream $\mathcal{E}(q, t)$ to the intensity evaluator. The intensity evaluator returns the intensity vector at the current time t . The simulation controller then decides whether to produce an event at time t .
- The target function evaluator computes and stores the target function value for simulated situations.
- The prediction generator combines the target values to produce a single prediction after all the N_S runs of simulation.

The simulation-based predictive system in Figure 4.2a results in a data dependency problem. The system needs to visit all historical events $\mathcal{E}(0, q)$ to make a prediction. As the approximate intensity function solves the data dependency problem in intensity evaluation, we replace the exact intensities by the approximate intensities. Figure 4.2b shows the modified predictive system with approximate intensities. The modified system takes two parameters $\mathbf{b}$ and $\mathbf{h}$. The two parameters control the approximate intensity function.

The approximate intensity evaluator solves the data dependency problem of the original system in Figure 4.2a by truncating the history. The modified predictive system in Figure 4.2b only relies on the recent history $\mathcal{E}(q - b_K, t)$ rather than the full history $\mathcal{E}(0, q)$. The approximate intensity function allows the truncated history to provide sufficient information to approximate the intensities in the simulation. To compute the intensity for time t , the evaluator only needs the historical event sequence $\mathcal{E}(t - b_K, t)$.

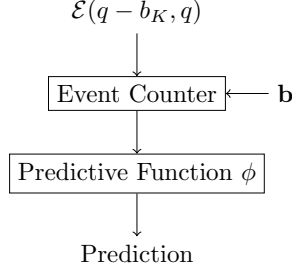


Figure 4.3: Elimination of simulation facilities

When we map the algorithms to a reconfigurable system, it is expected that the simulation controller generates intensity queries efficiently and keep the intensity evaluator busy. Nevertheless, the simulator is unable to generate intensity queries efficiently due to a data loop-back. Specifically, to decide whether to produce an event at time t , the simulator must finish the generation of all events in time interval $\{t' : q \leq t' < t\}$.

It is necessary to remove the data path from the event buffer to the intensity evaluator to improve the efficiency. Nevertheless, the predictive system without the data path is unable to operate. As a result, we propose to get rid of simulation. Specifically, we analyse the input and output of the simulation-based prediction method to find a way to replace the simulation process by a different prediction logic.

We may replace the simulator and all its downstream facilities by a predictive function that takes the intensity vector $\lambda(q)$ as the only input. However, the input of the predictive function may limit the predictive accuracy, because the intensity $\lambda(q)$ compresses the historical event stream with information loss. In particular, different historical event sequences may produce similar or identical intensities at time q , but the future situations after q may have different statistical properties.

We propose to find an enhanced predictive function by directly taking the count vector $\mathbf{c}(t, \mathbf{b})$ as an input, as shown in Figure 4.3. Compared to the original prediction process, the proposed prediction process no longer requires the intensity evaluator, the proposed model makes a prediction directly based on a count vector without calculating intensities.

From the perspective of statistics inference, the assumption of the Hawkes point process model with approximate intensities is that the count vector $\mathbf{c}(t, \mathbf{b})$ provides sufficient information to infer the intensities at time t . The assumption of our proposed model is a generalisation that $\mathbf{c}(t, \mathbf{b})$ provides sufficient information to directly infer the prediction target. In other words, we treat the data generator as a black box following the principles in [118].

Our predictive model for event streams includes two components. The first component is the set of separators $\mathbf{b}$. The second component is the predictive function ϕ . Before making predictions with a predictive system, it is necessary to fit the model by finding a proper

Algorithm 4: Fitting counting-based predictive model to data

Input: $R, \mathcal{E}(0, T), K$

Output: $\mathbf{b}^*, \phi^*$

```

1 for  $r \in [0..(R-1)]$  do
2    $\mathbf{b}_r \leftarrow \text{make-separators}(r, \mathcal{E}(0, T), K)$ 
3    $\mathcal{O} \leftarrow \text{make-regression-data}(\mathbf{b}_r, \mathcal{E}(0, T))$ 
4    $(\phi_r, J_r) \leftarrow \text{train-regression-model}(\mathcal{O})$ 
5 end
6  $r^* \leftarrow \arg \min_r (J_r)$ 
7  $\mathbf{b}^* \leftarrow \mathbf{b}_{r^*}$ 
8  $\phi^* \leftarrow \phi_{r^*}$ 

```

configuration of $\mathbf{b}$ and ϕ . We propose an algorithmic framework shown in Algorithm 4 for model fitting. The algorithm enumerates $\mathbf{b}$ and ϕ by trying R configurations of separators. Among the input arguments, the event sequence $\mathcal{E}(0, T)$ is the data; K is the number of count vectors. The correctness, effectiveness and complexity of the algorithmic framework depends on the design of the three subroutines namely **make-separators**, **make-regression-data** and **train-regression-model**. The definitions of the subroutines are as follows:

- **make-separators** is the subroutine to produce the r -th configuration of the separator sequence using the data. In this study, we use Equation 4.45 to produce separators.
- **make-regression-data** is the subroutine to generate training data for regression. Specifically, a data set is in the form

$$\begin{pmatrix} (\mathbf{c}(q_0, \mathbf{b}_r), \text{ptgt}(q_0)) \\ (\mathbf{c}(q_1, \mathbf{b}_r), \text{ptgt}(q_1)) \\ \vdots \\ (\mathbf{c}(q_{N_D-1}, \mathbf{b}_r), \text{ptgt}(q_{N_D-1})) \end{pmatrix} \quad (4.54)$$

- **train-regression-model** is the training algorithm of a regression model. It returns the regress function ϕ_r and the prediction error J_r on the training data.

4.2.2 Efficient Computation of Count Vectors

The count vector plays a key role in the proposed predictive model. Both the model fitting process and the prediction process need to query count vectors. In this subsection, we discuss the algorithms for the computation of count vectors. We first present and analyse a straightforward algorithm derived from the definition of count vectors. We then propose an algorithm with lower time complexity than the straightforward algorithm.

Algorithm 5: Backward scanning algorithm for count vectors

Input: $q, \mathbf{b}$ **Output:** $\mathbf{c}_m(q, \mathbf{b})$ for all $m \in [0..(M-1)]$

```
1  $i_{max} \leftarrow \arg \max_{i \in [0..(N-1)]} \{t_i : t_i < q\}$ 
2  $i \leftarrow i_{max}$ 
3  $k \leftarrow 0$ 
4 while  $i > 0$  and  $k < K$  do
5   while  $i > 0$  and  $q - t_i \leq b_k$  do
6      $m \leftarrow l_i$ 
7      $c_{m,k} \leftarrow c_{m,k} + 1$ 
8      $i \leftarrow i - 1$ 
9   end
10   $k \leftarrow k + 1$ 
11 end
12 for  $m \in [0..(M-1)]$  do
13    $\mathbf{c}_m(q, \mathbf{b}) \leftarrow (c_{m,0}, \dots, c_{m,K-1})^\top$ 
14 end
```

It is possible to design an algorithm to compute the count vector $\mathbf{c}_m(q, \mathbf{b})$ by its definition in Equation 4.32, as shown in Algorithm 5. The algorithm computes the count vectors using an event stream $\mathcal{E}(0, q)$, a query time q and a separator sequence $\mathbf{b}$.

The algorithm locates the latest event before q and scan backwards to count the events. Specifically, the first three lines initialise the algorithm. Line 1 finds out the latest event before q and stores the index of the event as i_{max} . Line 2–3 initialise two indices namely i and k . The index i maintains the index of the event that the algorithm is counting. The index k keeps the index of the count interval into which the i -th event falls. The nested loop in Line 4–11 controls the counting procedure.

The algorithm is correct because the index k -interval that contains the i -th event whenever $c_{m,k}$ updates. In particular, the nested loop in Line 4–11 keeps the correspondence between the events and indices as follows:

- The inner loop controls the visits to the events. In each iteration of the inner loop, the algorithm visits all the events, if there is any, in the k -th interval. When visiting an event (t_i, l_i) , the algorithm checks the event type l_i and adds the corresponding count by one. The inner loop terminates when the algorithm finishes visiting all the points in the k -th interval.
- The outer loop updates the interval index k according to i . When the inner loop terminates, if the event index i is greater than or equal to zero, it must point to an event in the $(k-1)$ -th interval. Therefore, the algorithm decreases the value of k by

one so that k points to the interval where event o_i locates. The outer loop terminates when either the event index i or the interval index k reaches zero.

The worst-case time complexity of the algorithm is $O(N)$ where N is the number of events in the sequence. The time spent on the algorithm consists of two parts namely initialisation time and the execution time of the nested loop.

- The initialisation time includes the time spent on Line 1–3. The calculation of i_{max} in the first line takes at most $O(N)$ time. It is possible to reduce the time complexity with additional assumptions. For instance, we may assume that the list holding the data offers an indexing operation in $O(1)$ time. In this case, it is possible to use a binary search algorithm to find i_{max} . Therefore, the time complexity reduces to $O(\log N)$. The initialisation operation in Line 2–3 are memory copy operations. The two operations altogether take $O(1)$ time.
- The nested loops take at most $O(N)$ time. Both the event index i and the interval index k decrease in a monotonic manner. The calculations in each run of the inner loop consume $O(1)$ time. In the worse case, the event index i decreases from i_{max} to zero. Therefore, the total time spent on all runs of the inner loop algorithm is $O(N)$.

As the time spent on one query is $O(N)$, the time spent on N_Q queries is $O(N \cdot N_Q)$. Therefore, the algorithm scales poorly with the length of the sequence because of the complexity, but it is impossible to further reduce this time complexity. In the worst case where all historical events are in the count interval, the algorithm must visit all points to ensure the correctness.

We consider it difficult to design an efficient pipelined dataflow engine to compute count vectors based on Algorithm 5. The binary search algorithm for i_{max} gives a low time complexity, but the algorithm is difficult to deploy in pipelined hardware as it requires random memory access frequently over a large memory space.

It is difficult to improve the straightforward algorithm for lower time complexity or higher pipeline friendliness. We believe that the key reason for the difficulty is that the straightforward algorithm processes each count vector query independently regardless the relations between the count vectors for different time points. Instead of processing the count vector queries individually, we compute the count vector for a time t_B based on the count vector for an earlier time t_A .

We consider the process of calculating the counter vector for t_B as a process of sliding the count intervals from t_A to t_B . At any time t during the sliding process, we maintain a collection of counters that keeps the number of events for each interval. In other words, the collection of counters keeps the count vector $\mathbf{c}(t, \mathbf{b})$ when the rightmost separator reaches t .

Algorithm 6: Counting by incremental updating

Input: $(t_0, l_0, k_0) \dots (t_{N_U-1}, l_{N_U-1}, k_{N_U-1})$; $Q = \{q_0, \dots, q_{N_U-2}\}$

Output: $\mathbf{c}_m(q_0) \dots \mathbf{c}_m(q_{N_U-2})$ for all $m \in [0..(M-1)]$

```

1 for  $(m, k) \in [0..(M-1)] \times [0..(K-1)]$  do
2   |  $c_{m,k} \leftarrow 0$ 
3 end
4 for  $i \in [0..(N_U-2)]$  do
5   | if  $k_i > 0$  then
6     |  $c_{l_i, k_i-1} \leftarrow c_{l_i, k_i-1} - 1$ 
7   | end
8   | if  $k_i < K$  then
9     |  $c_{l_i, k_i} \leftarrow c_{l_i, k_i} + 1$ 
10  | end
11  | for  $m \in [0..(M-1)]$  do
12    |  $\mathbf{c}_m(q, \mathbf{b}) \leftarrow (c_{m,0}, \dots, c_{m,K-1})^\top$ 
13  | end
14 end

```

We initialise the vector to $\mathbf{c}(t_A, \mathbf{b})$. When the rightmost separator reaches t_B , we harvest the count vector $\mathbf{c}(t_B, \mathbf{b})$.

The algorithm produces correct results because the count vector changes if and only if a separator swipes over an event. In particular, any combination of an event and a separator constitutes an update to the count vector. As the set of events and the set of separators are finite and deterministic, the set of updates is also finite and deterministic. The set of updates of the count vector is

$$U = \{(t, l, k) : (t - b_k, l) \in \mathcal{E}, k \in [0..K]\} \quad (4.55)$$

We may pre-compute all the updates by enumerating the combinations of events and separators. In particular, the update produced by the event $o_i = (t_i, l_i)$ and the separator b_k is

$$u_{i,k} = (t_i + b_k, l_i, k) \quad (4.56)$$

The set of updates U contains $N \cdot (K+1)$ elements in total. Therefore, the generation of the updates takes $O(NK)$ time. Once all the updates are available, it is possible to compute the count vector for any time t .

As the count vector stays unchanged during a pair of consecutive updates, we only need to calculate one count vector after each update. Specifically, let $(t_0, l_0, k_0) \dots (t_{N_U-1}, l_{N_U-1}, k_{N_U-1})$ be a sequence of updates. Let q_i be any time point in the interval $\{t' : t_i < t' \leq t_{i+1}\}$. We

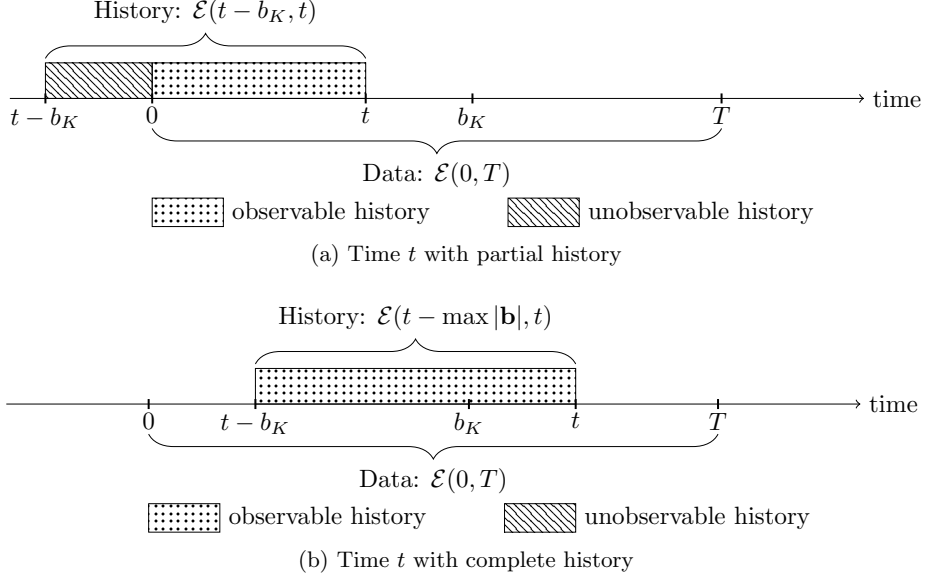


Figure 4.4: Completeness of history

may compute the count vectors at time q_i for all $i \in [0..(N_U - 2)]$ using Algorithm 6.

It is possible to map Algorithm 6 to a reconfigurable hardware block. We may use the on-chip memory to maintain the values of $c_{m,k}$ for all $(m, k) \in [0..(M - 1)] \times [0..(K - 1)]$. Once an update query for the count vector (t_i, l_i, k_i) arrives, the hardware block updates the value of $c_{l_i, k_i - 1}$ and c_{l_i, k_i} in the on-chip memory.

4.2.3 Data Preparation for Regression

The proposed event stream prediction model calls a regression model to make predictions. The training of this regression model requires a set of data. In this section, we propose a method to extract a data set for regression from the original event stream data.

Regarding the completeness of the history, the timeline $\{t' : 0 \leq t' < T\}$ consists of two intervals, as shown in Figure 4.4. A time point in the first interval $\{t' : 0 \leq t' < b_K\}$ only has a partial history. In contrast, a time point in the second interval $\{t' : b_K \leq t' < T\}$ has a complete history. As a result, we only collect data for regression in the second interval because the first interval provides underestimated counts.

Regarding the completeness of future, the timeline $\{t' : 0 \leq t' < T\}$ also consists of two intervals, as shown in Figure 4.5. It is possible to compute the perfect predictions for all time points in the first interval $\{t' : 0 < t' \leq T - \delta t\}$. However, it is impossible to compute the perfect prediction for any time point in the second interval $\{t' : T - \delta t < t' \leq T\}$. Therefore, we only collect data for regression from the first interval.

