TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Implementation of Mutual Information and Transfer Entropy Algorithms with FPGA-based SuperComputer

---

Author:
Konstantinos IORDANOU

Supervisor:
Prof. Apostolos DOLLAS

A thesis submitted in fulfillment of the requirements
for the degree of Diploma in Electrical and Computer Engineering

in the

Microprocessor and Hardware Laboratory
School of Electrical and Computer Engineering

June 2017

"People who are more than casually interested in computers should have at least some idea of what the underlying hardware is like.Otherwise the programs they write will be pretty weird…"

Donald Ervin Knuth

Technical University of Crete

Abstract

Microprocessor and Hardware Laboratory
School of Electrical and Computer Engineering

Diploma in Electrical and Computer Engineering

Implementation of Mutual Information and Transfer Entropy Algorithms with FPGA-based SuperComputer

by Konstantinos IORDANOU

It is widely known that contemporary applications are bounded by massive computational demands. With conventional CPUs falling out of favor due to their limitations, the industry of Hybrid-SuperComputers using reconfigurable logic which is a growing field in the area of Computer Systems. This thesis explores the Convey Computer, and more specifically the platform HC-2ex which is a hybrid platform with increased computational capacity as well as a combination of a high-bandwidth memory interface with an architecture featuring multiple levels of computational parallelism. This platform selected in order to efficiently map computationally intensive algorithms in modern hardware. We address two challenging problems within this framework, the first being time-series analysis by focusing on the calculation of the Mutual Information (MI) statistical value and the second being the Transfer Entropy (TE) statistical value between two time-series. The problems of Mutual Information and Transfer Entropy respectively, have been addressed by the research community for low-precision arithmetic applications, but the performance of these algorithms have not been evaluated on platforms like Convey Computer. This is the first work to extensively study of using this platform, by identifying the pros and cons of Convey Computer with computationally intensive algorithms as well as describing how these algorithms can efficiently utilized. In terms of result, Mutual Information and Transfer Entropy implementations compared with implemented architectures on other platforms like Maxeler. Compared to the reference software, the implementation of MI algorithm yielded 13x speedup as well as the implementation of TE yielded 15x speedup for high dimensional data using 32-bit precision arithmetic on Convey HC-2ex.

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Apostolos Dollas for his patience, motivation, immense knowledge and for our long discussions about my future steps. Also I am deeply grateful to him for inspiring me throughout the first years of my studies, for providing me the right motivations to follow my dreams as well as for introducing me to the world of computer architecture.

I would like to thank Prof. Minos Garofalakis and Prof. Ioannis Papaefstathiou for accepting to be in my committee.

Furthermore I would like to thank Pavlos Malakonakis for his technical and personal support. He always finding a way to point me to the right direction in my first steps in research.

Also I would like to thank Dr.Gregory Chrysos for his technical support during this thesis and his valuable knowledge and guidance related to logic circuits and FPGA-prototyping.

In addition, I would like to thank my friend Nikolaos Kyparissas for the perfect cooperation we had as students at Technical University of Crete.

A big thank to my parents for everything they offered to me, George and Konstantina as well as to my younger sister, Helen for supporting my decisions and for their patience in my concerns.

I am thankful also to my dear Vasiliki Kalliga for supporting every step in my life and for her motivation to pursue my dreams...

Contents

List of Figures

List of Tables

Dedicated to George, Konstantina, Helen and Vasiliki…

Chapter 1

Introduction

It is widely known that contemporary applications are bounded by massive computational demands. With conventional CPUs falling out of favor due to their limitations, the industry of Hybrid-SuperComputers using reconfigurable logic which is a growing field in the area of Computer Systems. This thesis explores the Convey Computer, and more specifically the platform HC-2ex which is a hybrid platform with increased computational capacity as well as a combination of a high-bandwidth memory interface with an architecture featuring multiple levels of computational parallelism. This platform selected in order to efficiently map computationally intensive algorithms in modern hardware.

The idea of entropy of random variables and processes which proposed by Claude Shannon, signified the beginnings of information theory and the modern age of ergodic theory. It is a foregone conclusion that entropy and related information measures, provide useful descriptions of the long term behavior of random processes and this behavior is a key factor in developing the coding theorems of information theory. There are various notions of entropy for random variables, vectors, processes and dynamical systems. More specific the statistical analysis of financial time series has resulted in the possibility of predicting unobserved values of time series. The interest in forecasting future stock values does not only stem from the fact that it is a very challenging research problem but also from the people's desire to make money. From an algorithmic perspective, identifying time series with the highest probability to reveal information about themselves and most importantly about other time series is a very challenging problem, especially when time is critical importance, such as risk analysis. In this thesis our aim is to accelerate the computation of Mutual Information as well as the computation of Transfer Entropy statistical measures between two time-series to identify dependence.

1.1 Thesis Contribution

In general, Information Theory studies the quantification, storage and communication of information. It was originally proposed by Claude E. Shannon in 1948 to find fundamental limits on signal processing and communication operations such as data compression. This theory has found applications in many other areas, including statistical inference, natural language processing, cryptography, neurobiology, model selection in ecology, thermal physics, quantum computing, linguistics, plagiarism detection, pattern recognition and anomaly detection. A key measure in information theory is entropy. Some others important measures in information theory are mutual information,

channel capacity, relative entropy and error exponents. Applications of fundamental topics of information theory include lossless data compression, lossy data compression and channel coding. Considering the above the Mutual Information and Transfer Entropy computations have been proven redemptive for several scientific fields with massive amounts of data to be processed and high computational demands. The problem of accelerating these computations has been tackled by several research groups. In this work we utilize the HC-2ex Hybrid-SuperComputer provided by Convey Computers to map these two algorithms and accelerate them. To the best of our knowledge this is the first work that address the whole problem of designing two architectures, the first being the calculation of Mutual Information statistical value and the second being the calculation of Transfer Entropy statistical value from the beginning until the end using the Convey computer. We decided to create two architectures (MI and TE) that produce the exact same solution as the reference software on high dimensional datasets using the Convey Hybrid Core platform which is a heterogeneous computing system, efficiently pairing traditional general purpose CPUs with reconfigurable hardware coprocessor units, in order to increase application performance beyond what is typically possible in a standard x86 system. Hence, we considered the problems of calculating the Mutual Information and Tranfer Entropy statistic values between time-series and achieved 13x and 15x speedup respectively, compared to the respective software implementations.

1.2 Thesis outline

Chapter 2 initially introduces us to the Convey System platform and present the theoritical background of the algorithms studied in this thesis. The goal of these descriptions is to provide a deep comprehension regarding the outcomes of modifying our algorithms based on the properties of the Convey platform. Chapter 3 presents all the related works that have been proposed for the acceleration of the Mutual information and Transfer Entropy algorithms with the use of hardware-based platforms or multicore processors. Chapter 4 analyses the two algorithms from the hardware designer's perspective. Also in this chapter comes up the methology followed to efficiently map these to algorithms to the FPGA's of the Hybrid platform as well as how these algorithms modified in order to become integrated to the Convey System. In Chapter 5 we describe the verification process followed during the implementation and present a series of performance comparisons between our implementations, the reference software and implementations on Maxeler platforms. In chapter 6 a conclusion about the presented work is provided in order to give the proper and worth considering directions for future work.

Chapter 2

Background

2.1 Convey System Description

In the race of developing novel chip technologies to accelerate special-purpose designs, several companies have attempted to propose specialized solutions. One of these enterprises is the Convey Computer. The Convey Computer Architecture integrates two types of processor architecture in one system: the Intel 64 architecture implemented by an Intel processor and a reconfigurable architecture implemented as a co-processor designed and implemented by Convey.

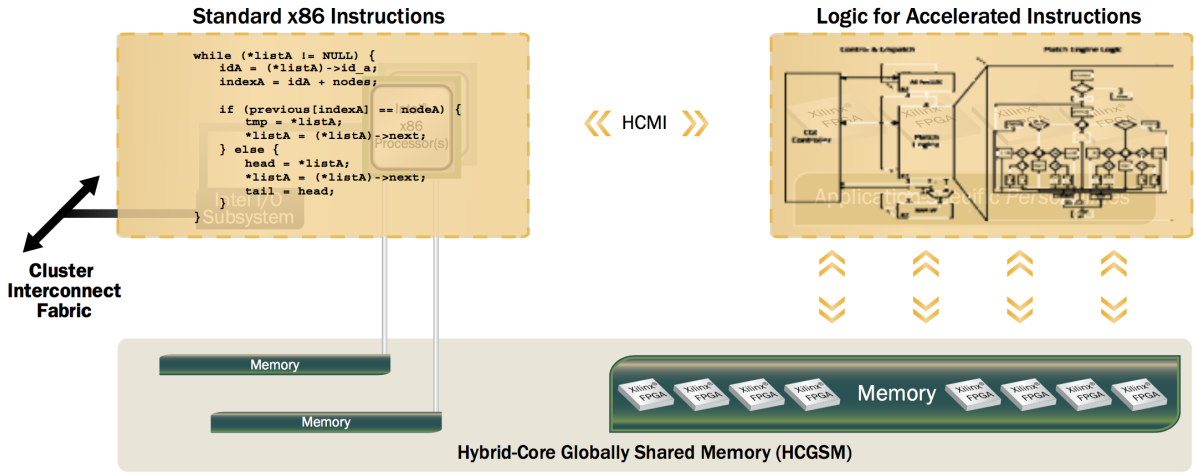

FIGURE 2.1: Hybrid-Core Computing as appeared to an application

The ability to support different instruction sets in a common hardware platform allows the implementation of new instruction sets. An application executable contains both Intel and co-processor instructions, and those instructions execute in a common, coherent address space. The huge reduction in implementation time makes it practical to develop instruction sets tailored to specific applications and algorithms.

2.1.1 System Architecture

Convey Hybrid Core systems utilize a commodity motherboard that includes an Intel 64 host processor and standard Intel I/O chipset, along with a reconfigurable co-processor based on FPGA technology. This co-processor can be reloaded dynamically

while the system is running with instructions that are optimized for different workloads. It includes its own high bandwidth memory subsystem. Co-processor instructions can be considered as extensions to the Intel instructions set. The co-processor supports multiple instruction sets referred to as personalities.

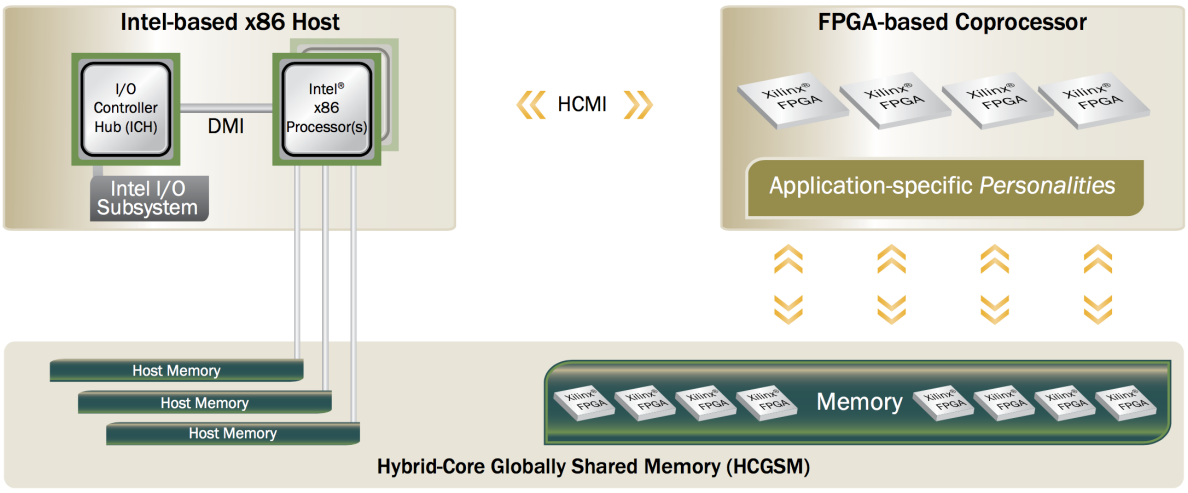

FIGURE 2.2: Convey Hybrid Core System Diagram

2.1.2 Co-processor

At the other end of the spectrum the co-processor has three major sets of components referred to as the Application Engine Hub (AEH), the Memory Controllers (MCs) and the Application Engines (AEs). To begin with, the AEH is the central hub for the co-processor. More specific it implements the interface to the host processor and to the Intel I/O chipset, fetches and decodes instructions, and executes scalar instructions. As a result it processes coherence and data requests from the host processor as well as routing requests for addresses in co-processor memory to the MCs. Scalar instructions are executed in the AEH, while extended instructions are passed to the AEs for the execution. In order to support the bandwidth, 8 Memory Controllers support a total 16 DDR2 memory channels providing an aggregate of over 80 GB/sec of bandwidth to ECC protected memory. The memory controllers support standard DIMMs as well as Convey designed Scatter-Gather DIMMs. The main task is of these kind of DIMMs is to translate virtual to physical addresses on behalf of the AEs. Together the AEH and the MCs implement features that are present in all personalities. This ensures that important features such as memory protection, access to co-processor memory or the communication with the host processor are always available. The Application Engines (AEs) implement the extended instructions that deliver performance for a personality. In general the Application Engines (AEs) are the heart of the co-processor and implement the extended instructions that deliver performance for a personality. There are four AEs, connected to the AEH by a command bus that transfers opcodes and scalar operands, and connected via a network of point-to-point links to each of the memory controllers. Each AE instruction is passed to all four AEs. Although it is known that the clock rate for the FPGAs used to implement the co-processor is lower than that of a

commodity processor, each AE will have many functional units operating in parallel. This high degree of parallelism gives an advantage to the co-processor's performance.

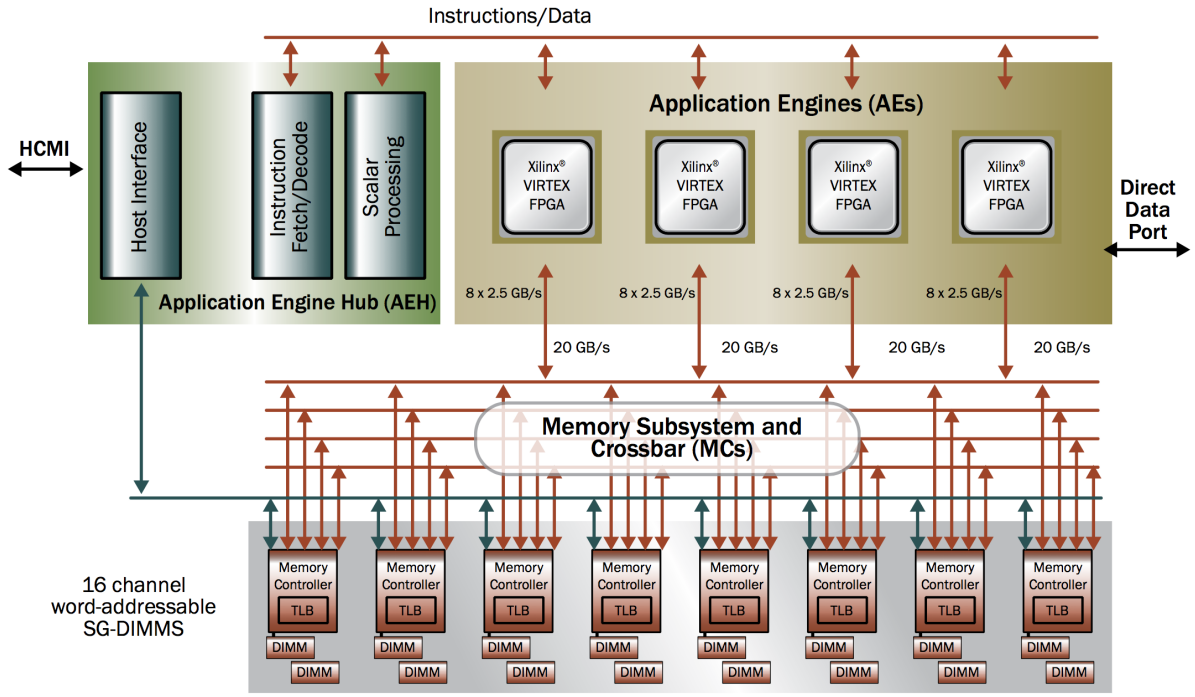

FIGURE 2.3: The Convey HC-2 memory Subsystem

2.1.3 Personalities

Personalities include precompiled FPGA bit files, a description of the machine state model sufficient for the compiler to generate and schedule instructions, and an ID used by the application to load the correct image at runtime. More specific a personality implements two types of instructions, the scalar instructions and the extended instructions. The scalar instructions are common among all personalities and ensure that all personalities share the same basic functionalities, while the extended instructions provide customization for different applications and workloads. All personalities have some elements in common. More specifically the defined Convey Instruction Set controls and initiates via instructions the co-processor execution. The co-processor instructions use virtual addresses and coherently share memory with the host processor. The host processor and I/O system can access co-processor memory and the co-processor can access host memory. Furthermore all the personalities use a common host interface to dispatch co-processor instructions and return status. On the other hand all the personalities support the canonical instruction set which can be generated and executed. In general a personality therefore implements a computer architecture customized for a particular type of algorithm or workload. It is widely known that many applications don't fit in classic architectural models. Convey provides a Personality Development Kit (PDK) that allows custom instructions to be implemented to support such applications.