All the time points in the interval $D = \{t' : b_K \leq t' < T - \delta t\}$ have both a complete

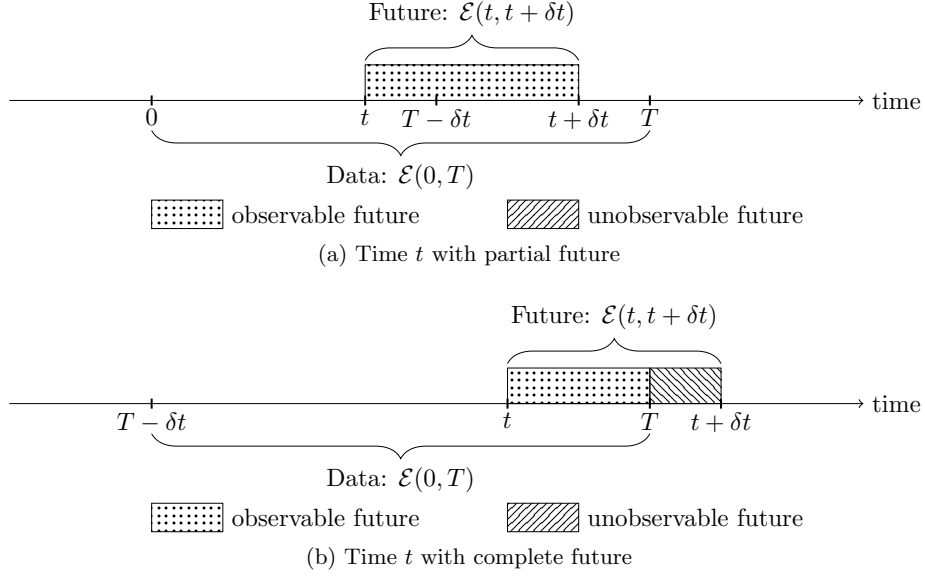


Figure 4.5: Completeness of future

history and a complete future. In other words, one may compute both a count vector and its perfect prediction for any time point in $\mathcal{D}$. We create a data point using the middle point q of each pair of consecutive updates in $\mathcal{D}$. Specifically, we use the count vector $\mathbf{c}(q, \mathbf{b})$ as the feature vector of the data point; we use the perfect prediction $\text{ptgt}(\mathcal{E}(q, q + \delta t))$ as the label of the data point. With these settings, we may generate the feature vectors of the data points using a software or hardware implementation of Algorithm 6.

4.2.4 Gradient Descent Engine for Linear Regression

We design a reconfigurable gradient descent engine to train linear regression models. One may feed the count vectors and prediction targets to the engine to train a linear regression model. Algorithm 1 shows the stochastic gradient descent algorithm that we map to hardware.

The stochastic gradient descent algorithm updates the linear coefficients iteratively. Each iteration contains three steps. The first step is to compute a prediction $\hat{y}_i$ using the current parameter $\mathbf{w}$ and a data point $o_i = (\mathbf{x}_i, y_i)$. The second step is to calculate the gradients using the difference between the prediction $\hat{y}_i$ and the actual label y_i . The third step is to update the parameter $\mathbf{w}$ using the gradients.

We map the three steps in an iteration to a hardware data path. Assume that we have a reconfigurable computing platform with sufficient resources for the deployment of all the calculations for the parameter update operation for a c -dimensional feature vector in a fully pipelined manner. We show a hardware data path in Figure 4.6 which involves three major hardware blocks.

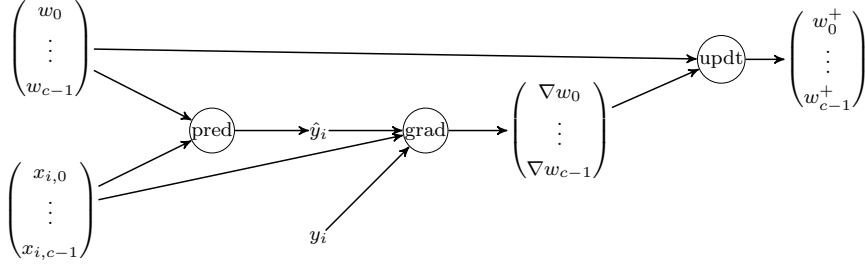


Figure 4.6: Data path for one iteration of SGD training

1. Prediction: the ‘pred’ block generates a prediction $\hat{y}_i$ using a data point and the current parameter $\mathbf{w}$. A fully-pipelined hardware block for this calculation may take c multipliers and c adders.
2. Gradient computation: the ‘grad’ block computes the gradients for all dimensions, which requires the prediction $\hat{y}_i$ and the actual target value y_i . The calculation consumes at least one adder to subtract the actual target value from the prediction. The linear regression model requires c multipliers for a fully-pipelined design.
3. Weight update: the ‘updt’ block updates the weights using the gradient information. A typical update of the parameter consists of two steps. The first step is to multiply the gradient vector by a small learning rate to construct a revision vector. The second step is to subtract the revision vector from the old parameter to produce the new parameter. A fully-pipelined hardware for the two steps take c multipliers and c adders.

The number of multipliers and adders in the above data path scales with the number of dimensions c . In particular, assume that all the calculations in the data path are with an identical data type. The data path may take as many as $3c$ multipliers and $(2c + 1)$ adders. As a consequence, the data path may consume a considerable amount of hardware resources when c is large.

When the dimensionality D is high enough, it is impossible to deploy the full data path on a reconfigurable computing platform with satisfactory clock frequency. In this case, it is necessary to break down the calculation into smaller pieces to meet the constraints of hardware resources. The critical calculation to break down is the dot product in the generation of prediction. It is relatively uncomplicated to retime the calculation into multiple cycles. Assume that we have a prediction block for the c -dimensional data, then it is possible to calculate the dot product of two D -dimensional vectors by invoking the hardware block for $\lceil D/c \rceil$ times and collect the results using an accumulator.

After loading all segments from data point $(\mathbf{x}_i, y_i)$, it is difficult to start processing the next data point $(\mathbf{x}_{i+1}, y_{i+1})$ immediately because the latest parameter $\mathbf{w}$ is not available due

to the latency of the pipeline. We solve this problem by taking an older version of parameter updated by $(\mathbf{x}_{i-1}, y_{i-1})$. In other words, we maintain two parameters. One parameter updates with the data points with odd indices. The other parameter updates with the data points with even indices. After streaming the data through the engine, we take the average vector of the two parameters.

4.3 Experimental Evaluation

In this section, we evaluate the proposed predictive model in an empirical way. We build two predictive systems based on the proposed model with different settings. We then compare the two systems with a system based on the Hawkes point process model.

4.3.1 Benchmarks

We propose five benchmark data sets consisting of four real-world data sets and a synthesised data set. We build the four real-world data sets namely ‘crime’, ‘earthquake’, ‘gbpusd’ and ‘nasahttp’ by converting and reformatting data from the public domain. In addition to the real-world data, we synthesis a data set ‘cantor’. Figure 4.7 shows a segment of each data set.

1. The ‘crime’ data set records the criminal cases in seven districts in Washington D.C. from 2011 to 2013 [119]. The data set contains 102952 events in seven types. Each event is the record of a crime case. The type label of an event marks the district ID of its occurrence.
2. The ‘earthquake’ data set marks earthquakes of different magnitudes in Northern California from 1984-2011 [120, 121, 122]. The data set contains 513278 events in four types. Each event is the record of an earthquake. The four event types correspond to different earthquake magnitudes.
3. The ‘gbpusd’ data set consists of the changes in the exchange rate between United States Dollars and and British Pounds on 30 June 2014. The data set contains 32608 events in four types. The four event types correspond to ‘bid price rise’, ‘bid price drop’, ‘ask price rise’ and ‘ask price drop’ respectively.
4. The ‘nasahttp’ data set is the log of the HTTP requests of different file types on the web servers of National Aeronautics and Space Administration (NASA) in July 1995 [123]. The data set contains 1450296 events in five types. Each event is an HTTP file query. The type label indicates the type of the queried file. The five event types respectively correspond to HTML files, JPEG images, GIF images, plain text files and other files.

5. The ‘cantor’ data set presents a Cantor set [124] in the time domain. The data set contains 18915 events in eight types. We design this benchmark to test the ability of the predictive models of capturing fractal and chaotic patterns.

4.3.2 Experimental Configuration

In the experiments, we build two predictive systems based on our proposed model. One system works with support vector regression (SVR). The other system works with linear regression (LR). We select the two regression models for two reasons.

1. Both regression models have efficient training solutions. The training of the regression model is a necessary part of model fitting. Therefore, we pay special attention to the efficiency of training. We have proposed a reconfigurable training engine for linear regression for our task. Although we have not proposed hardware engines to train support vectors regression models, there are fast software implementation such as LibSVM [125] and LaSVM [126].
2. Both regression models have uncomplicated prediction procedures. There are efficient tools for the prediction procedure for both regression methods. The prediction procedure of linear regression only requires dot product operations. The prediction procedure of SVR is relatively complex, but there are efficient hardware accelerators [127].

Table 4.2 summarises the detailed configurations of the three predictive systems used in the experiments. In the system with linear regression, we use a hardware implementation of Algorithm 6 for event counting and the SGD engine proposed in Section 4.2.4 for regression training. In addition to the two systems based on the proposed predictive model, we also build a predictive system based on the Hawkes point process model. In particular, we use the hardware accelerator proposed in Chapter 3 to accelerate the parameter estimation process. In the prediction stage, we simulate the Hawkes process for 2000 times for each test case and take the average result as the final prediction.

We use all the five benchmarks to test the predictive models. We solve an arrival prediction problem for each event type of each benchmark. For each benchmark, we evenly divide the timeline into two intervals, producing two event sequence $\mathcal{E}(0, \frac{T}{2})$ and $\mathcal{E}(\frac{T}{2}, T)$. We fit the predictive systems using $\mathcal{E}(0, \frac{T}{2})$ and test the systems using $\mathcal{E}(\frac{T}{2}, T)$. We set up a collection of $N_c = 1000$ checkpoints using

$$Q_c = \{q_i : q_i = \frac{T}{2}(1 + \frac{i}{N_c}), i \in [0..N_c - 1]\} \quad (4.57)$$

Each predictive system makes a prediction at each checkpoint for each event type. We calculate the mean absolute error using the predictions produced by the system and the perfect

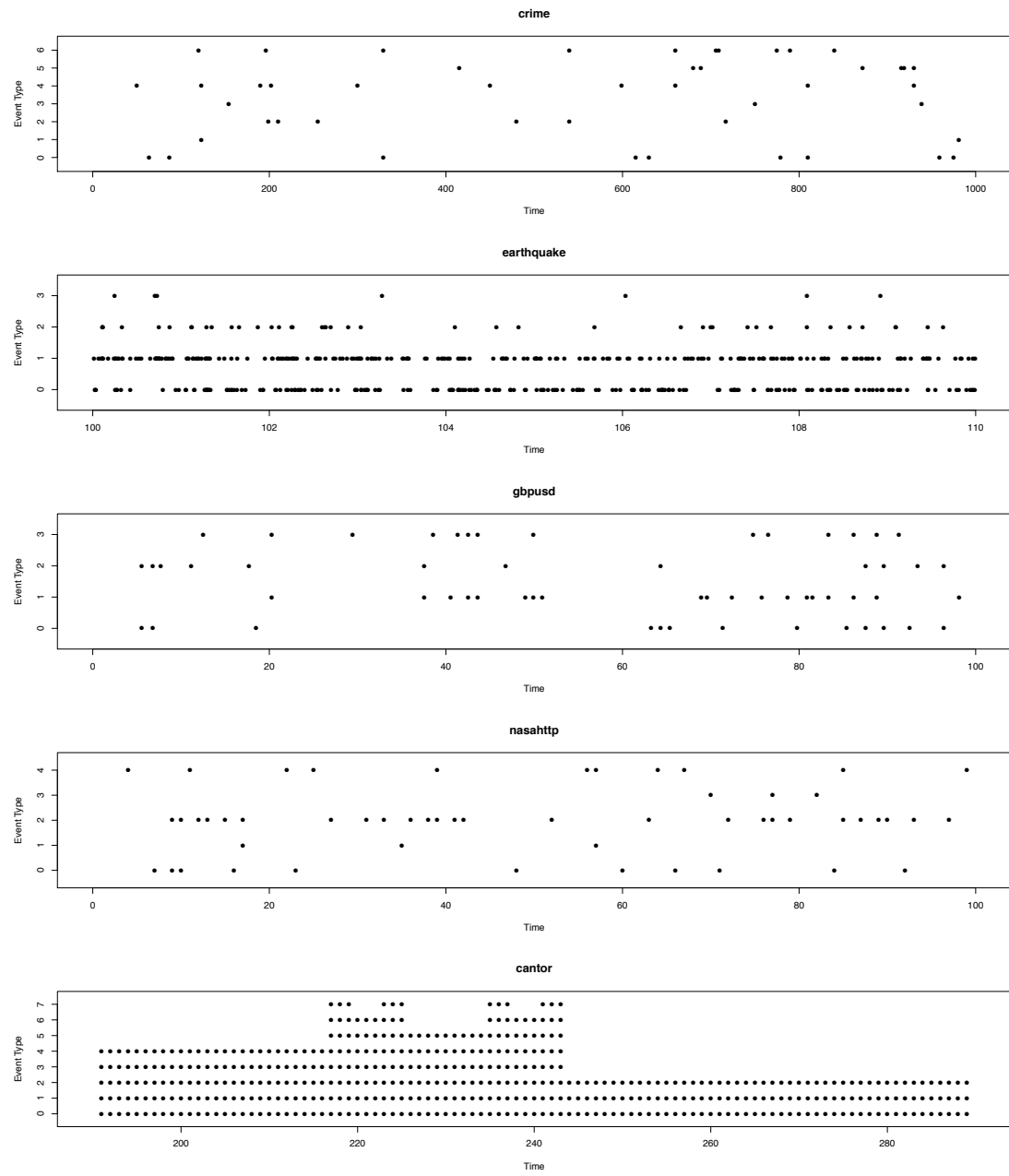


Figure 4.7: Segments taken from benchmarks

Table 4.2: Configuration of experimental predictive systems

Configuration	P-SVR (Software)	P-LR (Hardware)	Hawkes (Hardware)
Predictive model	Proposed	Proposed	Hawkes process
Data transformation	Counting	Counting	No transformation
Parameters for counting	K=100, R=100	K=100, R=100	N/A
Decision core	Support Vector Reg.	Linear Reg.	Simulation
Host CPU	Intel Xeon X5650 @ 2.67GHz		
No. CPU cores used	6	6	6
Host memory	48GB DDR3		
Host code language	C++	C	C
Host code compilation	Intel C/C++ compiler 12.1.4 with O3 optimisation		
Libraries	LibSVM 3.20 [125]	OpenMP 3.0	OpenMP 3.0
Regression training	LibSVM default	SGD	N/A
Acceleration engine	No acceleration	Maxeler MAX3 acceleration platform	
Number of FPGAs used	0	1	1
FPGA type	N/A	Xilinx Virtex-6 SX475T	
FPGA frequency	N/A	100MHz	120MHz
Maximum training time	3600 seconds		
Data sampling	10000 count vectors	Full training set	

predictions.

We record the computation efficiency by the execution time of training. In the two predictive systems with reconfigurable accelerators, the training time includes the data transmission overhead from the host computer to the acceleration platform. Also, we present the prediction accuracy of a Poisson process-based model as a baseline for accuracy.

4.3.3 Results and Discussion

We record respectively the experimental results for prediction error and computational efficiency in Tabel 4.3 and Tabel 4.4. In Tabel 4.3, all the numbers are mean absolute errors. Therefore, a smaller number stands for a more accurate result. In Tabel 4.4, the two ‘speedup’ columns respectively record the speedup of the two proposed systems over the accelerated system based on the Hawkes point process.

- Regarding the predictive accuracy, all the three predictive systems beat the baseline defined by the Poisson process. The two systems based on the proposed model are comparable with the system based on the Hawkes process model in predictive accuracy. Among the three systems, the proposed system with SVR is the most accurate in 26 out of 28 test cases in arrival prediction. The system based on linear regression is not as accurate as the system based on SVR, but it is still as accurate as the system based on the Hawkes point process.
- Regarding the computational efficiency, the proposed system with linear regression is

Table 4.3: Prediction error (mean absolute error)

Data	Type	P-SVR	P-LR	Hawkes	Poisson
crime	0	87.895	90.720	91.758	95.524
crime	1	102.681	106.022	107.149	111.587
crime	2	77.167	79.690	80.510	83.869
crime	3	93.435	96.492	97.492	101.572
crime	4	93.073	96.148	97.141	101.189
crime	5	96.265	99.457	100.506	104.693
crime	6	99.083	102.295	103.530	107.675
earthquake	0	0.100	0.110	0.059	0.086
earthquake	1	0.044	0.048	0.119	0.035
earthquake	2	0.144	0.156	0.198	0.172
earthquake	3	1.890	1.985	2.059	2.078
gbpusd	0	16.205	16.709	16.841	17.598
gbpusd	1	15.361	15.831	16.130	16.652
gbpusd	2	16.645	17.219	17.572	18.121
gbpusd	3	17.066	17.545	17.780	18.481
nasahttp	0	9.075	9.398	9.542	9.871
nasahttp	1	85.567	88.303	89.335	92.978
nasahttp	2	4.725	4.858	4.807	5.091
nasahttp	3	131.653	135.921	137.390	143.089
nasahttp	4	15.974	16.508	16.907	17.386
cantor	0	0.477	0.495	0.575	0.500
cantor	1	168.649	174.124	176.015	183.297
cantor	2	242.809	250.686	253.483	263.920
cantor	3	258.598	267.058	269.883	281.084
cantor	4	260.849	269.339	272.333	283.524
cantor	5	259.568	268.034	270.942	282.143
cantor	6	257.058	265.475	268.359	279.451
cantor	7	253.888	262.118	265.055	275.941

the fastest among the three systems. It is 5–66 times faster than the system based on the Hawkes point process. The system based on SVR is slowest in three out of the five benchmarks. This is not surprising as the system runs on a single CPU core. In the experiment with ‘earthquake’ and ‘nasahttp’, the parameter estimation procedure of the Hawkes process stops as it reaches the time limit for model fitting. Therefore, the speedups of the two proposed systems on the two benchmarks are conservative estimates. The software system with SVR is faster than the system with the Hawkes process model if we disable the accelerator for likelihood evaluation and estimate the parameters solely on the CPU platform. With the three benchmarks, ‘crime’, ‘gbpusd’ and ‘cantor’, where the accelerated system with the Hawkes process model finishes model fitting within the 3600-second time limit, the execution times to repeat the model fitting process on the

Table 4.4: Computational efficiency

(An entry in the ‘P-SVR’, ‘P-LR’ and ‘Hawkes’ columns presents the average training time of all event types in the data set using the corresponding predictive system. An entry in the two ‘speedup’ columns records the speedup over the Hawkes system.)

Data	P-SVR	Speedup	P-LR	Speedup	Hawkes
crime	691.6s	0.16x	21.1s	5.17x	109.0s
earthquake	116.8s	30.82x	53.9s	66.81x	3600.0s
gbpusd	916.0s	0.25x	3.4s	65.86x	225.5s
nasahttp	1103.8s	3.26x	224.8s	16.02x	3600.0s
cantor	306.6s	0.54x	3.9s	42.66x	165.4s

host computer are respectively 752.8 seconds, 1266.5 seconds and 895.4 seconds. As a result, the software system with SVR is 1.09 times, 1.38 times, and 2.92 times faster than the software system with the Hawkes process model on six CPU cores.

The results suggest that the proposed system with SVR may be a good candidate if we need accurate predictions. The proposed system with linear regression may be the choice if we have limited time for model fitting. However, the system based on the Hawkes process model does not show a significant advantage compared to the two proposed systems.

The accuracy and efficiency is for real-world applications, because we usually need to model a data set before a hard deadline without compromising the accuracy. For instance, one application of event stream models is the algorithmic high-frequency trading [128]. A stock market may produce millions of random events in every trading day. After the closure of the market, it is desirable to update the trading model with the random events recorded during the trading hours. It is necessary to finish the calculation overnight before the stock market opens again on the next day. However, it may take a few days to fit the model with sufficient accuracy using a desktop computer. The reconfigurable acceleration techniques proposed in this chapter may cut down the time for model fitting from a few days to a few hours so that overnight model updates become possible.

4.4 Summary

In this chapter, we propose a novel predictive model for event streams. Section 4.1 discusses an approximate intensity function for the Hawkes point process. One may evaluate the approximate intensity function by counting the recent events without visiting the full history. Section 4.2 proposes a novel predictive model for event streams using the counting technique in the approximate intensity function. The proposed predictive model works without point process models, which lead to efficient model fitting algorithms. Section 4.3 presents an empirical study to compare predictive systems based on the proposed model with an accelerated system based on the Hawkes process model. The two proposed predictive systems achieve

comparable accuracy with the system based on the Hawkes point process model. Among the three systems, the proposed system based on support vector regression produces the most accurate predictions while the proposed system with linear regression is the fastest in model fitting.

Chapter 5

Pipelined Data Flow Engines for Time Series Analysis

The study of time series is attracting the attention of researchers from various application areas such as financial risk management, statistical biology and seismology. In this chapter, we propose two reconfigurable accelerators for two time series analysis techniques. The first accelerator is for the heteroskedasticity and autocorrelation consistent (HAC) covariance matrix estimation. The second accelerator is for the encoding operation of ordinal patterns. The main novel aspects in this chapter are as follows:

- A novel pipeline-friendly algorithm for HAC estimation derived from the Newey-West estimator. (Section 5.1.3)
- A reconfigurable HAC estimation engine for multivariate time series based on the proposed algorithm. (Section 5.1.4)
- A two-level ordinal pattern encoding scheme which avoids sorting. (Section 5.2.1 and 5.2.2)
- A reconfigurable ordinal pattern encoding engine based on the proposed encoding scheme. (Section 5.2.3)

The rest of this chapter is organised as follows. Section 5.1 presents a fast estimation engine for the long-run covariance matrix which detects correlations between components in multivariate time series. Section 5.2 describes a hardware-based ordinal pattern encoding engine that discovers chaotic properties of time series. Section 5.3 gives a summary of our work.

5.1 Long-Run Covariance Matrix Estimation

An important technique in time series analysis is *heteroskedasticity and autocorrelation consistent (HAC) covariance matrix estimation*, or *HAC estimation* in short. This technique produces an estimate of the long-run covariance matrix for a multivariate time series. The HAC estimation is computationally expensive for long time series data. Moreover, it is challenging to design reconfigurable hardware to accelerate it, because the straightforward algorithm for HAC estimation incurs high memory bandwidth consumption. In this section, we address this challenges by designing a pipeline-friendly algorithm for HAC estimation. The proposed algorithm, as shown in Algorithm 9, saves memory bandwidth by promoting data reuse.

The long-run covariance matrix measures the correlation between different components in a time series. It plays a similar role to the ordinary covariance matrix for non-temporal multivariate data. However, different from the estimation of ordinary covariance matrices for non-temporal data, HAC estimation considers the unique features of time series data such as serial correlation. HAC estimation serves as a standard method in model verification and hypothesis testing. For instance, studies about stock markets [129, 130, 131] employ HAC estimation quantify risks of trading strategies.

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

The organisation of the rest of this section is as follows. Section 5.1.1 explains the reason that we do not map the straightforward HAC estimation algorithm to hardware. Section 5.1.2 transforms the expression of the long-run covariance matrix into a pipeline-friendly form. Section 5.1.3 proposes a pipeline-friendly HAC estimation algorithm and discusses its hardware-oriented features. Section 5.1.4 describes a hardware design that maps our algorithm to a fully-pipelined hardware architecture. Section 5.1.5 provides experimental results of an implementation of our hardware architecture.

5.1.1 Analysis of the Straightforward Algorithm

It is straightforward to design an algorithm following Equation 2.33 for HAC estimation. For instance, Algorithm 7 gives a direct mapping of Equation 2.33. The subroutine `AutoCov(h)` in Algorithm 8 computes a single autocovariance matrix where $\hat{\mu}_m$ is the sample mean of $y_{0,m} \dots y_{N-1,m}$. We call this algorithm the *straightforward HAC estimation algorithm* because

it is directly derived from the definition of the Newey-West estimator. Popular implementation of this algorithm include the ‘sandwich’ econometrics package [132] and the GNU regression, econometrics and time-series library [133].

Algorithm 7: Straightforward HAC Estimation Algorithm

Input: H
Output: $\hat{S}$

```

1  $\hat{S} \leftarrow \text{AutoCov}(0)$ 
2 for  $h \in [1..H]$  do
3    $\hat{\Omega} \leftarrow \text{AutoCov}(h)$ 
4    $\hat{S} \leftarrow \hat{S} + k(\frac{h}{H+1})(\hat{\Omega} + \hat{\Omega}')$ 
5 end
```

Algorithm 8: AutoCov(h)

Input: h
Output: $\hat{\Omega}$

```

1  $\hat{\Omega} \leftarrow \mathbf{0}_{D \times D}$ 
2 for  $t \in [h..(N-1)]$  do
3   for  $i \in [0..(D-1)]$  do
4     for  $j \in [0..(D-1)]$  do
5        $\hat{\Omega}_{i,j} \leftarrow \hat{\Omega}_{i,j} + (y_{t,i} - \hat{\mu}_i)(y_{t-h,j} - \hat{\mu}_j)$ 
6     end
7   end
8 end
```

Taking arithmetic operations as basic operations, the time complexity of the algorithm is $O(D^2HN)$, which means that the execution time grows linearly with D^2 , H and N . Moreover, the lag truncation parameter H should grow with T to keep the results statistically feasible [105]. As a consequence, the algorithm may become computational demanding with long and high-dimensional time series.

The most time-consuming part of the algorithm is the computation of the autocovariance matrix in Algorithm 8. Technically, it is possible to implement this part in a reconfigurable device. However, the memory efficiency is low because the algorithm only executes one multiplication and one addition after two data access operations [134]. It is critical to find optimisations of the algorithm to improve memory efficiency.

5.1.2 Pipeline-Friendly Expression for Long-Run Covariance

In this subsection, we show a way to transform the expression of $\hat{S}$ in Equation 2.33 to eliminate conditionals, to parallelise arithmetic, and to promote data reuse. Then, we present

our new estimation algorithm.

We build the mathematical foundation of our pipeline-friendly algorithm by deriving a novel expression of $\hat{S}$ in Equation 2.33. We first simplify $\hat{S}$ by centralising the data and merging Ω_0 into the weighted sum. This simplification eliminates redundant arithmetic operations and complex conditional logic in the computation. Then we propose an expression of $\hat{S}$ using vector algebra. This expression exposes the parallelism in arithmetic operations and enables considerable data reuse.

The computation of $\hat{S}$ only concerns centralised values of data points. Specifically, the computation only involves $(y_i - \hat{\mu})$ for all data points y_i . As the algorithm may access the centralised values multiple times, we precompute the centralised time series $(u_0, u_1, u_2, \dots, u_{N-1})$ by

$$u_i = y_i - \hat{\mu} \quad (5.1)$$

Let u_i be a zero vector if $t \notin [0..(N-1)]$. This is for simplicity in the presentation of related algorithms. By our precomputing method, Equation 2.34 is equivalent to

$$\hat{\Omega}_h = \frac{1}{N} \sum_{i=h}^{N-1} u_i u'_{i-h} \quad (5.2)$$

When $h = 0$, Ω_h downgrades from a autocovariance matrix to an ordinary covariance matrix which is always symmetric. In other words,

$$\Omega_0 = \Omega'_0 \quad (5.3)$$

By Equation 2.33, we merge Ω_0 to the weighted sum by setting

$$w_h = \begin{cases} \frac{1}{2} & \text{if } h = 0 \\ \kappa(\frac{h}{H+1}) & \text{if } 0 < h \leq H \\ 0 & \text{otherwise} \end{cases} \quad (5.4)$$

By introducing the coefficient $w_0 = \frac{1}{2}$, the expression of $\hat{S}$ in Equation 2.34 becomes

$$\hat{S} = w_0(\hat{\Omega}_0 + \hat{\Omega}'_0) + \sum_{h=1}^H w_h(\hat{\Omega}_h + \hat{\Omega}'_h) \quad (5.5)$$

$$= \sum_{h=0}^H w_h(\hat{\Omega}_h + \hat{\Omega}'_h) \quad (5.6)$$

In other words, the transformation merges the term $\hat{\Omega}_0$ into the weighted sum. The major

consideration behind this merge is that we hope to avoid spacial cases in hardware computation. After merging this computation in the main computational logic, the algorithm may treat $\hat{\Omega}_h$ for all $h \in [0..H]$ in the same way. On the other hand, we may rewrite the expression of $\hat{S}$ as

$$\begin{aligned}\hat{S} &= \sum_{h=0}^H w_h \left[\left(\frac{1}{N} \sum_{i=h}^{N-1} u_i u'_{i-h} \right) + \left(\frac{1}{N} \sum_{i=h}^{N-1} u_i u'_{i-h} \right)' \right] \\ &= \frac{1}{N} (\Psi + \Psi')\end{aligned}\tag{5.7}$$

where

$$\Psi = \sum_{h=0}^H w_h \sum_{i=h}^{N-1} u_i u'_{i-h}\tag{5.8}$$

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

$$G = \lceil \frac{H+1}{c} \rceil - 1\tag{5.9}$$

where $\lceil \cdot \rceil$ is the ceiling operator.

$$\Psi = \sum_{g=0}^G \sum_{h=gc}^{gc+c-1} w_h \sum_{i=h}^{N-1} u_i u'_{i-h}\tag{5.10}$$

If c is a factor of $(H + 1)$ then Equation 5.10 obversely holds. If not, terms with $h > H$ will be calculated, but Equation 5.10 still holds in this case because $w_h = 0$ for all $h > H$. The value of a single entry of Ψ can be computed by

$$\begin{aligned}\Psi_{d,d'} &= \sum_{g=0}^G \sum_{h=gc}^{gc+c-1} w_h \sum_{i=h}^{N-1} u_{i,d} \cdot u_{i-h,d'} \\ &= \sum_{g=0}^G \tilde{w}_{g,c} \tilde{r}_{g,c,d,d'}\end{aligned}\tag{5.11}$$

where

$$\tilde{w}_{g,c} = \begin{pmatrix} w_{gc} & w_{gc+1} & \dots & w_{gc+c-1} \end{pmatrix} \quad (5.12)$$

$$\tilde{r}_{g,c,d,d'} = \begin{pmatrix} \sum_{i=gc}^{N-1} u_{i,d} \cdot u_{i-gc,d'} \\ \sum_{i=gc+1}^{N-1} u_{i,d} \cdot u_{i-(gc+1),d'} \\ \vdots \\ \sum_{i=gc+c-1}^{N-1} u_{i,d} \cdot u_{i-(gc+c-1),d'} \end{pmatrix} \quad (5.13)$$

By Equation 5.13, we decompose $\tilde{w}_{g,c}$ as a product of two vectors. The aim this decomposition is to seek for data reuse opportunities. One may construct the vector $\tilde{w}_{g,c}$ using the weights precomputed by Equation 5.4. We only need to focus on the computation of $\tilde{r}_{g,c,d,d'}$. The forms of all entries in $\tilde{r}_{g,c,d,d'}$ are similar. We take advantage of the structural similarity to simplify the expression of $\tilde{r}_{g,c,d,d'}$ and observe the data access pattern. Aligning the lower bounds of the summation operators in Equation 5.13, we have

$$\tilde{r}_{g,c,d,d'} = \begin{pmatrix} \sum_{k=0}^{N-gc-1} u_{k+gc,d} \cdot u_{k,d'} \\ \sum_{k=0}^{N-gc-2} u_{k+gc+1,d} \cdot u_{k,d'} \\ \vdots \\ \sum_{k=0}^{N-gc-c} u_{k+gc+c-1,d} \cdot u_{k,d'} \end{pmatrix} \quad (5.14)$$

We have defined that $u_i = 0$ when $i \geq N$. Therefore, we can set the upper bounds of all summation operations in Equation 5.14 to $(N - gc - 1)$. Then the expression of $\tilde{r}_{g,c,d,d'}$ can be further simplified as

$$\tilde{r}_{g,c,d,d'} = \begin{pmatrix} \sum_{k=0}^{N-gc-1} u_{k+gc,d} \cdot u_{k,d'} \\ \sum_{k=0}^{N-gc-1} u_{k+gc+1,d} \cdot u_{k,d'} \\ \vdots \\ \sum_{k=0}^{N-gc-1} u_{k+gc+c-1,d} \cdot u_{k,d'} \end{pmatrix} \quad (5.15)$$

$$= \sum_{k=0}^{N-gc-1} u_{k,d'} \begin{pmatrix} u_{k+gc,d} \\ u_{k+gc+1,d} \\ \vdots \\ u_{k+gc+c-1,d} \end{pmatrix} \quad (5.16)$$

In Equation 5.16, we unify the lower bounds and upper bounds of the summation operations respectively. The expression of $\tilde{r}_{g,c,d,d'}$ becomes the sum of a collection of scalar products. Given any $k \in [0..(N - gc - 1)]$, the data entries involved in Equation 5.16 constitute a contiguous subsequence of the i -th component of the centralised time series data.