2.2 Mutual Information

In this section we present a comprehensive background about Mutual Information (MI) as formulated in [12]. Also we will make some short references to other fundamental notions of Information theory closely related to MI in order acquire a better understanding of the notion of Information Theory. All the references that will presented in this section they are referred with great detail in [12] and [1].

It is a foregone conclusion that Information Theory is one of the few scientific fields fortunate enough to have an identifiable beginning, and that is the Claude Shannon's 1948 paper [23]. Also Information is a very broad and abstract notion. It is widely known that Information Theory answers two fundamental question in communication theory. What is the ultimate data compression and what is the ultimate transmission rate of communication. This approach led some scientists to consider that information theory to be a subset of communication theory and only targeted to the area of communication. In this section also we will describe why this belief is misunderstood, and we will present the fundamental contributions in the fields of prediction, filtering an learning.

2.2.1 Entropy

This section presents the notion of entropy, which is a measure of the uncertainty of a random variable. Before defining entropy in order to yield a better overall understanding of entropy it is worthwhile to present a quantified description of information. Let's suppose that *p(m)* is the probability that, message m is chosen from all the possible choices in the message space M. That means:

$$p(m) = Pr(M = m) \tag{2.1}$$

So the self-Information of a message based on Shannon's measure of Information is:

$$I(m) = log_2(\frac{1}{p(m)}) \tag{2.2}$$

So considering the above equations, we say that for the message m we received *I(m)* bits. We have to notice that the base of the logarithm only affects a scaling factor and consequently, the units in which the measured information content is expressed. If the logarithm in base 2 the measure of information is expressed in bits. If the logarithm is in base of e then the measure of information is expressed in nats. For example on tossing a coin the chance of 'tail' is 0.5. When it is proclaimed that indeed 'tail' occurred, this amounts to I('tail')= log2(1/0.5)=log2(2)=1 bit of information. Also when throwing a fair dice the probability of 'four' is 1/6. When it is proclaimed that 'four' has been thrown, the amount of information is I('four')= log2(1/(1/6)) = log2(6) = 2.585 bits. Thus, when independently, two dice are thrown, the amount of information associated with throw_1='two' and throw_2='four' equals I('throw'_1 is two and throw_2 is four') = (1/P(throw_1='two' and throw_2='four')) = log2(1/(1/36)) = log2(36) = 5.170 bits. This outcome equals the sum of the individual amounts of self-information associated with (throw 1 = 'two') and (throw 2 = 'four'), namely 2.585 + 2.585 = 5.170 bits. Considering the above, we are going to examine the definition of Entropy. Generally entropy is a measure of the uncertainty of a random variable. Let X be a discrete

random variable with alphabet X and probability mass function $p(x) = Pr(X = x)$, $x \in X$. We denote the probability mass function by $p(x)$ rather than $p_x(x)$, for convenience. Thus, $p(x)$ and $p(y)$ refer to two different probability mass functions,$p_x(x)$ and $p_y(y)$, respectively. So the entropy *H(X)* of a discrete random variable X is defined by

$$H(X) = -\sum_{x \in X} p(x) log_b(p(x)) \tag{2.3}$$

where b is the base of the logarithm used. Common values of b as referred above are 2, Euler's number e, and 10, and in the unit of entropy is Shannon for b=2, nat for b=e and hartley for b=10. When b=2, the units of entropy are also referred as bits. For example, the entropy of a fair coin toss is 1 bit. We will use the convention that 0 log0 = 0, which is easily justified by continuity since $xlogx \to 0$ as $x \to 0$. Adding terms of zero probability does not change the entropy. In order to understand better the notion of entropy we present some examples with the notion of entropy.

Let's assume that:

$$X = \begin{cases} 1, \text{with propability p} \\ 0, \text{with propability 1-p} \end{cases}$$

Then

$$H(X) = -plog_2(p) - (1-p)log_2(1-p) = H(p) \tag{2.4}$$

In particular H(X)=1 bit when p = 1/2.One of the basic properties of entropy is that entropy is concave function of distribution and equals 0 when p=0 or 1.It is a foregone conclusion that when p=0 or 1 the variable is not random and there is no uncertainty. Similarly, the uncertainty is maximum when p=1/2, which also corresponds to the maximum value of entropy.

Let's assume that:

$$X = \begin{cases} a, \text{with propability } 1/2 \\ b, \text{with propability } 1/4 \\ c, \text{with propability } 1/8 \\ d, \text{with propability } 1/8 \end{cases}$$

The entropy of X in bits is :

$$H(X) = -\frac{1}{2}log_2(\frac{1}{2}) - \frac{1}{4}log_2(\frac{1}{4}) - \frac{1}{8}log_2(\frac{1}{8}) - \frac{1}{8}log_2(\frac{1}{8}) = \frac{7}{4}$$

2.2.2 Joint Entropy and Conditional Entropy

In the previous section we defined entropy and we gave some examples in order to get an overall understanding of this notion. In this section we present the notion of joint entropy and the notion of conditional entropy. The same formula 2.3 can be extended to pairs of random variable vectors. Based on the above assumption, the joint entropy

H(X,Y) of a pair of discrete random variables *(X,Y)* with joint distribution $p(x, y)$ is defined as:

$$H(X, Y) = -\sum_{x \in X} \sum_{y \in Y} p(x, y) log_2(\frac{1}{p(x,y)}) \tag{2.5}$$

We also define the conditional entropy of a random variable X given another random variable Y as the expected value of the entropies of the conditional distributions, averaged over the conditioning random variable. More specifically the conditional entropy *H(Y | X)* is defined as:

$$H(Y \mid X) = \sum_{x \in X} p(x) H(Y \mid X = x) = \sum_{x \in X, y \in Y} p(x, y) log_2(\frac{1}{p(y \mid x)}) \tag{2.6}$$

The joint entropy *H(X,Y)* and conditional entropy *H(Y | X)* are correlated by the chain rule which says that the entropy of a pair of random variables is equal to the entropy of one variable plus the conditional entropy of the other. More specifically:

$$H(X, Y) = H(X) + H(Y \mid X) \tag{2.7}$$

Of course we must note that

$$H(Y|X) \neq H(X|Y)$$

but

$$H(X) - H(X|Y) = H(Y) - H(Y|X).$$

So now let's consider the follow example: *"If I know the student's major could I predict if he likes computer games?"*. Assuming that we have the following information as input about the college majors and as output we are trying to predict if the specific student likes Computer games or no. All the information described as presented on the the table 2.1:

TABLE 2.1: The input data for the example of the calculation of Conditional Entropy.

X	**Y**	**X**	**Y**
Math	Yes	Math	No
History	No	CS	Yes
CS	Yes	History	No
Math	No	Math	Yes

Considering the table 2.1, now we can create a new table and we can calculate the marginal distribution for X and Y respectively in the table 2.2.

TABLE 2.2: The calculation of Conditional Entropy.

$X \backslash Y$	YES	NO	**Marginal Distribution X**
Math (M)	0.25	0.25	**P(X=Math)=0.5**
CS (C)	0.25	0	**P(X=CS)=0.25**
History (H)	0	0.25	**P(X=History)=0.25**
Marginal Distribution Y	**P(Y=Yes)=0.5**	**P(Y=No)=0.5**	

Now let's calculate the specific Conditional Entropy $H(Y|X = \nu)$.This entropy of Y calculated only on those records in which X has value v. For example:

$$\begin{aligned} H(Y|X = math) = &-p(Yes|X = math)log(p(Yes|X = math)) \\ &- p(No|X = math)log(p(No|X = math)) = 1 \end{aligned} \tag{2.8}$$

Respectively the conditional entropies for *CS* and *History* are: H(Y | X=H) = 0 and H(Y | X=C) = 0.

Considering the formula 2.6, the conditional entropy *H(Y | X)* which is the average conditional entropy of *Y*, we can easily now calculate the *H(Y | X)*.

$$H(Y|X) = 0.5 \times 1 + 0.25 \times 0 + 0.25 \times 0 = 0.5.$$

So now let's calculate the joint entropy *H(X,Y)*. As we prove in formula 2.7 the joint entropy *H(X,Y)* and conditional entropy *H(Y | X)* are correlated by the chain rule which says that the entropy of a pair of random variables is equal to the entropy of one variable plus the conditional entropy of the other. So in our example the joint entropy is:

$$H(X, Y) = H(X) + H(Y|X) = 1.5 + 0.5 = 2.$$

2.2.3 Relative Entropy and Mutual Information

In this section we introduce two related notions, relative entropy and mutual information. The relative entropy is a measure of the distance between two distributions. In statistics, it arises as an expected logarithm of the likelihood ratio. The relative entropy $D(p \parallel q)$ is a measure of inefficiency of assuming that the distribution is q when the true distribution is p. For example, if we knew the true distribution p of the random variable, we could construct a corresponding encoding with average descriptive information *H(p)*.If,instead, we associate distribution q with the random variable, we would need $H(p) + D(p \parallel q)$ bits on the average to describe the random variable. The relative entropy between two probability mass functions $p(x)$ and $q(x)$ is defined as:

$$D(p \parallel q) = \sum_{x \in X} p(x) log_2(\frac{p(x)}{q(x)}) \tag{2.9}$$

where for the extreme cases $0log(\frac{0}{q}) = 0$ and $plog(\frac{p}{0}) = inf$ based on the arguments for the continuity principle. Similarly to entropy, relative entropy is a non-negative

measure which is equal to 0 if and only if, the assumed distribution is exactly the same as the true distribution.