An intuitive explanation of the mathematical reformulation is that we partition the infor-

Algorithm 9: Pipeline-Friendly HAC Estimation

Input: G
Output: $\hat{S}$

```

1 for  $d \in [0..D-1]$  do
2   for  $d' \in [0..D-1]$  do
3      $\Psi_{i,j} \leftarrow 0$ 
4     for  $g \in [0..G]$  do
5        $\tilde{w} \leftarrow (w_{gc} \ w_{gc+1} \ \dots \ w_{gc+c-1})$ 
6        $\tilde{r} \leftarrow \text{Pass}(g, c, d, d')$ 
7        $\Psi_{d,d'} \leftarrow \Psi_{d,d'} + \tilde{w} \tilde{r}$ 
8     end
9   end
10 end
11  $\hat{S} \leftarrow \frac{1}{N}(\Psi + \Psi')$ 

```

Algorithm 10: Subroutine Pass

Input: g, c, d, d'
Output: $\tilde{r}$

```

1  $\tilde{r} \leftarrow \mathbf{0}_{D \times 1}$  for  $k \in [0..(N - gc - 1)]$  do
2    $\tilde{r} \leftarrow \tilde{r} + u_{k,d'} \begin{pmatrix} u_{k+gc,d} \\ u_{k+gc+1,d} \\ \vdots \\ u_{k+gc+c-1,d} \end{pmatrix}$ 
3 end

```

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

5.1.3 Pipeline-Friendly HAC Estimation Algorithm

We design a tangible algorithm to compute $\hat{S}$ using the equations developed in the last subsection. Following a top-down design approach, we first investigate the way to compute $\hat{S}$ assuming the availability of all $\tilde{r}_{g,c,d,d'}$; then we discuss the way to compute $\tilde{r}_{g,c,d,d'}$.

Suppose we are able to compute the value of $\tilde{r}_{g,c,d,d'}$ for all g, c, i and j . We can then compute all entries in Ψ by Equation 5.11. Once the computation of Ψ finishes, it is then possible to compute $\hat{S}$ by Equation 5.6, as shown in Algorithm 9. The algorithm queries the value of $\tilde{r}_{g,c,d,d'}$ by invoking the subroutine $\text{Pass}(g, c, d, d')$ shown in Algorithm 10. This subroutine passes through the data and compute $\tilde{r}_{g,c,d,d'}$ by Equation 5.16. In Algorithm 9

and 10, the variables $\tilde{w}$ and $\tilde{r}$ correspond respectively to $\tilde{w}_{g,c}$ and $\tilde{r}_{g,c,d,d'}$ in Equation 5.11.

We call Algorithm 9 the *pipeline-friendly HAC estimation algorithm* because we consider its most time-consuming subroutine, $\text{Pass}(g, c, d, d')$, as an excellent candidate to be mapped to a pipelined hardware architecture. The reasons are as follows.

- This subroutine contains no conditional statements. We consider it beneficial to avoid such statements because they may fork the data path and lead to redundant resource consumption on reconfigurable hardware.
- Many arithmetic operations in the algorithm may execute in parallel. We can observe from Line 2 in Algorithm 10 that $\tilde{r}_{g,c,d,d'}$ is the sum of the a collection of vector scalar products. As the components in the vector are independent, the addition and multiplication operations on all the c components of the vector can take place in parallel.
- There is a considerable data reuse pattern behind the subroutine. In the computation of $\tilde{r}_{g,c,d,d'}$, when $k = k_0$, the accessed data elements are $u_{k_0,i}$ and $u_{k_0+gc,j} \dots u_{k_0+gc+c-1,j}$; when $k = k_0 + 1$, the accessed data elements are $u_{k_0+1,i}$ and $u_{k_0+gc+1,j} \dots u_{k_0+gc+c,j}$. All data elements accessed when $k = k_0 + 1$, except $u_{k_0+1,i}$ and $u_{k_0+gc+c,j}$, have been previously accessed when $k = k_0$. We may design a caching scheme to take advantage of this data reuse pattern. With a perfect caching scheme, the algorithm retrieves only two data elements from the main memory for each scalar product. This is our essential idea to crack the memory bandwidth bottleneck.

Admittedly, the time complexity of the pipeline-friendly algorithm is still $O(D^2HN)$ which is identical to the straightforward algorithm. Moreover, the pipeline-friendly design may incur redundant computations when c is not a factor of $(H + 1)$. However, the pipeline-friendly properties enable us to achieve significant acceleration in practice.

5.1.4 Hardware Design

In this section, we discuss a pipelined architecture for the subroutine $\text{Pass}(g, c, d, d')$. We first present the hardware architecture and explain its interactions with the host computer. Then we discuss a theoretical model for its execution time.

The elementary computational unit, which we call a *bead*, is described in Figure 5.1. A bead handles the computation for one component of $\tilde{r}$ in Line 2 in Algorithm 10. We link up c beads with the organisation shown in Figure 5.2. Specifically, we use a single buffer register to store a single data element from its input stream. For ease of discussion, we call this buffer register the *broadcasting buffer* hereafter. The broadcasting buffer serves as one input to all the c beads. At the end of each cycle, the buffer retrieves a data element from the input stream.

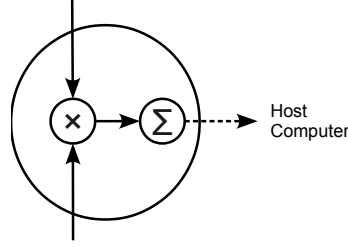


Figure 5.1: Structure of a bead

Admittedly, the broadcasting buffer may have a high fan-out when the number of beads is large. However, hardware compilers may apply optimisations to resolve fan-outs automatically. In addition, it is also possible to manually reduce the fan-outs [135].

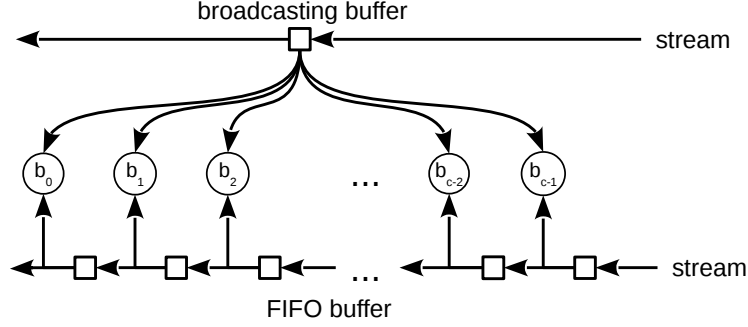


Figure 5.2: Proposed architecture for HAC estimation

We customise on-chip fast memory in the reconfigurable hardware device, e.g. block RAMs, to become a first-in-first-out (FIFO) buffer with c storage units. This buffer is used to store c consecutive data elements of its input stream. At the end of each cycle, each element of the FIFO buffer accepts the data element from its right neighbour. The previous leftmost element is moved out of the buffer and discarded. The input stream supplies data to the rightmost element in the FIFO buffer. Each storage unit in the FIFO buffer contributes another input to a bead.

Each time when Algorithm 10 is invoked, the data streams $u_{0,i} \dots u_{T-gc-1,i}$ and $u_{gc,j} \dots u_{T-1,j}$ are streamed into the architecture for the computation of $\tilde{r}$. This streaming process can be divided into the following three stages.

1. Initialisation stage: all registers in all beads reset to zero. The broadcasting buffer loads the first element of the stream $u_{0,d'} \dots u_{N-gc-1,d'}$. The FIFO buffer is filled with the first c elements of the stream $u_{gc,d} \dots u_{N-1,d}$. In other words, at the end of the initialisation stage, the broadcasting buffer contains $u_{0,d'}$ and the FIFO buffer contains $u_{gc,d} \dots u_{gc+c-1,d}$.

2. Pipeline processing stage: in every cycle, each bead accumulates the product of its inputs. Then both the broadcasting buffer and the FIFO buffer load the next data element from their corresponding input stream. This process runs for $(N - gc)$ cycles before termination.
3. Result summation stage: each bead reports its accumulation result to the host computer. The accumulation result of the q -th bead is the q -th component of $\tilde{r}$. The host computer then revises $\Psi_{i,j}$ according to $\tilde{r}$.

We can observe from the hardware design that the pipeline-friendly features behind our algorithm are exploited. The FIFO buffer provides a perfect caching mechanism to take advantage of the data reuse pattern discussed in Section 5.1.3. In each cycle, the hardware only loads two data elements from the input stream, and all the remaining elements are obtained from the FIFO buffer. In other words, the memory bandwidth requirement is both small and constant. The bandwidth requirement is independent of the number of beads, which suggests that the computational efficiency may scale up well with the amount of on-chip logical resources without being limited by the memory bottleneck. Furthermore, the architecture is highly modularised. The major computational modules of the architecture are the beads and the two buffers. These modules are structurally uncomplicated and can be tested individually, which reduces the potential effort in implementation and debugging.

We are interested in processing long time series in this study. Therefore, the pipeline processing stage would be the most time-consuming one. The computation of each entry of Ψ needs to invoke $\text{Pass}(g, c, d, d')$ for $(G + 1)$ times. The number of cycles spent on the g -th invocation is $(N - gc)$. Let F be the clock frequency of the reconfigurable device. The total execution time of this stage in all invocations is

$$ET_P = \frac{1}{F} \sum_{g=0}^G (T - gc) = \frac{(G + 1)(2T - Gc)}{2F} \quad (5.17)$$

For ease of discussion, we will call this time *the theoretical pipeline processing time* hereafter. Let ET_ϵ be the total execution time that is not spent on the pipeline processing stage. Then the total execution time is

$$ET = ET_P + ET_\epsilon \quad (5.18)$$

We do not attempt to model ET_ϵ since we consider it platform-dependent, unpredictable and negligible. Specifically, ET_ϵ depends on the configuration and execution status of the acceleration platform, and this quantity is unlikely to be significant compared to the pipeline processing time, especially when the time series is long.

5.1.5 Experimental Evaluation

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

In our implementation, we deploy 384 beads and set the clock frequency to 100MHz. We also build a CPU-based system by implementing the straightforward HAC estimation algorithm on the host computer of the acceleration platform. We validate the results of the proposed systems against a MATLAB implementation of the straightforward HAC estimation algorithm. The computational efforts for all entries of the long-run covariance matrix are identical. Therefore, we describe the computational efficiency in terms of the computation time of a single entry. Following the experiment scheme in [105], we generate the data using a vector autoregression (VAR) model [99]. The time series and lag parameters tested in the two experiments are different, and we will present the settings individually.

We first study the computational efficiency of our data flow engine. We enable all the 384 beads on the data flow engine to demonstrate its full power. Time series with different lengths are tested with different lag truncation parameters. Specifically, we use four time series with length $T = 10^5, 10^6, 10^7, 10^8$. Following [105], we use lag truncation parameters H in the form

$$H = \lfloor \gamma T^{\frac{1}{3}} \rfloor \quad (5.19)$$

where $\lfloor x \rfloor$ is the smallest integer not larger than x ; γ is a data-dependent positive real number. to simulate the computation of different types of data, for each data set we select 12 different values of γ in the range $0.25 \leq \gamma \leq 3.00$ which is slightly wilder than the range investigated in [105].

We present the experiment results in Figure 5.3. Each row of the figures corresponds to a time series length. The first two columns of figures are computational efficiency results from the CPU-based system and the FGPA-based system respectively. FPGA(P) is the theoretical pipeline processing time. The third column records the speedup of the FPGA-based system over the CPU-based system.

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

The execution time of all systems increases as the lag truncation parameter grows. However, the growth pattern of the FPGA based system is significantly different from that of CPU-based systems. In particular, the execution times of the two CPU-based system grow linearly with different slope. This linear growth can be explained by the time complexity of the algorithm. The time of the FPGA-based system increases like stairs. This is because the FPGA-based system handles the computation for different lags in batches.

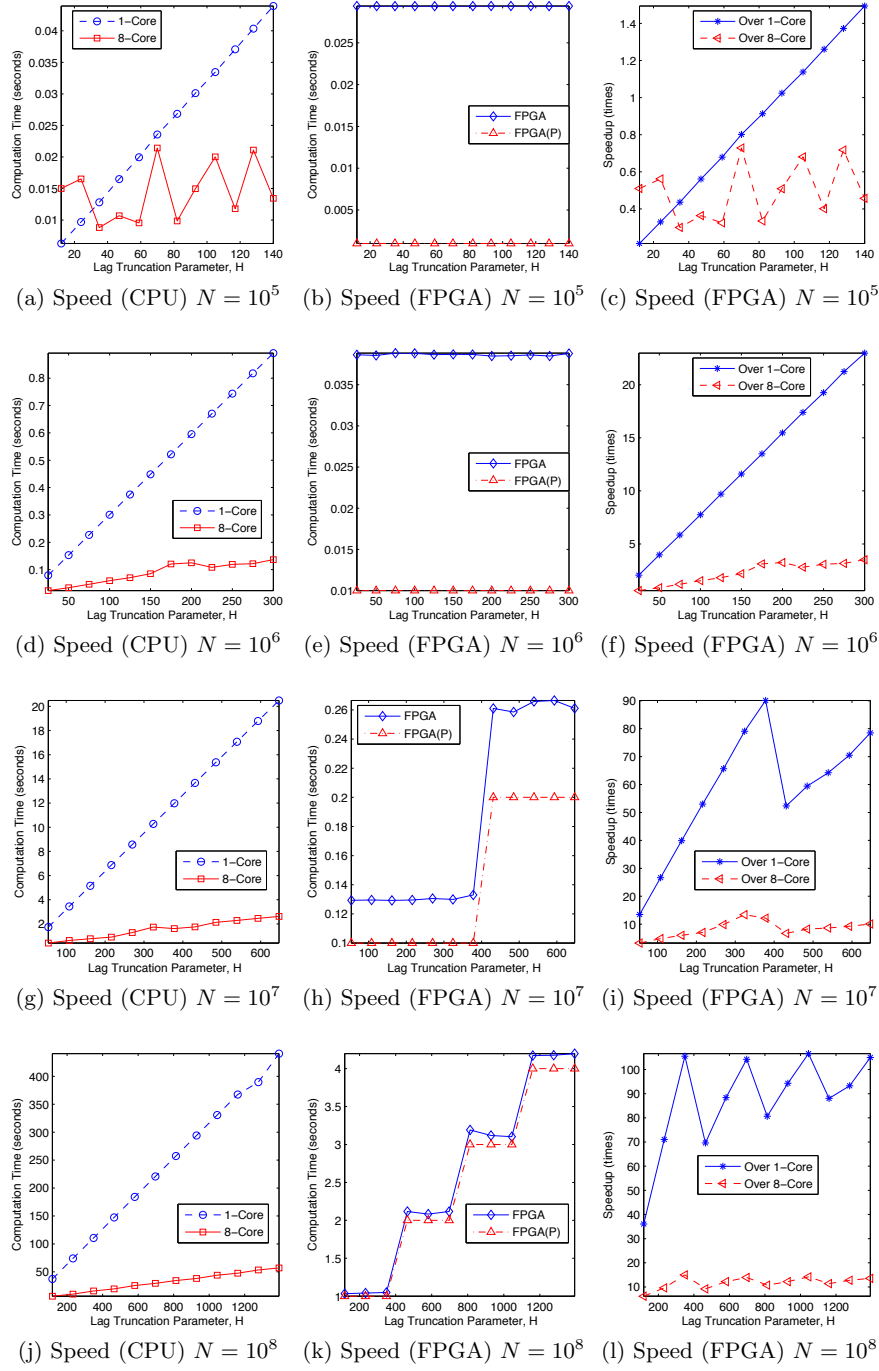


Figure 5.3: Results on computational efficiency

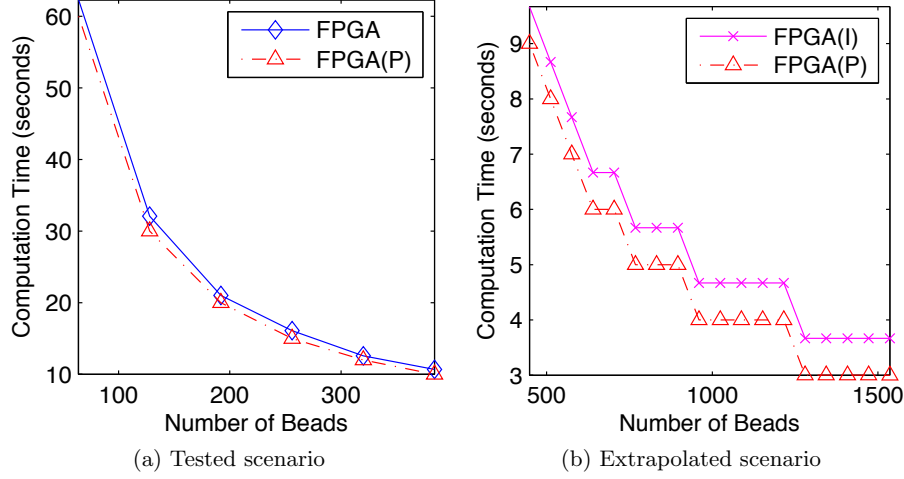


Figure 5.4: Results on scalability

Due to the difference in the growth pattern in execution time, it is not surprising that the speedup of the FPGA-based system over the CPU-based one appears an zig-zag pattern in a periodical manner, as shown in Figure 5.3i and 5.3l.

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

We also test the computational efficiency of the FPGA-based system with different numbers of beads. The maximum number of beads that we manage to deploy is 384. Therefore, we only test the configurations where the number of beads is no more than 384. We plot the results in Figure 5.4 along with the corresponding theoretical pipeline processing time. In this figure, FPGA(P) is the theoretical pipeline processing time. FPGA(E) is the estimated computational efficiency of the FPGA-based system in the extrapolated scenario. When c , the number of beads, reaches 384, the execution time reaches 10.67s, achieving 111 and 14 times speedup over the CPU-based system on one core and eight cores respectively.

There is a significant trend that the execution time is shortened as the number of beads c increases. Another trend, which is not significant in the figure, is that the gap between the experimental execution time and the theoretical pipeline processing time becomes narrower as the number of beads c increases. The gap decreases from 2.2934s when $c = 64$, to 0.6683s when

$c = 384$. As shown in Figure 5.4, the theoretical pipeline processing time continues to decrease when more beads are deployed along the pipeline. We can obtain a conservative estimate of the computational efficiency of a system with more than 384 beads, by extrapolating the gap value of 0.6683s for 384 beads as an upper bound for designs with more than 384 beads. The computation time values can then be estimated by adding this gap value to the corresponding theoretical pipeline processing time. This estimation suggests that our proposed architecture scales well with the amount of available computational resources.

5.2 Ordinal Pattern Encoding

Ordinal pattern encoding is a fundamental calculation in the ordinal analysis of time series. It is computationally demanding particularly for high query orders and large time series data. Reconfigurable computing is a promising acceleration technology for encoding ordinal patterns. However, it is challenging to design a reconfigurable accelerator because the calculations involve sequence sorting that does not have an efficient hardware implementation. We address this challenge by proposing a two-level hardware-oriented ordinal pattern encoding scheme. The proposed encoding scheme, as shown in Figure 5.6, may extract ordinal patterns from time series without sorting.

The rest of this section is organised as follows. Section 5.2.1 presents the first level of encoding which produces an intermediate representation for an ordinal pattern without sorting. Section 5.2.2 presents the second level of encoding that compresses the intermediate representation to a compact final result. Section 5.2.3 describes a hardware design based on the two-level encoding scheme. Section 5.2.4 provides experimental results of an implementation of our proposed hardware design, and explains experimental observations. Section 5.2.5 describes a case study applying our proposed architecture to calculate the permutation entropy.

5.2.1 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 is the sequence of indexes in the sorted pairs. 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 $(6 - 0, 8 - 1, 2 - 2)$. Sorting the pairs by the value, we get $(2 - 2, 6 - 0, 8 - 1)$. Therefore, the ordinal pattern in the permutation representation is $(2, 0, 1)$.

Although the 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. Both methods may result in troubles in reconfigurable computing.

- Comparison-based sorting algorithms are mathematically acceptable for reconfigurable computing. There are papers adopting reconfigurable architectures for sorting sequences [136, 137, 138, 139]. 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 $O(n \log_2 n)$. In terms of hardware computation, although it is possible to use any number of comparators to perform sorting, one needs to deploy at least $n \log_2 n$ comparators to make the design fully pipelined. Furthermore, it is necessary to plan additional data paths and buffers to support at least n data movement operations.
- Non-comparison-based sorting algorithms usually have lower time complexity compared to 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 if all entries are different from each other. However, the amount of on-chip memory in a hardware platform is usually limited.

In this study, we use the Lehmer code [140] to avoid sorting. We first encode the ordinal pattern of a sequence into a Lehmer code rather than a permutation. Let $x = (x_0, x_1, \dots, x_{n-1})$ be a sequence of length n , the Lehmer code of x is also a sequence with length n in the form $\mathcal{L}(x) = (l_0, l_1, \dots, l_{n-1})$ where

$$l_i = \text{card}(\{x_j : i < j, x_j < x_i\}) \quad (5.20)$$

and $\text{card}(S)$ 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 $\mathcal{L}(x) = (1, 5, 2, 0, 1, 0, 0)$. A useful 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 5.20 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 [141] of the Lehmer code draw our attention:

1. If the Lehmer code of $x = (x_0, \dots, x_{n-1})$ is $\mathfrak{L}(x) = (l_0, \dots, l_{n-1})$, then the Lehmer code of $x' = (x_1, \dots, x_{n-1})$ must be $\mathfrak{L}(x') = (l_1, \dots, l_{n-1})$. 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_0 directly result in the removal in the corresponding Lehmer code entry l_0 .
2. If the Lehmer code of $x' = (x_0, \dots, x_{n-1})$ is $\mathfrak{L}(x') = (l_0, \dots, l_{n-1})$, then the Lehmer code of $x'' = (x_0, \dots, x_{n-1}, x_n)$ must be in the form $\mathfrak{L}(x'') = (l_0^+, \dots, l_{n-1}^+, 0)$, where

$$l_i^+ = \begin{cases} l_i + 1 & \text{if } x_i > x_{n+1} \\ l_i & \text{otherwise} \end{cases} \quad (5.21)$$

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 sub-sequences from a time series. 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. Specifically, the above two properties enable us to encode a sequence by updating the last encoding result. Let $x = (x_0, \dots, x_{N-1})$ be a univariate time series of length N , and let n be the query order. When the Lehmer code of a sequence $(x_{k+1}, \dots, x_{k+n})$ is available, we can easily obtain the Lehmer code of $(x_{k+2}, \dots, x_{k+n+1})$ using the two properties.

To encode the ordinal pattern of a sequence with length n , we only need $(n-1)$ comparison operations. The hardware 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.

5.2.2 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. Therefore, 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 a Lehmer code into an unsigned integer using the factorial number system [142, 143]. Let $\mathfrak{L}(x) = (l_1, \dots, l_n)$ 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.

$$\mathfrak{C}(x) = \overline{l_n l_{n-1} \dots l_2 l_1} = \sum_{i=1}^n l_i \cdot (i - 1)! \quad (5.22)$$

For example, we treat the Lehmer code $(2, 3, 1, 0, 0)$ as $\overline{23100}_!$ in the factorial number system. The number $\overline{23100}_!$ is equal to $2 \times 4! + 3 \times 3! + 1 \times 2! + 0 \times 1! + 0 \times 0! = 68$ in decimal or 1000100 in binary. In this research, we use an unsigned integer to represent an ordinal pattern. For instance, we list the compress codes for all the six ordinal patterns of order 3 in Table 5.1. We provide a complete example of the two-level encoding method in Table 5.2 where we encode $x = (6, 2, 3, 8, 1, 4)$ with $n = 3$.

Table 5.1: Lehmer code for ordinal patterns with order 3

Ordinal pattern	Permutation	Lehmer	Compressed (D)	Compressed (B)
	(0, 1, 2)	(0, 0, 0)	0	000
	(0, 2, 1)	(0, 1, 0)	1	001
	(1, 0, 2)	(1, 0, 0)	2	010
	(1, 2, 0)	(1, 1, 0)	3	011
	(2, 0, 1)	(2, 0, 0)	4	100
	(2, 1, 0)	(2, 1, 0)	5	101

We may analysis the code compactness by measuring the 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 $[0..(n - 1)]$. Each integer requires $\lceil \log_2 n \rceil$ to store. As a result, the number of

Table 5.2: Example of encoding process

Time series	Ordinal pattern	Lehmer code	Compressed code
6 2 3 8 1 4		(2, 0, 0)	4
6 2 3 8 1 4		(0, 0, 0)	0
6 2 3 8 1 4		(1, 1, 0)	3
6 2 3 8 1 4		(2, 0, 0)	4

required bits is

$$b_P(n) = n \lceil \log_2 n \rceil \quad (5.23)$$

- 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 $[0..(n-i-1)]$. It is unnecessary to record the n -th entry because it is always zero. Aside from the last entry, the i -th entry takes $\lceil \log_2(n-i) \rceil$ bits. As a result, the number of bits required by Lehmer code is

$$b_L(n) = \sum_{i=0}^{n-2} \lceil \log_2(n-i) \rceil \quad (5.24)$$

- 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

$$b_C(n) = \lceil \log_2(n!) \rceil \quad (5.25)$$

We plot $b_P(n)$, $b_L(n)$ and $b_C(n)$ against the query order n in Figure 5.5. It can be observed from the figure that $b_C(n) \leq b_L(n) \leq b_P(n)$ for all $1 \leq n \leq 20$. Moreover, 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 compress the code with the best possible compression ratio. This compactness of the compressed code guarantees the lowest

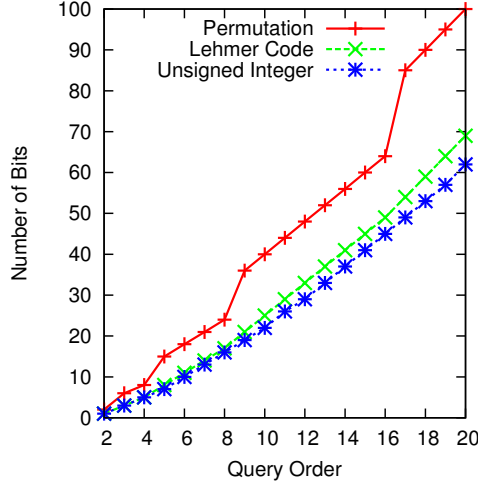


Figure 5.5: Number of bits required by each encoding scheme

possible bandwidth consumption from the reconfigurable accelerator to another device.

The typical aim of encoding is to model the distribution of ordinal patterns. Therefore, a useful 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 [144]. The hash table operations take more time than the array operations of the integer representation.

5.2.3 Pipelined Reconfigurable Accelerator

We map the proposed two-level encoding scheme to a reconfigurable dataflow engine. The structure of the data path is shown in Figure 5.6. The design parameter N is the maximum query order allowed. The time series data flows from 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 5.21. 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 input. This component is in charge of updating the Lehmer code using Equation 5.21.
4. The compressor is composed of $(\aleph - 1)$ multipliers and a single adder. This structure corresponds to the compressing operation in Equation 5.22. 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 [145, 146] before streaming data into the pipeline.

An example demonstrating the encoding process of a sequence is shown in Figure 5.7. In the initial stage shown in Figure 5.7(a), the accelerator has just finished the encoding of the subsequence (3, 8, 4, 2). The storage units in the data buffers shift to the left. A new data entry 6 occupies the rightmost unit. We temporary disable the input port of the FIFO buffer to focus on the encoding process of (8, 4, 2, 6). The increment decider decides whether to revise each entry of the Lehmer code as shown in Figure 5.7(b). Then the Lehmer code

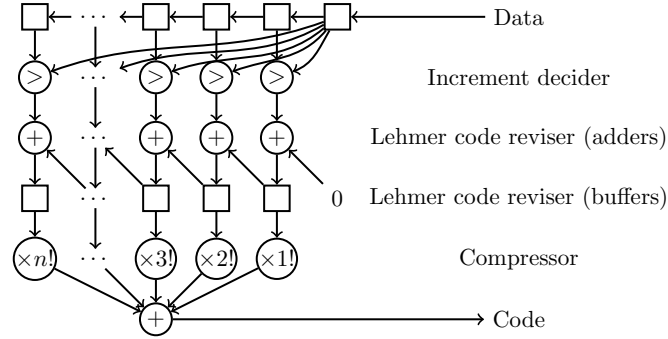


Figure 5.6: Structure of the data path

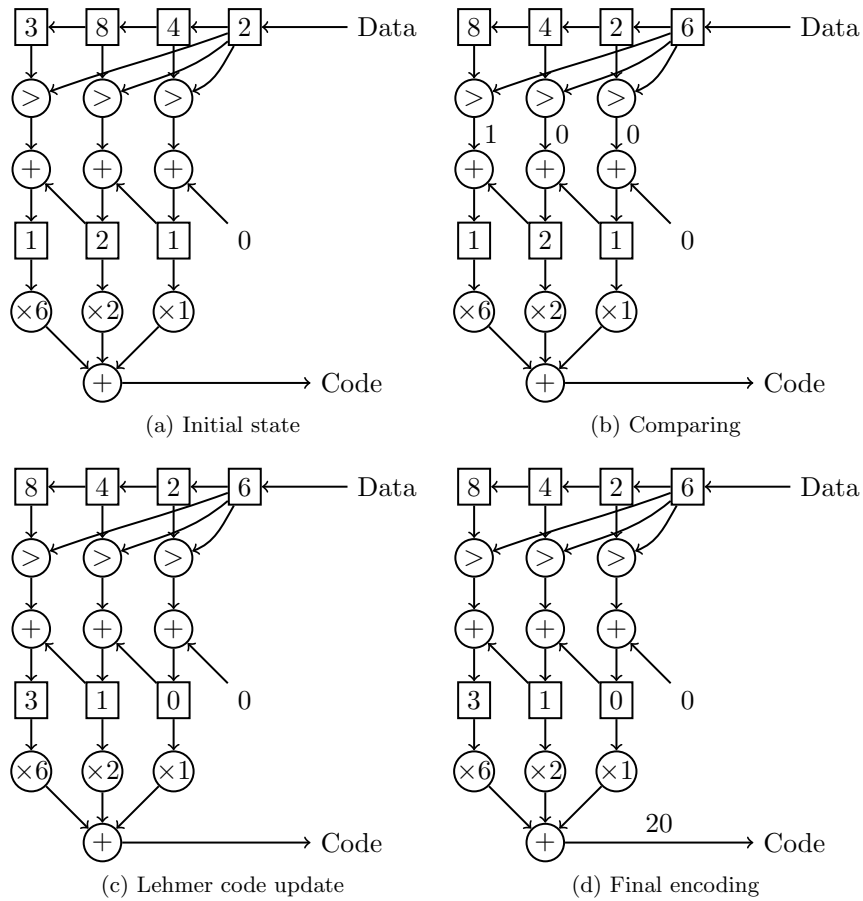


Figure 5.7: Example execution of the data path

reviser updates the Lehmer code using the adders as shown in Figure 5.7(c). Finally, the compressor translates the Lehmer code into an unsigned integer as shown in Figure 5.7(d).

Suppose that 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 5.21. 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.

5.2.4 Experimental Evaluation

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

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 [107]. 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 data paths in parallel. A designer may use fewer data paths to fit a smaller FPGA, or more data paths to exploit available resources. In our experimental implementation, we deploy 6 data paths 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.

We validate the correctness of our implementation using the same data for computational efficiency evaluation. We compare the histogram generated by our system with the one pro-

duced by the MATLAB implementation described in [147]. We observed no difference between the two histograms in all tests.