Considering all the above it is obvious that we cannot regard relative entropy as a true distance between distributions for two main reasons. The first is the triangle inequality and the second is that it is not symmetric. Now we introduce the notion of Mutual Information that is the most important notion for this thesis. In the previous sections we presented that if entropy is regarded as a measure of *"uncertainty about a random variable"*, then *H(Y | X)* is a measure of *"what X does not say about Y."* To sum up briefly Mutual Information *I(X;Y)* computes the amount of information a random variable includes about another random variable. For example, suppose two discrete random variables *X,Y* which *X* represents the roll of a fair six-sided dice, whereas *Y* shows whether the roll is odd or even. Then, it is clear that the two random variables share information, as by observing one, we receive knowledge about the other. For a pair of discrete random variables *X*, *Y* with joint probability function $p(x, y)$ and marginal probability functions $p(x)$ and $p(y)$ respectively, the mutual information *I(X;Y)* is the relative entropy between the joint distribution and the product distribution:

$$I(X;Y) = \sum_{x \in X} \sum_{y \in Y} p(x,y) log_2(\frac{p(x,y)}{p(x)p(y)}) = D(p(x,y) \parallel p(x)p(y)) \tag{2.10}$$

Considering the above we must note that MI is nonnegative $I(X;Y) \geq 0$ and symmetric $I(X;Y) = I(Y;X)$. Moreover $I(X;Y) = 0$ if and only if, *X* and *Y* are independent random variables. This is easy to see in one direction: if *X* and *Y* are independent, then $p(x, y) = p(x)p(y)$ and therefore $log(p(x, y)/p(x)p(y)) = log1 = 0$.

2.2.4 Relation between Entropy and Mutual Information

In this section we can re-write the definition of Mutual Information as follows :

$$I(X;Y) = \sum_{x \in X} \sum_{y \in y} p(x,y) log_2(\frac{p(x,y)}{p(x)p(y)}) \tag{2.11}$$

$$= \sum_{x \in X} \sum_{y \in y} p(x,y) log_2(\frac{p(x|y)}{p(x)}) \tag{2.12}$$

$$= - \sum_{x \in X, y \in Y} p(x,y) log_2(p(x)) + \sum_{x \in X, y \in Y} p(x,y) log_2(p(x \mid y)) \tag{2.13}$$

$$= - \sum_{x \in X} p(x) log_2(p(x)) - (- \sum_{x \in X, y \in y} p(x,y) log_2(p(x \mid y))) \tag{2.14}$$

$$= H(X) - H(X \mid Y) \tag{2.15}$$

The above formula shows that assuming the entropy *H(X)* as *"the uncertainty about the random variable X"* and the *H(Y | X)* is the *"the amount of uncertainty remaining about Y after X is known"* the right side of these equalities can be read as *"the amount of uncertainty in X, minus the amount of uncertainty in X which remains after Y is known"*, which is equivalent to *"the amount of uncertainty in Y which is removed by knowing X"*. This corroborates the intuitive meaning of mutual information as *the amount of information (that*

is, reduction in uncertainty) that knowing either variable provides about the other. The chain rule does not apply to entropy, but also relative entropy and mutual information. More specifically, in the case of entropy the general chain rule says that assuming $X_1, ..., X_n$ are drawn according to $p(x_1, ..., x_n)$ then:

$$H(X_1, .., X_n) = \sum_{i=1}^{n} H(X_i \mid X_{i-1}, .., X_1) \tag{2.16}$$

Similarly, the chain rule for relative entropy between two joint distributions on a pair of random variables can be expanded as the sum of a relative entropy and a conditional relative entropy:

$$D(p(x, y) \parallel q(x, y)) = D(p(x) \parallel q(x)) + D(p(y \mid x) \parallel q(y \mid x)) \tag{2.17}$$

Finally, Mutual information also satisfies the chain rule which yields:

$$I(X_1, .., X_n; Y) = \sum_{i=1}^{n} I(X_i; Y \mid X_{i-1}, .., X_1) \tag{2.18}$$

2.3 Transfer Entropy

2.3.1 Definition of Transfer Entropy

In this section we present the notion of Transfer Entropy. Transfer entropy was designed to determine the direction of information transfer between two, possibly coupled, processes, by detecting asymmetry in their interactions. More specifically, it is a Shannon information-theoretic quantity which measures directed information between two time series. Formally, transfer entropy shares some of the desired properties of mutual information but takes the dynamics of information transport into account. As an example, if there are two stocks, namely *A* and *B*, it is important to know whether *A* is correlated to *B*, but it is also important to know that, for instance, *A* affects *B*, which may mean that if *A's* price rises, then *B's* price will also rise, but the opposite might not hold (i.e., "when *B's* price rises, *A's* price will rise too").

Assuming two time series *X*,*Y* the Transfer Entropy from *Y* to *X* can be given by:

$$T_{Y \to X} = \sum_{x_{n+1}, x_n, y_n} p(x_{n+1}, x_n, y_n) log_2(\frac{p(x_{n+1} | x_n, y_n)}{p(x_{n+1} | x_n)}) \tag{2.19}$$

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

$$T_{X \to Y} = \sum_{y_{n+1}, x_n, y_n} p(y_{n+1}, x_n, y_n) log_2(\frac{p(y_{n+1} | x_n, y_n)}{p(y_{n+1} | y_n)}) \tag{2.20}$$

Considering the definition of conditional probabilities can be rewritten as:

$$T_{Y \to X} = \sum_{x_{n+1}, x_n, y_n} p(x_{n+1}, x_n, y_n) log_2(\frac{p(x_{n+1}, x_n, y_n) p(x_n)}{p(x_{n+1}, x_n) p(x_n, y_n)}) \tag{2.21}$$

$$T_{X\to Y} = \sum_{y_{n+1},x_n,y_n} p(y_{n+1},x_n,y_n) log_2(\frac{p(y_{n+1},x_n,y_n)p(y_n)}{p(y_{n+1},y_n)p(x_n,y_n)}) \tag{2.22}$$

2.3.2 Relation between Transfer Entropy and Mutual Information

In the previous section we presented the notion of Transfer Entropy and we described the basic equations. Considering the above we conclude that Transfer Entropy can detect the directed exchange of information between two time series, unlike the Mutual Information which is designed to ignore static correlations due to the common history. Transfer entropy is introduced by Thomas Schreiber in his paper *"Measuring Information Transfer"* [22]. In this section we describe the relation between Transfer Entropy and Mutual Information through one example which described in [22]. In this paper there is an example in order to detect whether heart beat results in breath or vice versa.

Let's assume that the figure 2.4 describes a bivariate time series of the breath rate and instantaneous heart rate of a sleeping human suffering from sleep apnea. The data is sampled at 2 Hz. Both traces have been normalized to zero mean and unit variance. We can calculate the Transfer Entropy from heart to breath $T_{heart\to breath}$ and from breath to heart $T_{breath\to heart}$ and Mutual Information from breath to heart $M_{breath\to heart}$ and from heart to breath $M_{heart\to breath}$.

Next figure 2.5 shows transfer entropy *(solid line and dotted line)* compared with time delayed mutual information *(dashed line)*.More specifically small r means high resolution so r represents the granularity. As we can observe $T_{heart\to breath}$ *(solid line)* is always larger or equal to $T_{breath\to heart}$ *(dotted line)*.

As a conclusion we could say that heart beat results in breath. In contrast, time delayed mutual information failed to distinguish causality because the two dashed lines corresponding to $M_{heart\to breath}$ and $M_{breath\to heart}$ *(dashed line)* overlap in the figure. So we could yield as a result that transfer entropy overpowers against the information-based measures.

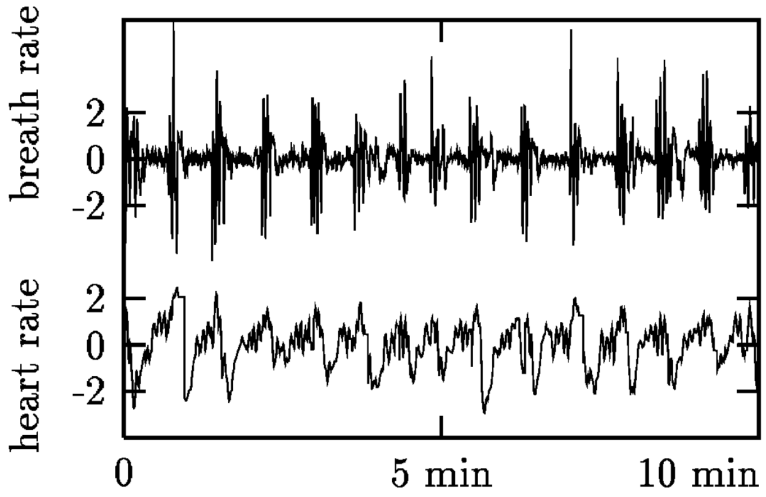

FIGURE 2.4: Bivariate time series of the breath rate (upper) and instantaneous heart rate (lower) of a sleeping human. The data is sampled at 2 Hz. Both traces have been normalized to zero mean and unit variance.The figure borrowed from [22]

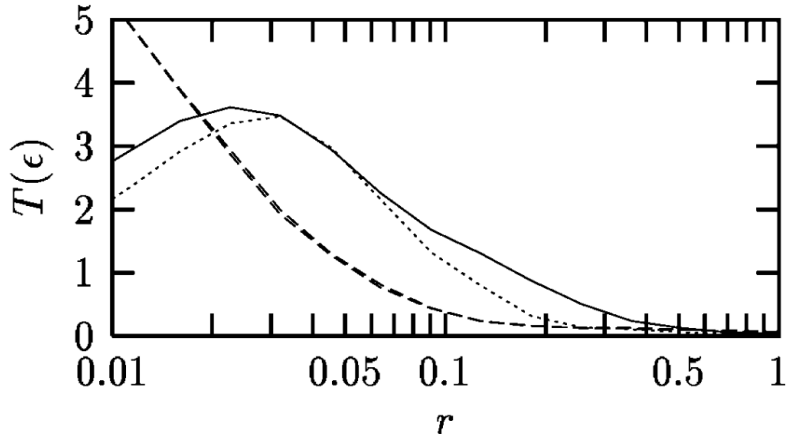

FIGURE 2.5: Transfer entropies $T_{heart \to breath}$ *(solid line)* $T_{breath \to heart}$*(dotted line),*and time delayed mutual information M(t=0.5 s) (directions indistinguishable, dashed line).The figure borrowed from [22]

Chapter 3

Related Work

In this section we present a few hardware-based approaches which have been proposed for the acceleration of Mutual Information and Transfer Entropy respectively. Because of Transfer Entropy is quite similar measure with Mutual Information we present the proposed architecture for both algorithms in one section. So in this section we present a few FPGA-based and GPU-based hardware approaches for both algorithms.

3.1 FPGA

One of the first FPGA-based approach for the acceleration of Mutual Information was presented by Castro-Pareja and Shekhar in [2]. In this paper the proposed architecture can be used to accelerate linear and image registration algorithms which use Mutual Information as a similarity measure. This architecture called FAIR-II and is targeting to perform image registration in real time but without the use of a supercomputer. To the best of our knowledge this architecture has two basic steps. In the first step the mutual and individual histograms are created. More specifically this step creates these histograms with an algorithm which convert the floating image's coordinates to the respective reference ones with the use of Partial Volume interpolation. In the second step the accumulator modules send the partial joint histogram values to the entropy accumulator and in the entropy accumulator the individual and point probabilities are calculated from the joint histogram. The results are then accumulated, thus obtaining the mutual information value. Their architecture mapped on an Altera Stratix EP1S40 FPGA and it could process 50 million reference image voxels per second. This architecture achieved x30 speedup for linear registration and x100 speedup for elastic registration compared to a software implementation on a 3.2-GHz Xeon with 1GB of 266 MHz DDRAM.

The first reconfigurable computing solution to accelerate Transfer Entropy presented in the papers of Shao in [9] and from the same author in [24]. The novel aspects of this approach include a new techinque for the probability estimation based on Laplace Rule of Succesion. At the other end of the spectrum on hardware they managed to map some small and medium-sized tables on BRAMs during initialization while sending large ones at run-time. In this way, they pipeline data transfer and computing. Moreover in their design in order to optimise memory allocation and to reduce I/O overheaded, they adopted the technique of bit-width narrowing. Their implementation

targeted a Xilinx Virtex-6 SX475T FPGA. In their experiments, the proposed FPGA-based solution is up to 111.47 times faster than one Xeon CPU core, and 18.69 times faster than a 6-core Xeon CPU.

3.2 GPU

In this section we present a few GPU-based approaches for the Mutual Information and Transfer Entropy. In the recent decade, the computing capacities of the graphics processor units (GPUs) have improved exponentially. This has made GPU-accelerated computation a viable option for many applications.

Shams and Barnes in their paper [20] presented an efficient method for mutual information computation between images (2D or 3D) for NVIDIA's compute unified device architectures (CUDA) compatible devices. In order to use the 1D joint histogram code for joint histogram calculation in the preprocessing stage they convert the 2D joint histogram calculation to 1D joint histogram calculation. After the preprocessing stage the pmf calculation distributed to L thread blocks each with N threads. Each block will maintain a partial histogram of its own in the global memory for the portion of the input data assigned to the block. Partial histograms are finally summed up using a very efficient multi-threaded reduction function. They implement their architecture on an NVIDIA 8800 GTX platform. Moreover, results indicated that in the case of a 3D image with approximately 7x10 million voxels and 256 threads the GPU-based registration was around 25 times more efficient.

In addition another paper from Lin and Medioni [32] presented a GPU implementation to compute both mutual information and its derivatives. They show that computations of Mutual Information and its derivatives are fully parallelizable and can be efficiently ported onto the GPU architecture. They focus on the speed up of Viola's approach to computing mutual information, since the approach follows a close form solution of the mutual information derivatives. More specifically in order to reduce the number of exponential computations they pre-computing the Gaussian densities of all possible density values into a lookup table. Also they find a way to parallelize the computation of Viola's algorithm in order to achieve a significant speed-up by using the GPU. The probability density is estimated using Parzen Window method which directly uses the samples drawn from an unknown distribution, and uses Gaussian Mixture model to estimate its density, which is robust to noise. After that they tackle the image registration problem. They solve this problem by estimating the transformation T that best aligns two images. In order to do so they maximize mutual information by approximating its derivative. For the proposes of parallelism the statistics computed for each element, are independent from the others, which can be computed efficiently in GPU. Their design mapped on an Nvidia GeForce 8800 GTX platform. For 1000 samples the computation time for both mutual information and its derivatives is reduced up to a factor of 170 and 400 respectively compared with a work station level CPU.

Another paper by Guo and Luk [7] presents an FPGA accelerator for ordinal pattern encoding, a statistical method for analyzing the complexity of time series. Even though we are not studying analyzing the complexity of time series in this thesis, it is worth to make a reference on this paper because the complexity of time series covers our implementation of Mutual Information and Transfer Entropy. They apply it to the

computation of permutation entropy. They propose a two-level hardware-oriented ordinal pattern encoding scheme to avoid sequence sorting operations in the accelerator, enabling theoretically best code compactness. Second, they develop a hardware mapping method by promoting data reuse, by parallelizing arithmetic operations, and by pipelining the data path. The FPGA system is implemented on a Xilinx Virtex-6 FPGA, and is integrated with a quad-core Intel i7-870 CPU running at 2.93GHz. Experimental results show that the hybrid system is up to 11 times faster than the CPU-only solution.

In addition to the aforementioned implementations, other works have also been proposed for hardware-based Mutual Information computation [21], [10]. However, apart from [9], all these approaches focus on accelerating specifically the image registration problem.

To conclude for Transfer Entropy, as it is a new statistical metric, we are not aware of any published work on its hardware acceleration on FPGA.

Chapter 4

Implementation

This Chapter focuses on the profiling of the Mutual Information and Transfer Entropy algorithms as well as on the implementation of them based on the specific features of Convey HC-2ex. We describe the optimization of the computationally intensive part of these two algorithms. We propose hardware architectures which accelerate the algorithms of Mutual Information and Transfer Entropy with respect to the computational features of the Convey HC-2ex in order to achieve the best possible performance.

4.1 Mutual Information

4.1.1 Software Profiling of MI Algorithm

In order to optimize the calculation of Mutual Information, it is necessary to profiling the software algorithm, in order to find the most-computationally intensive part of the algorithm and accelerate it on hardware.