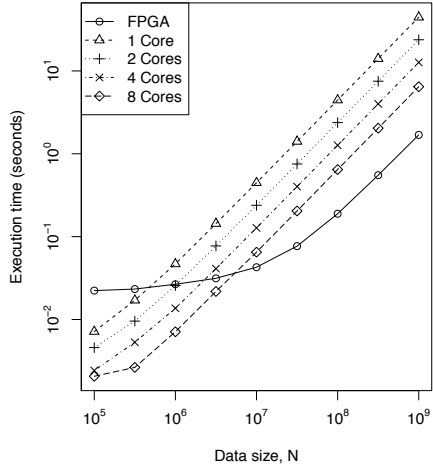
The computational efficiency results are shown in Figure 5.8. The two figures in the first row record the execution times in ordinal pattern encoding for query orders $n = 6, 12$. To observe the computational efficiency for different data sizes, we generate data with lengths from 10^5 to 10^9 . Lengths 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 two figures in the second row show the speedups of the FPGA system over the CPU system with one core to eight cores. 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 data path. 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 data path.

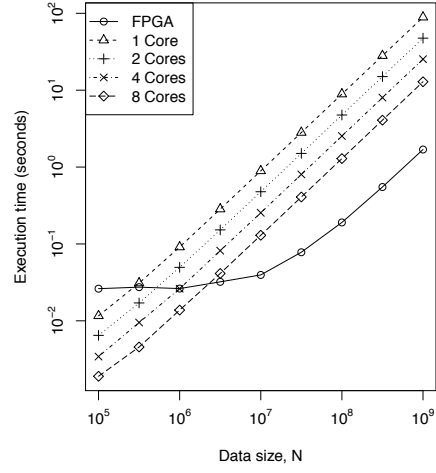
The maximum speedup is observed when the query order n is 12 and the data size reaches 10^9 . In this case, the FPGA implementation is respectively 52.5 times, 28.1 times, 15.0 times and 7.6 times faster than the CPU implementation running on one core, two cores, four cores, and eight cores.

5.2.5 Case Study: Calculating Permutation Entropy

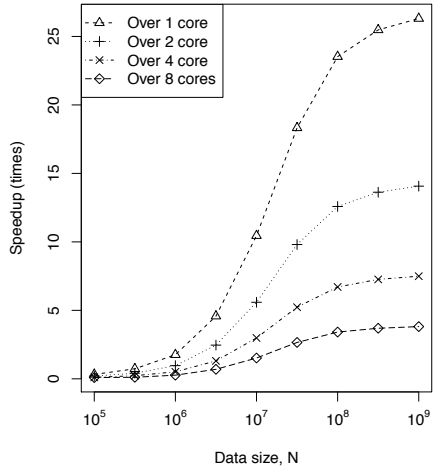
Permutation entropy [107] is a real number that measures the statistical complexity and predictability for a time series. This quantity is similar to the Shannon entropy [148] for non-temporal data. However, they are very different in a statistical sense since the permutation entropy contains information about unique features of time series data such as self-coherence. The permutation entropy is widely used to detect dynamic changes [149] and to estimate the predictability [150] 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.



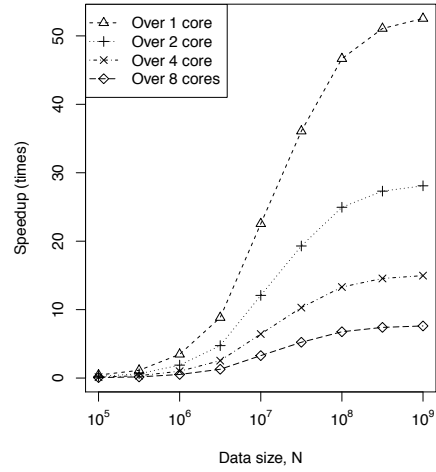
(a) Execution time, $n=6$



(b) Execution time, $n=12$



(c) Speedup, $n=6$



(d) Speedup, $n=12$

Figure 5.8: Computational efficiency results

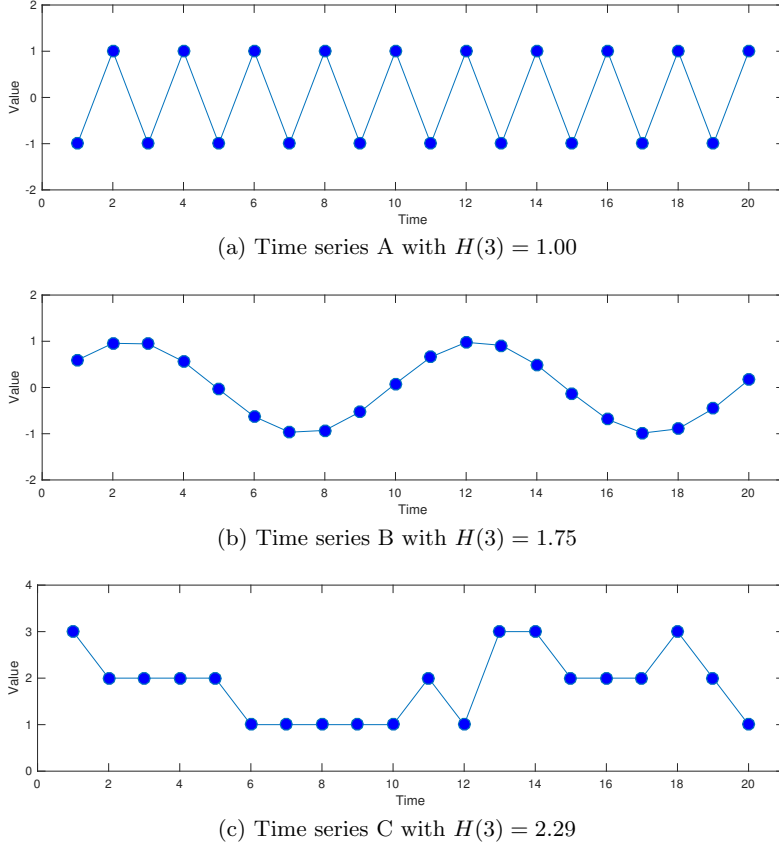


Figure 5.9: Time series with different complexity

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

$$p(\pi_m) = \frac{\text{card}(\{t : \mathfrak{F}(x_{t+1}, \dots, x_{t+n}) = \pi_m\})}{T - n + 1} \quad (5.26)$$

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

$$H(n) = - \sum_{m=1}^M p(\pi_m) \log_2 p(\pi_m) \quad (5.27)$$

We provide an example to show the relationship between the permutation entropy and

the statistical complexity of time series. Figure 5.9 shows three time series with different complexity. The permutation entropy of the three time series with order 3 are respectively 1.00, 1.75 and 2.29. The three entropy values suggests that, among the three time series, A is the least complex one while C is the most complex one.

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.* [151] calculate the permutation entropy of the electroencephalogram (EEG) to quantify the effect of anaesthetic drug. Frank *et al.* [152] use the permutation entropy to analyse heartbeat sequences. They conclude that permutation entropy significantly improves the classification accuracy of fatal behavioural states. Nicolaou *et al.* [153] use the 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.* [154] propose to use the permutation entropy to 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.* [155] suggest the use of a modified version of permutation entropy to estimate uncertainty and volatility in markets. Ortiz-Cruz *et al.* [156] use permutation entropy along with other entropy measures to analyse the dynamics of crude oil market.

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 5.26; and (iii) entropy computation where the distribution of ordinal patterns is extracted from the histogram and the permutation entropy is calculated by Equation 5.27.

The ordinal pattern encoding is the most time-consuming procedure. We accelerate this part using the architecture proposed in Section 5.2.3. The other two operations are unsuitable 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.

On-chip memory resources such as the block RAM in FPGAs are able to support efficient random memory access. However, the storage capacity of the on-chip 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

$$|h| \leq \min\{(T - n + 1), n!\} \quad (5.28)$$

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 on-chip memory resources required by the histogram far exceed the available resources in a Xilinx Virtex-6 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 5.27.

Evaluation with Real-World Data

We use two large data sets to test the computational efficiency of the proposed hybrid permutation entropy calculator. One data set is an EEG data set with 38400000 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 computational efficiency results are shown in Table 5.3. The execution times for the CPU system and the FPGA-based system are respectively recorded in the third and fourth column in the table. The fifth column, titled ‘Speedup’ shows the speedups of the FPGA-based system over the CPU system considering both the histogram construction time and the entropy calculation time.

The FPGA-based system achieves 1.8–15.9 times speedup over the CPU system in the tests. We observe lower speedups in permutation entropy calculation than ordinal pattern

Table 5.3: Computational efficiency for permutation entropy

Data	Order	CPU+FPGA	CPU	Speedup
EEG	3	0.1357s	1.0544s	7.8 times
EEG	6	0.1346s	1.7549s	13.0 times
EEG	9	0.2639s	1.2419s	4.7 times
EEG	12	3.9813s	7.2017s	1.8 times
GBPUSD	3	0.7648s	6.4079s	8.4 times
GBPUSD	6	0.7701s	12.2070s	15.9 times
GBPUSD	9	1.6540s	17.5380s	10.6 times
GBPUSD	12	7.5475s	30.7858s	4.1 times

encoding, 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, which is different from the observation concerning ordinal pattern encoding. This is because, when the query order is higher, the construction of the histogram takes longer time than small query orders. Therefore, the overall speedup is negatively affected even if we have high speedup in ordinal pattern encoding.

5.3 Summary

This chapter presents two reconfigurable accelerators for time series analysis. Section 5.1 presents a reconfigurable computing solution to the heteroskedasticity and autocorrelation consistent (HAC) covariance matrix estimation for multivariate time series. Rather than providing a hardware design for an existing HAC estimation algorithm, we derive a novel algorithm which is designed specifically for hardware implementation to eliminate conditionals, to parallelise arithmetic, and to promote data reuse. We then propose an efficient and scalable hardware architecture based on our algorithm and analyse its computational efficiency using both theoretical and empirical approaches. Section 5.2 proposes a reconfigurable acceleration solution for ordinal pattern encoding. We begin by proposing a two-level ordinal pattern 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 and by parallelising arithmetic operations. In addition, we conduct a case study applying the ordinal pattern encoding engine to the calculation of permutation entropy.

Chapter 6

Conclusion

This thesis focuses on reconfigurable predictive systems for event streams. In this chapter, we recap our achievements and discuss directions for future development.

6.1 Summary of Achievements

Our achievements presented in this thesis include three parts. The first part is the design of reconfigurable acceleration solutions for the calculations in Hawkes self-exciting point process models. The second part is a novel predictive model for event streams with efficient model fitting. The third part is the design of reconfigurable accelerators for time series analysis.

6.1.1 Custom Computing for Hawkes Process

A critical calculation in Hawkes process models is intensity evaluation. The intensity of the Hawkes process is stochastic, changes in time and depends on the history, which makes the calculation computationally expensive. Our achievements on intensity evaluation are as follows.

- Identification of the computationally demanding part of intensity evaluation.
- A hardware accelerator for the computationally demanding part.
- Evaluation of the hardware with four data sets.

Another important calculation in Hawkes process models is likelihood evaluation. Likelihood evaluation is the necessary operation for parameter estimation. Our achievements on likelihood evaluation are as follows.

- Mathematical transformation that eliminates the complex data dependency in likelihood evaluation for univariate Hawkes process models.

- A pipelined hardware architecture for likelihood evaluation based on the proposed transformation for univariate Hawkes process models.
- The hyper-recursive likelihood evaluation strategy that eliminates the complex data dependency in likelihood evaluation for multivariate Hawkes process.
- A reconfigurable accelerator for intensity evaluation for multivariate Hawkes process models based on the hyper-recursive likelihood evaluation strategy.

To model the occurrence pattern of an event stream with a Hawkes process model, one needs to estimate the parameters using historical occurrence data. The parameter estimation problem of Hawkes processes can be solved by maximum likelihood estimation. Our achievements on parameter estimation are as follows.

- An efficient host-accelerator collaboration scheme for parameter estimation using the acceleration engine for likelihood evaluation.
- An empirical study of the accelerated parameter estimation solution.

6.1.2 Novel Predictive Model for Event Streams

The Hawkes point process model have limited predictive accuracy. Also, the model fitting procedures are time-consuming as the maximum likelihood evaluator has no analytical solution. To address the two problems, we propose a novel predictive model for event streams which works without point process models. Our achievements on the novel predictive model are as follows.

- An approximate form of the intensity function of the Hawkes point process.
- A novel predictive model for event streams that transforms an event stream prediction problem to a regression problem.
- Algorithms for fitting the proposed model to data.
- An empirical study of the proposed model regarding predictive accuracy and computational efficiency.

In the empirical study, a software system based on the proposed model reduces the prediction error by 3%–7% percent compared to a system based on the Hawkes process in four out of five benchmarks. A hardware-accelerated system based on the proposed model achieves a similar accuracy to the accelerated system based on the Hawkes process model, while the proposed system is 5–66 times faster in model fitting.

6.1.3 Reconfigurable Acceleration for Time Series

Measuring the correlation of time series is an important topic in the study of temporal data. However, the standard correlation measure for time series, the HAC estimation for long-run covariance matrices, is computationally demanding for large data. Our achievements on this problem are as follows.

- A pipeline-friendly HAC estimation algorithm derived from a mathematical specification.
- A pipelined accelerator for HAC estimation based on the proposed algorithm.
- An empirical study of the proposed accelerator on the FPGA platform.

In addition to the efficient measurement of the correlation between time series components, we also build pipelined accelerators for ordinal analysis to measure the statistical complexity of time series. Our achievements on the efficient ordinal analysis are as follows.

- A two-level ordinal pattern encoding method that avoids sequence sorting and reduces the amount of data transmission.
- A pipelined hardware architecture for the proposed encoding scheme.
- A case study where we apply the encoding engine to the calculation of permutation entropy.

6.2 Future Work

Although we have proposed solutions to accelerate the calculations in the predictive systems for event streams, we are aware of the limitations of our methods. In this section, we elaborate the limitations and discuss directions for future research.

6.2.1 Point Process Models with Non-Temporal Information

We propose acceleration solutions for the point process models in Chapter 3. Currently, we only analyse the temporal statistical pattern behind the data without being able to integrate the information from other aspects. However, in practical situations, random events usually come with other types of information. Future work in this direction includes the support for the following types of information:

- Geological information: instead of focusing exclusively on the temporal distribution of events, it is possible to develop accelerators for the models that involves the geological information of the events. A point process model with geological information may

generate predictions about the locations of future events. Accelerators for these models are particularly useful for seismology and public security.

- Numeric labels: instead of focusing on the events with categorical type labels, it is useful to consider the events with numeric labels. For instance, one may model the sharp changes in a stock price as a point process using our accelerator. The magnitude of change may provide important information about the future trend. However, our current models and accelerators are unable to take advantage of magnitude information.

6.2.2 Training Engines for Supervised Learning

We propose a training engine for regression in Chapter 4. Currently, we are unable to use it to train general-purpose regression models. The proposed training engine may produce weak models if the data redundancy is low because maintaining multiple versions of parameters with data partitioning may seriously reduce the convergence rate of the stochastic gradient algorithm [157]. However, it is possible to develop the training engine for general-purpose machine learning with a different optimisation strategy or a different model representation.

- Regarding the optimisation strategy, the proposed method has close relations to existing gradient-based optimisation methods. It is possible to improve the proposed data path to support more sophisticated optimisation strategies. In particular, it is promising to develop dataflow engines for batch gradient descent and mini-batch gradient descent. The batch gradient descent algorithm collects gradient information from all data points and update the parameters only once in each iteration. We may customise the proposed data path to collect gradient information without updating the parameters. The mini-batch gradient descent algorithm collects gradient information from a small batch of data and update the parameter accordingly. This method avoids frequent parameter update in stochastic gradient descent. As a result, it is possible to avoid the maintenance of multiple parameters using the mini-batch gradient descent method.
- Regarding the model representation, the proposed method only covers the linear model for regression. The linear regression model may be useful to obtain non-linear regression models or classifiers. In particular, it is promising to customise and improve the proposed data path to train Gaussian process regression models and Softmax classifiers. For Gaussian process regression, we may customise the proposed data path to compute the approximate inverse of the covariance matrix. For the Softmax classifier, we may replace the prediction function by the Softmax function so that the data path gains the ability to train classifiers with categorical labels.

6.2.3 Acceleration of Time Series Analysis Techniques

We propose hardware accelerators for two time series analysis techniques in Chapter 5. The two accelerators serve as analytic tool for time series data. Based on the two proposed techniques, it is promising to develop in the following two directions:

- An important application of HAC estimation is to measure the error of time series models. It is possible to use the result of HAC estimation to guide the actions of a real-time reconfigurable system. For instance, Guo *et al.* [158] propose the pipelined genetic propagation system which runs a hardware-oriented genetic algorithm on the FPGA platform. The system needs to reconfigure at run time when the results produced by the system are no longer useful. The HAC estimation engine may serve as the error measure of such systems that triggers run-time reconfiguration at proper times.
- The typical usage of ordinal patterns is to measure the complexity of time series. However, the ordinal patterns may also contain information that is valuable for making predictions about time series. For instance, Kalpakis *et al.* [159] develop classifiers with ordinal pattern analysis to predict the in-hospital mortality of brain-injury patients with high accuracy. It may be useful to build predictive systems based on the proposed ordinal pattern encoding engine for time-critical prediction problems.

6.2.4 Improving Reconfigurable Implementations and Their Evaluation

In this thesis, we propose reconfigurable acceleration solutions for event stream prediction. We put most of our efforts on the algorithmic and mathematical methods that facilitate reconfigurable computing. However, the proposed methods have not demonstrated their full potentiality as we have not thoroughly optimised and evaluated the implementations. Future work in this direction includes the following:

- Map the designs into recent FPGA devices. We use the Maxeler MAX3 acceleration platform in our experiments. The Xilinx Virtex-6 FPGA on the platform is no longer the latest FPGA technology. However, the proposed algorithms and mathematical methods work with other FPGA devices. Therefore, it is possible to improve the computational efficiency by mapping the proposed methods into recent and more advanced FPGA devices such as Xilinx Virtex UltraScale Plus devices [160] and Altera Stratix 10 devices [161].
- Explore the way to use the proposed methods with run-time reconfiguration [162]. The precision requirement of a model fitting algorithm may change over time. In particular, the initial stage of a model fitting process may tolerate low arithmetic precision [163]. At present, we use a single setting of arithmetic precision in each experimental system.

However, using run-time reconfiguration techniques, we may dynamically change the arithmetic precision in the model fitting process. Specifically, we may use low arithmetic precision at the beginning for higher efficiency, and reconfigure the system later to adopt high arithmetic precision to fine-tune the model.

- Evaluate trade-offs between computational efficiency, accuracy, resource usage, power and energy consumption of various implementations. Currently, we have only tested a limited number of implementations for each proposed method. However, an implementation for a practical problem needs to satisfy constraints regarding problem settings and hardware platforms. We may implement the proposed methods with different constraints to gain knowledge about how the factors change with each other. The knowledge on the trade-offs is particularly useful when tailoring and customising the hardware to meet specific goals at low cost.

6.3 Reflection

We conclude the thesis by considering the potential impact of the techniques discussed in this thesis. We have proposed the reconfigurable computing solutions for event stream modelling. The results suggest that reconfigurable computing may also be useful for the analysis of other types of data with time stamps, such as database logs.

Moreover, the most exciting and unexpected result in this research is the predictive accuracy of the novel predictive model for event streams proposed in Chapter 4. Our initial purpose was to approximate the Hawkes point process model to facilitate hardware acceleration, but we finally develop a standalone predictive model that works without point processes. In the experiments, the proposed system with counting vectors and support vector regression produces higher predictive accuracy than the Hawkes process model in the majority of test cases. The results suggest that it may be useful to think beyond the existing algorithms and propose new ones specifically tailored for hardware design. These algorithms may potentially bring improvements in statistical accuracy in addition to computational efficiency.

Bibliography

- [1] Y. Y. Kagan, “Statistical distributions of earthquake numbers: consequence of branching process,” *Geophysical Journal International*, vol. 180, no. 3, pp. 1313–1328, 2010.
- [2] Y. Ogata and K. Katsura, “Point-process models with linearly parametrized intensity for application to earthquake data,” *Journal of Applied Probability*, vol. 23, pp. 291–310, 1986.
- [3] E. Bacry, S. Delattre, M. Hoffmann, and J. Muzy, “Modelling microstructure noise with mutually exciting point processes,” *Quantitative Finance*, vol. 13, no. 1, pp. 65–77, 2013.
- [4] C. G. Bowsher, “Modelling security market events in continuous time: Intensity based, multivariate point process models,” *Journal of Econometrics*, vol. 141, no. 2, pp. 876–912, 2007.
- [5] E. Vinkovskaya, *A point process model for the dynamics of limit order books*. PhD thesis, Columbia University, 2014.
- [6] R. Dahlhaus, M. Eichler, and J. Sandkühler, “Identification of synaptic connections in neural ensembles by graphical models,” *Journal of neuroscience methods*, vol. 77, no. 1, pp. 93–107, 1997.
- [7] A. Hawkes, “Spectra of some self-exciting and mutually exciting point processes,” *Biometrika*, vol. 58, no. 1, pp. 83–90, 1971.
- [8] E. Bacry, I. Mastromatteo, and J.-F. Muzy, “Hawkes processes in finance,” *Market Microstructure and Liquidity*, vol. 1, no. 01, p. 1550005, 2015.
- [9] P. Hewlett, “Clustering of order arrivals, price impact and trade path optimisation,” in *Workshop on Financial Modeling with Jump processes, Ecole Polytechnique*, 2006.
- [10] A. Gunawardana, C. Meek, and P. Xu, “A model for temporal dependencies in event streams,” in *Advances in Neural Information Processing Systems 24* (J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, eds.), pp. 1962–1970, Curran Associates, Inc., 2011.

- [11] A. P. Parikh, A. Gunawardana, and C. Meek, “Conjoint modeling of temporal dependencies in event streams,” in *Proceedings of the UAI Bayesian Modelling Applications Workshop*, Citeseer, 2012.
- [12] H. Mannila, H. Toivonen, and A. I. Verkamo, “Discovery of frequent episodes in event sequences,” *Data Mining and Knowledge Discovery*, vol. 1, no. 3, pp. 259–289, 1997.
- [13] J. C. Weiss and D. Page, “Forest-based point process for event prediction from electronic health records,” in *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 547–562, Springer, 2013.
- [14] K. Heins and H. Stern, “A statistical model for event sequence data,” in *Proceedings of the International Conference on Artificial Intelligence and Statistics*, pp. 338–346, 2014.
- [15] P. S. Cowpertwait, “A generalized spatial–temporal model of rainfall based on a clustered point process,” in *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 450, pp. 163–175, The Royal Society, 1995.
- [16] P. S. Cowpertwait, “A spatial–temporal point process model of rainfall for the Thames catchment, UK,” *Journal of Hydrology*, vol. 330, no. 3, pp. 586–595, 2006.
- [17] W. Truccolo, U. T. Eden, M. R. Fellows, J. P. Donoghue, and E. N. Brown, “A point process framework for relating neural spiking activity to spiking history, neural ensemble, and extrinsic covariate effects,” *Journal of Neurophysiology*, vol. 93, no. 2, pp. 1074–1089, 2005.
- [18] G. O. Mohler, M. B. Short, P. J. Brantingham, F. P. Schoenberg, and G. E. Tita, “Self-exciting point process modeling of crime,” *Journal of the American Statistical Association*, vol. 106, no. 493, pp. 100–108, 2011.
- [19] R. Kobayashi and R. Lambiotte, “TiDeH: Time-dependent hawkes process for predicting retweet dynamics,” in *Proceedings of the International AAAI Conference on Web and Social Media*, 2016.
- [20] A. R. Brentnall, M. J. Crowder, and D. J. Hand, “A statistical model for the temporal pattern of individual automated teller machine withdrawals,” *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, vol. 57, no. 1, pp. 43–59, 2008.
- [21] G. M. Weiss and H. Hirsh, “Event prediction: learning from ambiguous examples,” *Proceedings of the Working Notes of the NIPS*, vol. 98, 1998.
- [22] L. Brown, N. Gans, A. Mandelbaum, A. Sakov, H. Shen, S. Zeltyn, and L. Zhao, “Statistical analysis of a telephone call center: A queueing-science perspective,” *Journal of the American Statistical Association*, vol. 100, no. 469, pp. 36–50, 2005.

- [23] R. Bekkerman, M. Bilenko, and J. Langford, *Scaling up machine learning: Parallel and distributed approaches*. Cambridge University Press, 2011.
- [24] K. Golshan, *Physical design essentials: An ASIC design implementation perspective*. Springer Science & Business Media, 2007.
- [25] I. Kuon, R. Tessier, and J. Rose, “FPGA architecture: Survey and challenges,” *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 2, pp. 135–253, 2008.
- [26] D. Chen, J. Cong, and P. Pan, “FPGA design automation: A survey,” *Foundations and Trends in Electronic Design Automation*, vol. 1, no. 3, pp. 139–169, 2006.
- [27] P. J. Ashenden, *The designer’s guide to VHDL*, vol. 3. Morgan Kaufmann, 2010.
- [28] D. Thomas and P. Moorby, *The Verilog Hardware Description Language*. Springer Science & Business Media, 2008.
- [29] Xilinx, “Vivado design suite user guide: High-level synthesis.” https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug902-vivado-high-level-synthesis.pdf, April 2016.
- [30] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, “From OpenCL to high-performance hardware on FPGAs,” in *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 531–534, IEEE, 2012.
- [31] T. Becker, O. Mencer, and G. Gaydadjiev, “Spatial programming with openspl,” in *FPGAs for Software Programmers*, pp. 81–95, Springer, 2016.
- [32] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, “Legup: high-level synthesis for FPGA-based processor/accelerator systems,” in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 33–36, ACM, 2011.
- [33] K. Compton and S. Hauck, “Reconfigurable computing: a survey of systems and software,” *ACM Computing Surveys (csuR)*, vol. 34, no. 2, pp. 171–210, 2002.
- [34] J. M. Cardoso, P. C. Diniz, and M. Weinhardt, “Compiling for reconfigurable computing: A survey,” *ACM Computing Surveys (CSUR)*, vol. 42, no. 4, p. 13, 2010.
- [35] O. Pell and V. Averbukh, “Maximum performance computing with dataflow engines,” *Computing in Science & Engineering*, vol. 14, no. 4, pp. 98–103, 2012.

- [36] D. Daley and D. Vere-Jones, *An introduction to the theory of point processes*. Springer, 2007.
- [37] J. F. C. Kingman, *Poisson processes*. Wiley Online Library, 1993.
- [38] F. P. Schoenberg, D. R. Brillinger, and P. Guttorp, “Point processes, spatial–temporal,” *Encyclopedia of Environmetrics*, 2002.
- [39] P. J. Diggle, *Statistical analysis of spatial and spatio-temporal point patterns*. CRC Press, 2013.
- [40] D. R. Cox, “Some statistical methods connected with series of events,” *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 129–164, 1955.
- [41] J. F. C. Kingman, “On doubly stochastic poisson processes,” *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 60, p. 923930, Oct 2008.
- [42] J. P. Cunningham, K. V. Shenoy, and M. Sahani, “Fast Gaussian process methods for point process intensity estimation,” in *Proceedings of the International conference on Machine learning*, pp. 192–199, ACM, 2008.
- [43] S. Linderman and R. Adams, “Discovering latent network structure in point process data,” in *Proceedings of the International Conference on Machine Learning*, pp. 1413–1421, 2014.
- [44] F. Lorenzen, *Analysis of order clustering using high frequency data: A point process approach*. PhD thesis, Tilburg School of Economics and Management Finance Department, 2012.
- [45] Y. Ogata, “On Lewis’ simulation method for point processes,” *IEEE Transactions on Information Theory*, vol. 27, no. 1, pp. 23–31, 1981.
- [46] Y. Ogata and H. Akaike, “On linear intensity models for mixed doubly stochastic poisson and self-exciting point processes,” in *Selected Papers of Hirotugu Akaike*, pp. 269–274, Springer, 1982.
- [47] T. Ozaki, “Maximum likelihood estimation of hawkes’ self-exciting point processes,” *Annals of the Institute of Statistical Mathematics*, vol. 31, no. 1, pp. 145–155, 1979.
- [48] I. M. Toke, “Market making in an order book model and its impact on the spread,” in *Econophysics of Order-driven Markets*, pp. 49–64, Springer, 2011.
- [49] Y. Y. Kagan and L. Knopoff, “Statistical short-term earthquake prediction,” *Science*, vol. 236, no. 4808, pp. 1563–1567, 1987.

- [50] J. F. Olson and K. M. Carley, “Exact and approximate em estimation of mutually exciting hawkes processes,” *Statistical Inference for Stochastic Processes*, vol. 16, no. 1, pp. 63–80, 2013.
- [51] J. Da Fonseca and R. Zaatour, “Hawkes process: fast calibration, application to trade clustering, and diffusive limit,” *Journal of Futures Markets*, vol. 34, no. 6, pp. 548–579, 2014.
- [52] P. F. Halpin and P. De Boeck, “Modelling dyadic interaction with hawkes processes,” *Psychometrika*, vol. 78, no. 4, pp. 793–814, 2013.
- [53] T. Liniger, *Multivariate Hawkes Processes*. PhD thesis, ETH Zurich, 2009.
- [54] A. H. Zadeh and R. Sharda, “Modeling brand post popularity dynamics in online social networks,” *Decision Support Systems*, vol. 65, pp. 59–68, 2014.
- [55] J. A. Nelder and R. Mead, “A simplex method for function minimization,” *The computer journal*, vol. 7, no. 4, pp. 308–313, 1965.
- [56] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright, “Convergence properties of the nelder–mead simplex method in low dimensions,” *SIAM Journal on optimization*, vol. 9, no. 1, pp. 112–147, 1998.
- [57] A. R. Hall, *Generalized Method of Moments. Advanced Texts in Econometrics Series*. Oxford University Press, 2005.
- [58] R. Storn and K. Price, “Differential evolution—a simple and efficient heuristic for global optimization over continuous spaces,” *Journal of global optimization*, vol. 11, no. 4, pp. 341–359, 1997.
- [59] C.-R. Hwang, “Simulated annealing: theory and applications,” *Acta Applicandae Mathematicae*, vol. 12, no. 1, pp. 108–111, 1988.
- [60] B. M. Lapham, *Hawkes processes and some financial applications*. PhD thesis, University of Cape Town, 2014.
- [61] C. Guo, H. Fu, and W. Luk, “A fully-pipelined expectation-maximization engine for Gaussian mixture models,” in *Proceedings of the International Conference on Field-Programmable Technology*, IEEE, 2012.
- [62] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*, vol. 1. Springer series in statistics Springer, Berlin, 2001.
- [63] K. Murphy, *Machine learning: a probabilistic perspective*. MIT Press, 2012.

- [64] P. Domingos, “A few useful things to know about machine learning,” *Communications of the ACM*, vol. 55, no. 10, pp. 78–87, 2012.
- [65] O. Bousquet and L. Bottou, “The tradeoffs of large scale learning,” in *Advances in neural information processing systems*, pp. 161–168, 2008.
- [66] J. Snymann, *Practical mathematical optimization: an introduction to basic optimization theory and classical and new gradient-based algorithms*, vol. 97. Springer Science & Business Media, 2005.
- [67] D. R. Wilson and T. R. Martinez, “The general inefficiency of batch training for gradient descent learning,” *Neural Networks*, vol. 16, no. 10, pp. 1429–1451, 2003.
- [68] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, “Learning to rank using gradient descent,” in *Proceedings of the International Conference on Machine learning*, pp. 89–96, ACM, 2005.
- [69] L. Bottou, “Stochastic gradient learning in neural networks,” *Proceedings of the International Conference Neural Networks & their Applications*, vol. 91, no. 8, 1991.
- [70] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter, “Pegasos: Primal estimated sub-gradient solver for SVM,” *Mathematical programming*, vol. 127, no. 1, pp. 3–30, 2011.
- [71] T. Schaul, S. Zhang, and Y. LeCun, “No more pesky learning rates,” in *Proceedings of the International Conference on Machine Learning*, vol. 28, pp. 343–351, 2013.
- [72] Z. Baker and V. Prasanna, “Efficient hardware data mining with the apriori algorithm on FPGAs,” in *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*, pp. 3–12, IEEE, 2005.
- [73] T. Saegusa and T. Maruyama, “An FPGA implementation of k-means clustering for color images based on kd-tree,” in *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 1–6, IEEE, 2006.
- [74] P. Moerland and E. Fiesler, “Neural network adaptations to hardware implementations,” tech. rep., The Idiap Research Institute, Switzerland, 1997.
- [75] J. Misra and I. Saha, “Artificial neural networks in hardware: A survey of two decades of progress,” *Neurocomputing*, vol. 74, no. 1, pp. 239–255, 2010.
- [76] A. R. Omondi and J. C. Rajapakse, *FPGA implementations of neural networks*, vol. 365. Springer, 2006.