Before to proceed to the calculation of the Mutual Information algorithm, the estimation of Mutual and joint Probability Density Functions (PDF) are necessary. Histograms used for the estimation of Mutual and joint Probability Density Functions (PDF). A histogram is a graphical representation of the distribution of numerical data. It is an estimation of the probability distribution of a continuous variable and was first introduced by Karl Pearson [8]. The first step to construct a histogram is, to divide the entire range of values into a series of small intervals. The second step is to count the number of values which fall into each interval.

More specifically the value range of the X and Y random variables is divided into R parts, with R being the number of Bins of the histogram. That means that each value of the time series is classified into one of the R Bins and the value of that Bin is increased by 1. In our approach, the outputs produced from the histogram construction are probabilities $p(x)$, $p(y)$ and $p(x,y)$, which correspond to the PDF functions of random variables X,Y and joint variable of X and Y respectively. The first two functions are one-dimensional vectors, whereas the third is depicted by a two-dimensional matrix. The result of the above is the estimation of the PDF's functions given two random variables. Many methods for PDF estimation have been proposed like Kernel Density Estimation(KDE) or K-Nearest Neighbor(KNN). We used the equi-distant histogram estimator which is the simplest non-parametric density estimator and it is easy to produced and understand.

Considering all the above the time required to produce the PDF estimations directly depends on the length of the time-series. The table 4.1 presents the dependency

between the execution time and the size of the inputs for the PDF estimation keeping the Resolution equal to 10000 because this value is the maximum value for the resolution we can reach. From the table 4.1 we conclude that the execution time for PDF estimation increases linearly with the input size.

TABLE 4.1: Dependency between the execution time and the size of the inputs for the PDF estimation of the Mutual Information Algorithm.

Time series(Elements)	Execution Time(sec)
10000	0.00016
100000	0.0043
1000000	0.043
10000000	0.42
100000000	4.8
1000000000	51.2

At the other end of the spectrum, on the second table 4.2 we can conclude that the increase of the number of bins can cause the squared increase of the execution time, keeping the size of time series to a fixed value of 100000 elements. Furthermore from the equation 2.11, goes without saying, that the time complexity for the MI computation is $O(R^2)$. To provide a sufficient understanding, for a very small number of Bins and a very large time series lengths the pdf estimation can be more time consuming, but for example for 1000 bins and for size of time series equal to 100000 elements, based on the MI computation requires 2.58 sec, whereas the creation of the histogram only needs 0.0043 sec which means that the most of the time consumed to the calculation of Mutual Information.

TABLE 4.2: Dependency between the execution time and the number of bins for the Mutual Information Algorithm.

Number of Bins(R)	Execution Time(sec)
100	0.002
500	0.012
1000	0.032
2000	0.11
5000	0.65
10000	2.58

To conclude, based on the tables 4.1 and 4.2 we can assume that the most computationally intensive part of the MI algorithm is the final computation of the statistic value. So after the software profiling of the MI algorithm we have been summoned to accelerate the MI calculation between two random variables, so the formula 2.11 needs to be parallelized and mapped to hardware.

4.1.2 CPU and FPGA Integrated System of MI Architecture

As we described in the previous section we can split the algorithm in two parts. The first part is the PDF estimation and the second part is the computationally intensive

part of the MI algorithm which is the final computation of this statistic value. The first part of the algorithm was implemented on the host side of the system and the second part was mapped on the hardware side. Also we must note that the PDF estimation, based on histograms, requires random accesses, which would be very slow if this part of the algorithm would have been implemented on the shared memory of Convey HC-2ex. We conclude that the most efficient approach for the PDF estimation is to be implemented in software side. The basic Mutual Information System illustrated in the figure 4.1 and the basic Mutual Information flowchart illustrated in the figure 4.2 respectively. As we can observe from the figure 4.1, the outputs of the PDF estimation stage are probability vectors $p(x)$, $p(y)$ and $p(x, y)$ which are the PDF function of the random variables X,Y and joint variable of X,Y respectively.

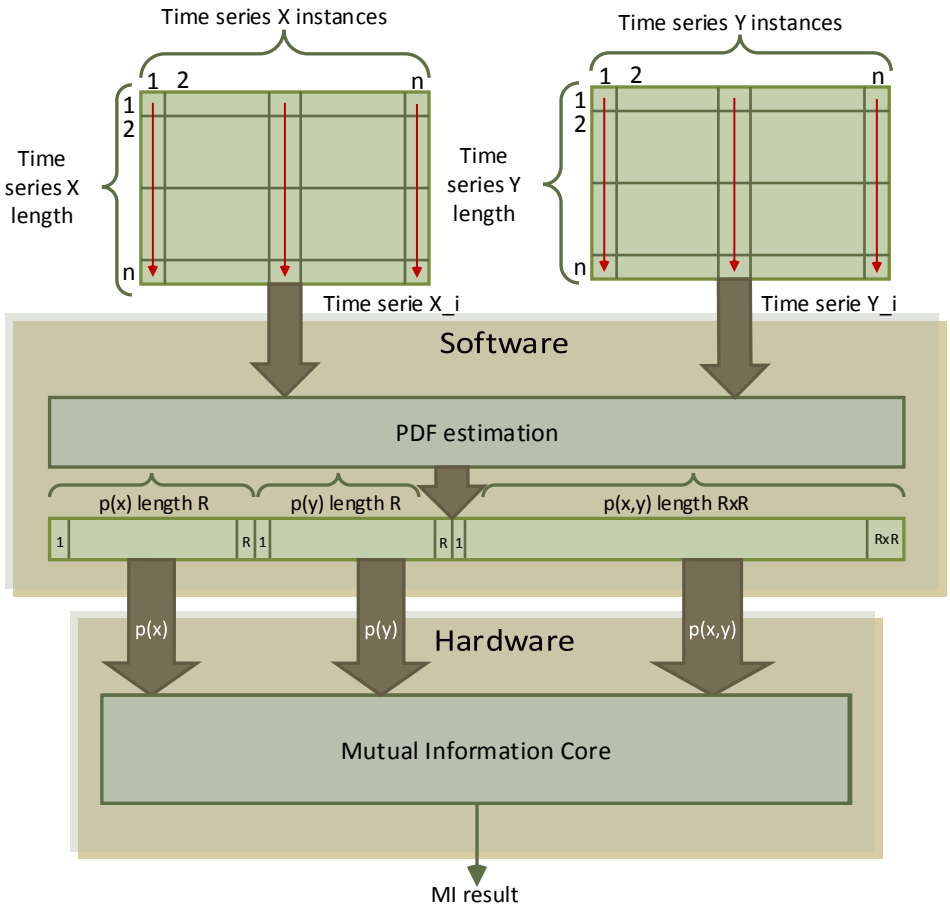

FIGURE 4.1: Basic System Architecture of Mutual Information

Also from the figure we can note that the first two function $p(x)$ and $p(y)$ are 1-dimensional vectors and $p(x, y)$ is a 2-dimensional vector. Furthermore the input time series are vectors which stored in one-dimensional structures of floats.

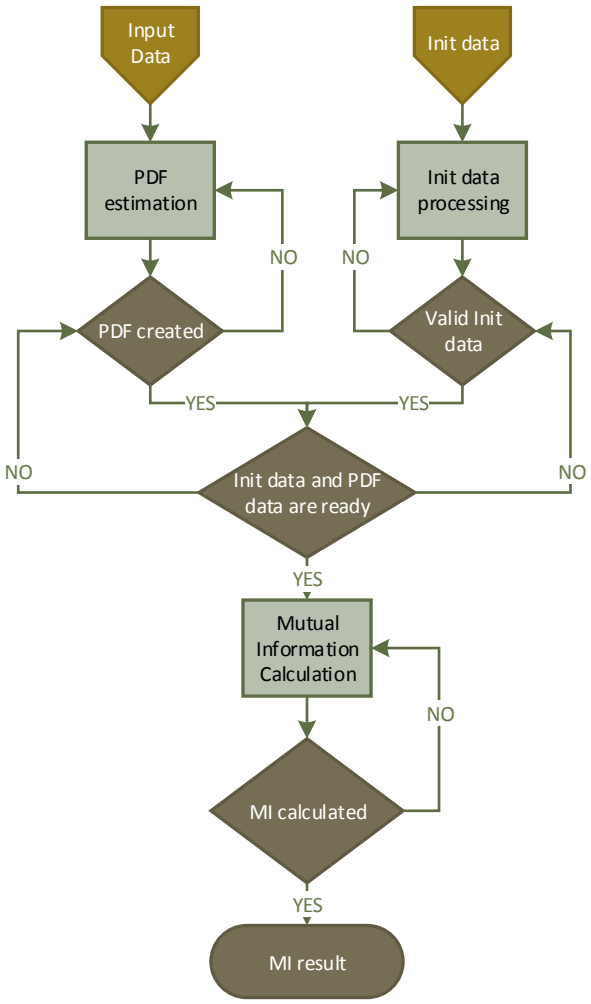

FIGURE 4.2: Basic flowchart of Mutual Information

4.1.3 Pre-processing on Host Processor

In this section we present the pre-processing of the dataset for the deployment of a single AE. We explore the way in which, Convey acts in order to process the data in the FPGA's. We note that when multiple AEs are deployed the same process is followed but with respect in some major differences.