- [77] S. G. Merchant and G. D. Peterson, “Evolvable block-based neural network design for applications in dynamic environments,” *VLSI Design*, vol. 2010, p. 4, 2010.
- [78] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, “Efficient backprop,” in *Neural networks: Tricks of the trade*, pp. 9–48, Springer, 2012.
- [79] J. G. Eldredge and B. L. Hutchings, “Density enhancement of a neural network using FPGAs and run-time reconfiguration,” in *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 180–188, IEEE, 1994.
- [80] K. Paul and S. Rajopadhye, “Back-propagation algorithm achieving 5 GOPs on the Virtex-E,” in *FPGA Implementations of Neural Networks*, pp. 137–165, Springer, 2006.
- [81] Y. Maeda and T. Tada, “FPGA implementation of a pulse density neural network with learning ability using simultaneous perturbation,” *IEEE Transactions on Neural Networks*, vol. 14, no. 3, pp. 688–695, 2003.
- [82] Y. Maeda, H. Hirano, and Y. Kanata, “A learning rule of neural networks via simultaneous perturbation and its hardware implementation,” *Neural Networks*, vol. 8, no. 2, pp. 251–259, 1995.
- [83] S.-W. Moon and S.-G. Kong, “Block-based neural networks,” *IEEE Transactions on Neural Networks*, vol. 12, no. 2, pp. 307–317, 2001.
- [84] W. Jiang, S. G. Kong, and G. D. Peterson, “ECG signal classification using block-based neural networks,” in *Proceedings of the IEEE International Joint Conference on Neural Networks*, vol. 1, pp. 326–331, IEEE, 2005.
- [85] W. Jiang, S. G. Kong, and G. D. Peterson, “Continuous heartbeat monitoring using evolvable block-based neural networks,” in *Proceedings of the IEEE International Joint Conference on Neural Networks*, pp. 1950–1957, IEEE, 2006.
- [86] W. Jiang and S. G. Kong, “Block-based neural networks for personalized ECG signal classification,” *IEEE Transactions on Neural Networks*, vol. 18, no. 6, pp. 1750–1761, 2007.
- [87] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, pp. 1097–1105, 2012.
- [88] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, “A high performance FPGA-based accelerator for large-scale convolutional neural networks,” in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2016.

- [89] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing FPGA-based accelerator design for deep convolutional neural networks,” in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170, ACM, 2015.
- [90] S. I. Venieris and C.-S. Bouganis, “fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs,” in *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*, 2016.
- [91] W. Zhao, H. Fu, W. Luk, T. Yu, S. Wang, B. Feng, Y. Ma, and G. Yang, “An FPGA-based framework for training convolutional neural networks,” in *Proceedings of the International Conference on Application-specific Systems, Architectures and Processors*, 2016.
- [92] B. E. Boser, I. M. Guyon, and V. N. Vapnik, “A training algorithm for optimal margin classifiers,” in *Proceedings of the fifth annual workshop on Computational learning theory*, pp. 144–152, ACM, 1992.
- [93] A. J. Smola and B. Schölkopf, “A tutorial on support vector regression,” *Statistics and computing*, vol. 14, no. 3, pp. 199–222, 2004.
- [94] A. Smola and V. Vapnik, “Support vector regression machines,” *Advances in neural information processing systems*, vol. 9, pp. 155–161, 1997.
- [95] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. P. Graf, “A massively parallel FPGA-based coprocessor for support vector machines,” in *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines*, pp. 115–122, IEEE, 2009.
- [96] D. Anguita, A. Boni, and S. Ridella, “A digital architecture for support vector machines: theory, algorithm, and FPGA implementation,” *IEEE Transactions on Neural Networks*, vol. 14, no. 5, pp. 993–1009, 2003.
- [97] D. Anguita, A. Ghio, S. Pisciutta, and S. Ridella, “A hardware-friendly support vector machine for embedded automotive applications,” in *Proceedings of the International Joint Conference on Neural Networks*, pp. 1360–1364, 2007.
- [98] J. Platt *et al.*, “Sequential minimal optimization: A fast algorithm for training support vector machines,” tech. rep., Microsoft research, 1998.
- [99] J. D. Hamilton, *Time series analysis*. Cambridge University Press, 1994.
- [100] A. Smith and E. Brown, “Estimating a state-space model from point process observations,” *Neural Computation*, vol. 15, no. 5, pp. 965–991, 2003.

- [101] B. M. Yu, J. P. Cunningham, G. Santhanam, S. I. Ryu, K. V. Shenoy, and M. Sahani, "Gaussian-process factor analysis for low-dimensional single-trial analysis of neural population activity," in *Advances in Neural Information Processing Systems 21* (D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, eds.), pp. 1881–1888, Curran Associates, Inc., 2009.
- [102] D. Sart, A. Mueen, W. Najjar, E. Keogh, and V. Niennattrakul, "Accelerating dynamic time warping subsequence search with GPUs and FPGAs," in *Proceedings of the International Conference on Data Mining*, pp. 1001–1006, 2010.
- [103] Z. Wang, S. Huang, L. Wang, H. Li, Y. Wang, and H. Yang, "Accelerating subsequence similarity search based on dynamic time warping distance with FPGA," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2013.
- [104] W. K. Newey and K. D. West, "A simple, positive semi-definite, heteroskedasticity and autocorrelation consistent covariance matrix," *Econometrica: Journal of the Econometric Society*, vol. 55, pp. 703–708, 1987.
- [105] W. K. Newey and K. D. West, "Automatic lag selection in covariance matrix estimation," *Review of Economic Studies*, vol. 61, pp. 631–653, 1994.
- [106] D. Andrews, "Heteroskedasticity and autocorrelation consistent covariance matrix estimation," *Econometrica: Journal of the Econometric Society*, vol. 59, pp. 817–858, 1991.
- [107] C. Bandt and B. Pompe, "Permutation entropy: a natural complexity measure for time series," *Physical Review Letters*, vol. 88, no. 17, p. 174102, 2002.
- [108] M. Zanin, L. Zunino, O. A. Rosso, and D. Papo, "Permutation entropy and its main biomedical and econophysics applications: a review," *Entropy*, vol. 14, no. 8, pp. 1553–1577, 2012.
- [109] P. A. Lewis and G. S. Shedler, "Simulation of nonhomogeneous poisson processes by thinning," *Naval Research Logistics Quarterly*, vol. 26, no. 3, pp. 403–413, 1979.
- [110] D. R. Cox, "Partial likelihood," *Biometrika*, vol. 62, no. 2, pp. 269–276, 1975.
- [111] N. Weaver, "Retiming, repipelining, and c-slow retiming," in *Reconfigurable Computing: the Theory and Practice of FPGA-based Computation* (S. Hauck and A. DeHon, eds.), Morgan Kaufmann, 2007.
- [112] I. Rubin, "Regular point processes and their detection," *IEEE Transactions on Information Theory*, vol. 18, no. 5, pp. 547–557, 1972.

- [113] S. Kirkpatrick, “Optimization by simulated annealing: Quantitative studies,” *Journal of statistical physics*, vol. 34, no. 5-6, pp. 975–986, 1984.
- [114] P. J. Van Laarhoven and E. H. Aarts, *Simulated annealing*. Springer, 1987.
- [115] L. Duczmal and R. Assuncao, “A simulated annealing strategy for the detection of arbitrarily shaped spatial clusters,” *Computational Statistics & Data Analysis*, vol. 45, no. 2, pp. 269–286, 2004.
- [116] I. H. Osman, “Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem,” *Annals of Operations Research*, vol. 41, no. 4, pp. 421–451, 1993.
- [117] S. Anily and A. Federgruen, “Simulated annealing methods with general acceptance probabilities,” *Journal of Applied Probability*, vol. 24, no. 3, pp. 657–667, 1987.
- [118] L. Breiman, “Statistical modeling: The two cultures (with comments and a rejoinder by the author),” *Statistical Science*, vol. 16, no. 3, pp. 199–231, 2001.
- [119] Government of The District of Columbia, “The district of columbia: Open data catalog,” <http://data.octo.dc.gov/>.
- [120] D. P. Schaff and F. Waldhauser, “Waveform cross-correlation-based differential travel-time measurements at the northern california seismic network,” *Bulletin of the Seismological Society of America*, vol. 95, no. 6, pp. 2446–2461, 2005.
- [121] F. Waldhauser and W. L. Ellsworth, “A double-difference earthquake location algorithm: Method and application to the northern hayward fault, california,” *Bulletin of the Seismological Society of America*, vol. 90, no. 6, pp. 1353–1368, 2000.
- [122] F. Waldhauser and D. P. Schaff, “Large-scale relocation of two decades of northern california seismicity using cross-correlation and double-difference methods,” *Journal of Geophysical Research: Solid Earth*, vol. 113, no. B8, 2008.
- [123] M. F. Arlitt and C. L. Williamson, “Web server workload characterization: The search for invariants,” in *ACM SIGMETRICS Performance Evaluation Review*, vol. 24, pp. 126–137, ACM, 1996.
- [124] H. J. S. Smith, “On the integration of discontinuous functions,” *Proceedings of the London Mathematical Society*, vol. 1, no. 1, pp. 140–153, 1874.
- [125] C.-C. Chang and C.-J. Lin, “LIBSVM: A library for support vector machines,” *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011.

- [126] A. Bordes, S. Ertekin, J. Weston, and L. Bottou, “Fast kernel classifiers with online and active learning,” *Journal of Machine Learning Research*, vol. 6, no. Sep, pp. 1579–1619, 2005.
- [127] J. Kane, R. Hernandez, and Q. Yang, “A reconfigurable multiclass support vector machine architecture for real-time embedded systems classification,” in *Proceedings of the Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pp. 244–251, IEEE, 2015.
- [128] L. Bauwens and N. Hautsch, “Modelling financial high frequency data using point processes,” in *Handbook of financial time series*, pp. 953–979, Springer, 2009.
- [129] N. Jegadeesh and S. Titman, “Returns to buying winners and selling losers: Implications for stock market efficiency,” *The Journal of Finance*, vol. 48, no. 1, pp. 65–91, 1993.
- [130] T. Bollerslev, G. Tauchen, and H. Zhou, “Expected stock returns and variance risk premia,” *Review of Financial Studies*, vol. 22, no. 11, pp. 4463–4492, 2009.
- [131] G. Bekaert, C. Harvey, C. Lundblad, and S. Siegel, “What segments equity markets?,” *Review of Financial Studies*, vol. 24, no. 12, pp. 3841–3890, 2011.
- [132] A. Zeileis, “Econometric computing with HC and HAC covariance matrix estimators,” tech. rep., Institut für Statistik und Mathematik, WU Vienna University of Economics and Business, 2004.
- [133] A. Cottrell and R. Lucchetti, “Gretl user’s guide.” Distributed with the Gretl library, 2012.
- [134] C. Y. Lin, H. K. So, and P. H. W. Leong, “A model for matrix multiplication performance on FPGAs,” in *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 305–310, 2011.
- [135] A. C. Jacob, J. D. Buhler, and R. D. Chamberlain, “Rapid RNA folding: analysis and acceleration of the zucker recurrence,” in *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*, pp. 87–94, 2010.
- [136] J. Harkins, T. El-Ghazawi, E. El-Araby, and M. Huang, “Performance of sorting algorithms on the src 6 reconfigurable computer,” in *Proceedings of the International Conference on Field-Programmable Technology*, pp. 295–296, 2005.
- [137] J. Martinez, R. Cumplido, and C. Feregrino, “An FPGA-based parallel sorting architecture for the burrows wheeler transform,” in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*, 2005.

- [138] D. Mihhailov, V. Sklyarov, I. Skliarova, and A. Sudnitson, “Parallel FPGA-based implementation of recursive sorting algorithms,” in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*, 2010.
- [139] V. Sklyarov, I. Skliarova, D. Mihhailov, and A. Sudnitson, “Implementation in FPGA of address-based data sorting,” in *Proceedings of the International Conference on Field Programmable Logic and Applications*, 2011.
- [140] D. H. Lehmer, “Teaching combinatorial tricks to a computer,” in *Proc. Sympos. Appl. Math. Combinatorial Analysis*, vol. 10, pp. 179–193, 1960.
- [141] 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.
- [142] M. Bóna, *Combinatorics of permutations*. CRC Press, 2012.
- [143] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*. Boston: Addison-Wesley, 1998.
- [144] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT Press, 2001.
- [145] 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*, vol. 1142 of *Lecture Notes in Computer Science*, pp. 230–236, Springer Berlin Heidelberg, 1996.
- [146] F. de Dinechin, V. Lefevre, *et al.*, “Constant multipliers for FPGAs,” tech. rep., ENS de Lyon, 2000.
- [147] 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.
- [148] C. E. Shannon, “A mathematical theory of communication,” *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.
- [149] Y. Cao, W.-w. Tung, J. Gao, V. Protopopescu, L. Hively, *et al.*, “Detecting dynamical changes in time series using the permutation entropy,” *Physical Review Series E*, vol. 70, pp. 046217–046217, 2004.
- [150] 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.

- [151] E. Olofsen, J. Sleight, 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.
- [152] 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.
- [153] 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.
- [154] 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.
- [155] 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.
- [156] 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.
- [157] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, “Parallelized stochastic gradient descent,” in *Advances in neural information processing systems*, pp. 2595–2603, 2010.
- [158] L. Guo, C. Guo, D. B. Thomas, and W. Luk, “Pipelined genetic propagation,” in *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machine*, pp. 103–110, IEEE, 2015.
- [159] K. Kalpakis, S. Yang, P. F. Hu, C. F. Mackenzie, L. G. Stansbury, D. M. Stein, and T. M. Scalea, “Outcome prediction for patients with severe traumatic brain injury using permutation entropy analysis of electronic vital signs data,” in *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, pp. 415–426, Springer, 2012.
- [160] S. Leibson and N. Mehta, “Xilinx ultrascale: The next-generation architecture for your next-generation architecture white paper.” http://www.xilinx.com/support/documentation/white_papers/wp435-Xilinx-UltraScale.pdf, May 2014.
- [161] A. Davidson, “A new FPGA architecture and leading-edge FinFET process technology promise to meet next generation system requirements.” <https://>

[//www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01220-hyperflex-architecture-fpga-socs.pdf](http://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01220-hyperflex-architecture-fpga-socs.pdf), June 2015.

- [162] W. Luk, N. Shirazi, S. Guo, and P. Y. Cheung, “Pipeline morphing and virtual pipelines,” in *International Workshop on Field Programmable Logic and Applications*, pp. 111–120, Springer, 1997.
- [163] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep learning with limited numerical precision,” in *Proceedings of the International Conference on Machine Learning*, 2015.

Appendix A

General Settings of Experiments

The hardware designs are implemented and tested on a Maxeler MAX3 acceleration platform. The acceleration platform is installed on a dual CPU server with 48GB DDR3 memory. The server has two Intel Xeon X5650 CPUs running at 2.67GHz. Each CPU contains six physical cores. With the Intel hyper-threading technology, one physical core appears as two logical cores to the operating system. Therefore, the two CPUs offer 12 physical cores which appear as 24 logical cores. The CPUs are based on the 32nm transistor technology which is more advanced than the 40nm technology of the Virtex-6 SX475T FPGA on the acceleration platform.

The CPU implementations in the experiments run exclusively on the server with the OpenMP library. To make a fair comparison, we perform a series of optimisations on this implementation, including avoiding redundant memory access and optimising the parallelisation scheme. The CPU implementations and the host code for FPGA implementations are written in the C programming language. The C code is compiled using the Intel C/C++ compiler 12.1.4 with the ‘-O3’ option for the highest compilation optimisation level. To ensure the vectorisation of the CPU code, a serial version for each piece of code is produced with vectorisation disabled. The largest data set in each experiment is tested to ensure the execution time of the vectorised version is less than that of the serial version.

The experiment for each data set is repeated for multiple times. At the end of each run, the experimental system checks whether the total execution time exceeds 3600 seconds. If so, the experiment process terminates immediately. The average execution time is calculated by dividing the total execution time by the number of runs. The caches are kept warm as the experiment process does not perform a cold reset at the end of each run. Therefore, the execution time of the CPU implementations may be underestimated, which results in pessimistic estimates of speedups.