As we described in the previous section before we proceed to the calculation of Mutual Information in the hardware side, we have to estimate the PDF functions $p(x)$, $p(y)$, $p(x,y)$ from the random variables X,Y. We presented why we choose to implement this estimation in the Host side. After the creation of the PDF function we have to hold the PDF functions to the shared memory of the system in order to get retrieved from the FPGA's via memory controllers. More specifically a C-based application, which is running on the host processor, is responsible for the creation of the PDF functions and for the allocation of the necessary data structures for the hardware execution.

To make the function call to the co-processor, some piece of information must be

sent to the AEs. In particular a pointer to the first address of the contiguous memory space in which the input data and the initialization constants are located. In our implementation, that means, a pointer to the first element of the PDF vector and a pointer to the first element of the initialization data. Secondly a pointer to the first address of the contiguous memory space that the output data will be stored. In order to, MI calculation takes place we need to have further information about the length of the time series and the length of the PDF functions. This information allocated also in the shared memory and get used by the FPGA's. In order to hardware execution take place a function call with the necessary addresses as arguments must be implemented.

As we described above the two first arguments are the pointers to the PDF vector and initialization data respectively. The rest of the values are constants for each architecture of MI core.

After the necessary allocations, the software is ready to call the co-processor to start processing the dataset, so at this point, a function call implemented through the convey-specific $copcall fmt()$ function. We note that the limited attributes of this function call is set to 15 64-bit values.The figure 4.3 illustrates the $copcall fmt()$ with the attributes for the MI calculation.

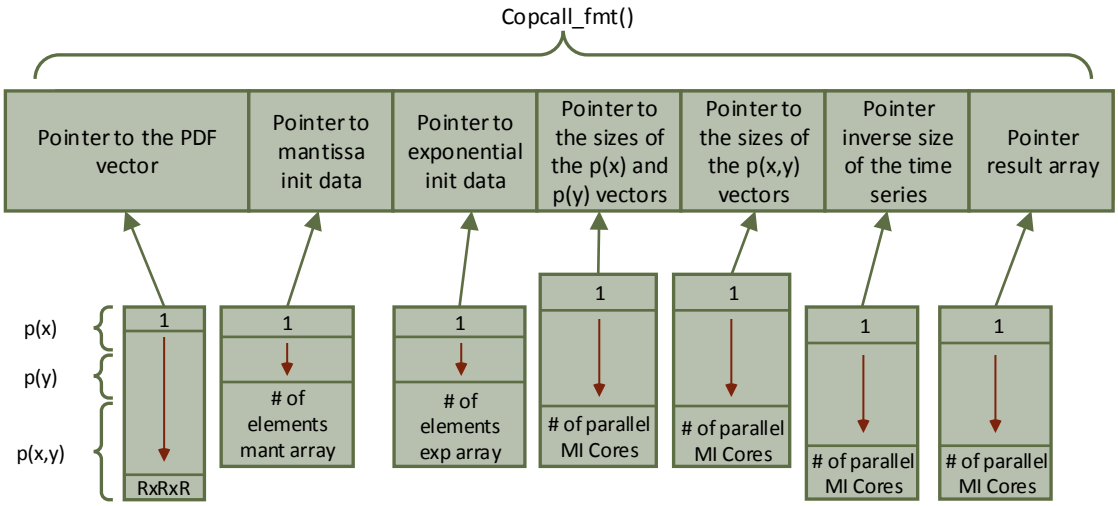

FIGURE 4.3: copcallfmt() function call with attributes of Mutual Information

Afterwards, all the initial addresses for each memory controller, are assigned via a short assembly script. This assembly script can be modified by the user. More specifically this script is responsible for the routing of all the necessary constants and addresses to the FPGA's, in order to start the processing. This script also is responsible for the sharing of the workload from one to four AEs, depending on the preferences of the programmer. In our implementation this script is designed for multi-FPGA instantiations as for single-FPGA ones, with the difference that each AE is assigned different addresses.

4.1.4 Hardware Architecture Instantiation

After the pre-processing all the necessary data are allocated on the shared memory of the Convey HC-2ex. Afterwards the co-processor is ready to process the data and calculate the Mutual Information. In this section we present the basic architecture which mapped onto the FPGA's. The proposed architecture is described using VHDL and developed with the Xilinx ISE 12.4 interface, while the floating-point cores and memory elements are generated by the Xilinx Core Generator.

4.1.4.1 Decomposition of the Problem

From the software profiling, we conclude that when $p(x)$, $p(y)$ and $p(x, y)$ vector created, the final result is ready after R^2 clock cycles. So taking into account the idea of parallel computing, we can split $p(y)$ and $p(x, y)$ into two streams.

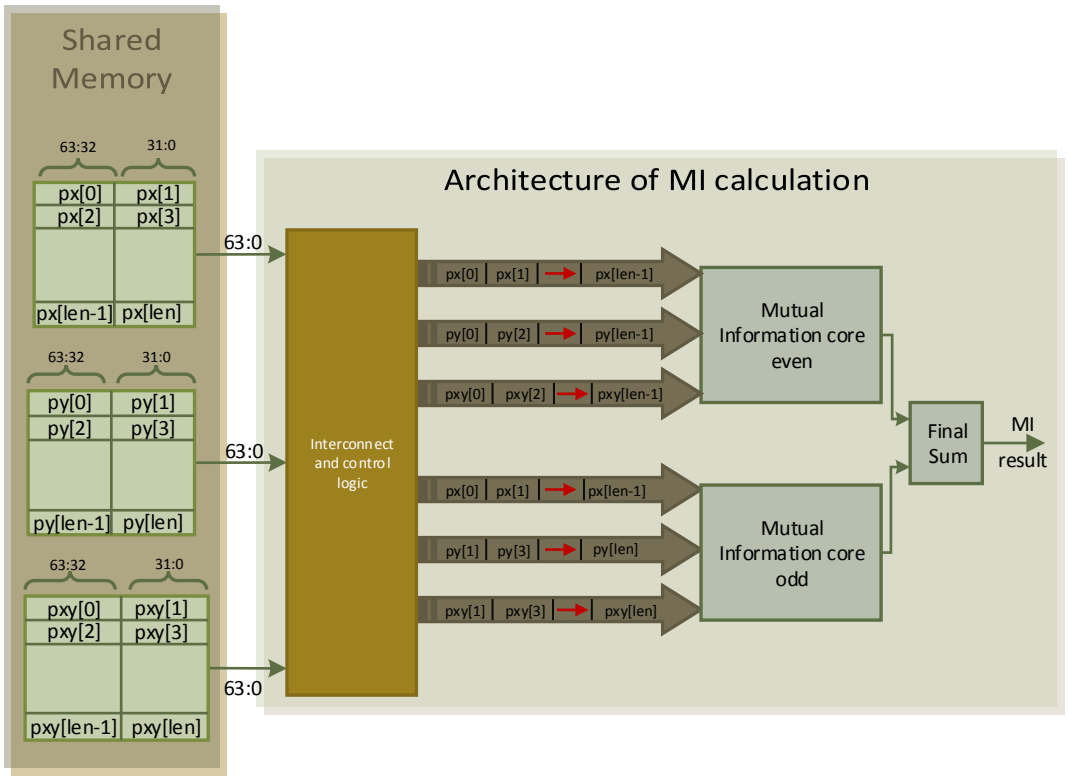

FIGURE 4.4: Partitioning of the Mutual Information Algorithm

Then, each half of $p(y)$ and $p(x, y)$ are processed simultaneously, as there is no dependence between the different levels of the distribution function. The system uses two parallel units to compute an MI value twice. Note that we do not have to split $p(x)$, we split only $p(y)$ and $p(x, y)$ vectors due to the fact that index y in the sum formula remains constant, thus allowing us to divide the vector, whereas index x receives all possible values in the range from 1 to R for a constant y. Therefore, the hardware produces two partial MI, the first produced from the elements in the even positions

of the vector and the second produced from the elements in the odd positions of the vector that are summed at the end of the partial calculation of the Mutual Information. With this simple consideration we divide the required time of R^2 by a factor of 2, which means finally $\frac{R^2}{2}$ cycles.

The figure 4.4 illustrates the idea of partitioning the MI algorithm in odd and even pipelined cores. We note that the hardware processing is fully pipelined, meaning that to produce correct partial MI values each cycle, the respective $p(x)$, $p(y)$ and $p(x, y)$ vectors should arrive simultaneously at each processing unit of the MI calculation.

The idea of splitting the $p(y)$ and $p(x, y)$ vectors into odd and even sub-vectors was invented by the way which the data allocated in the shared memory of the Convey HC-2ex. More specifically when an address requested through memory controller to the shared memory, the memory controller respond with an 64 bit value. The MI calculation uses single-precision floating-point arithmetic so in 64 bit we can store 2 single-precision floating-point. With this consideration we exploit the whole bandwidth of the system.

As a result with every request in the shared memory of the Convey HC-2ex through memory controllers, the memory controllers respond with 2 single-precision floating-point values each clock cycle.

4.1.4.2 Data Movement to Hardware Side

In this section we present the architecture between the computational modules of the architecture and the shared memory. More specifically we describe how to retrieve and manipulate the necessary data from the shared memory in order to calculate MI.

The main interface which connects the shared memory of the Hybrid-superComputer with the FPGA's is the interface of Memory Controllers. More specifically the Memory Controller interface provides the AEs with direct access to the co-processor memory. The 4 AEs are connected to 8 MCs, each via a 300 MHz DDR interface designed by Convey Computers, while each of the 8 interfaces is directly connected to a single MC. As a result the 300 MHz interface is converted into two 150MHz memory ports to and from the AE personality.

In our architecture every vector of $p(x)$, $p(y)$ and $p(x, y)$, binds one memory controller port each. Also when the core is ready for writing the result back to the shared memory one of these three accepts the responsibility to store the result. Furthermore that three memory controllers for each special-purpose core are responsible to retrieve the constants which are essential for the computation.

The retrieval of the data from the shared memory can split up in two discrete phases. To begin with:

- In the first phase the three memory controllers acquire the addresses of the allocated constants from the C-based application in order to get retrieved in our architecture. More specifically that means the length of the $p(x)$ vector and $p(y)$ consequently, the length of the $p(x, y)$ and the initialization data for the BRAM's for the logarithm unit.

- Afterwards in the second phase, we need to retrieve the PDF functions $p(x, y)$, $p(x)$ and $p(y)$ from the shared memory to the hardware side in order to MI calculation take place. In this phase as long as we retrieved the lengths of the vectors

and the BRAM's has already initialized, we calculating the requested address range for each vector. So as long as the length of each vector get retrieved as well as the address of the first element of the vector passed through as argument, the system starts to request the elements of the vectors in this address range through memory controllers. The figure 4.5 illustrates the connection between the shared memory of the system and our implementation. Note that all the FIFO's which are illustrated in the figure consist of two FIFO's instances which hold a floating point 32-bit value.

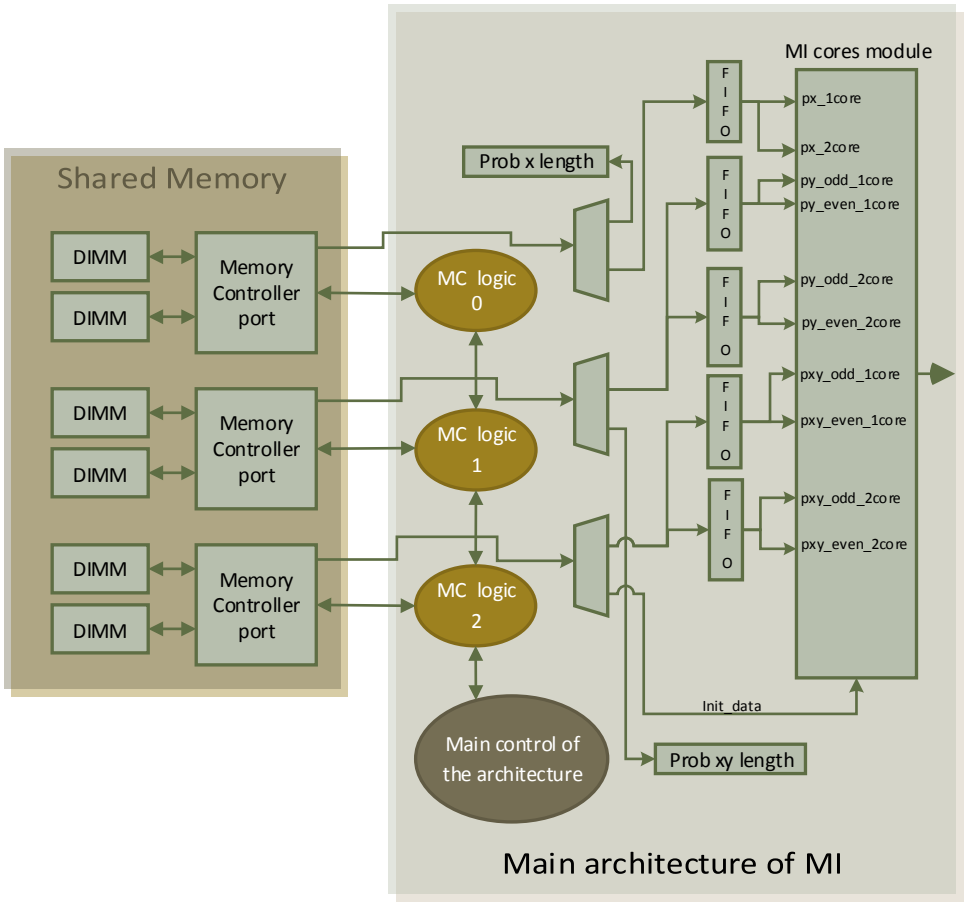

FIGURE 4.5: Connection between the architecture and the shared memory of the Convey HC-2ex

To begin with as we can observe from the figure 4.5 every port to the shared memory is connected with a logic module implemented in our architecture which acts in order to request data from the shared memory and manages all the responds and control signals from the memory. To be more specific we note that logic modules of the memory controllers are intimately related with the control signals of the memory controller. These three logic modules controlled by the main control unit as illustrated in the figure 4.5. More specifically we look into the finite state machines of the three logic modules as well as the finite state machine of the main control unit. In order to provide a sufficient understanding, we present the functionality of each logic module:

- **MC logic 1:** This logic module consists of two processes and retrieves the elements of the vector $p(x)$. The first process is responsible for the requests from our architecture to the shared memory and the second process is responsible for counting the elements which are retrieved from the memory. This logic module is connected with the interface of the first memory controller port of the system. The first process is a finite state machine which consists of three states. The first state is the state *idle*. In this state the logic module waits for a signal from the Main Control Unit in order to proceed to the next state. Note that when this logic module is in the *idle* state, the Main Control Unit is responsible for fetching the proper addresses to the logic module in order to request data. When the synchronization signal arises, the finite state machine proceed to the next state which is the state *load*. In this state the logic module has already fetched the length and the first address of the vector, so it can calculate the last address of it. Afterwards starts to request the addresses from the first address up to the last address of the vector. After that the logic module proceed to the *idle* state again and waits for the store signal which arises when the result is ready for store from the pipeline. This control signal synchronized from the Main Control Unit. When the store signal arises the logic module proceed to the *store* state in order to enable all the proper signals of the memory controller for a store. When the respond from the memory that the result stored arises, the logic module proceed to the *idle* state again and waits for the system to finish the calculation. As we mentioned above the logic module has two processes, in other words, two finite state machines in parallel. Up to this point we described the first FSM which is responsible for the requests of the data.

  The second FSM is responsible for counting the retrieved elements. It has two states the state *idle* and the state *count*. The function of *idle* state is exactly the same with the *idle* state in the first process. In the *count* the process waits for new elements and when a new requested element retrieved, this state keeps a counter updated.

  So with this logic module we retrieve a piece of elements based on their first address and their length and we can keep a counter updated about the number of elements which retrieved from the shared memory as well as to store the MI result.

- **MC logic 2:** This logic module fetches the elements of the vector $p(y)$ and consists of two processes. The first is responsible for the requests in the shared memory and the second is responsible for counting the retrieved elements, similarly with the *MC logic 1*. To begin with the first process is the same with the process of *MC logic 1* with the only difference the number of states. Because the store request to the shared memory implemented by the memory controller which fetches the vector $p(x)$, in this process we meet only two states, the *idle* state as well as the *load* state. The *idle* state is exactly the same with the *idle* state of the *MC logic 1*. As we described above the *load* state is the state which starts to request the addresses from the first address up to the last address of the vector. The only difference with *MC logic 1* is that when the vector retrieved, then the *load* state starts from scratch in order to retrieve the vector $p(y)$ *R* times. The fact that distribution estimations $p(x)$ and $p(y)$ are one-dimensional vectors and the calculation of MI assumes that

for one retrieved element of $p(x)$ vector, needs to retrieve all the elements of $p(y)$ as well as R elements from the vector $p(x, y)$ for one iteration of element $p(x)$, lead us to the decision to stream $p(y)$ vector R times into the architecture. When the retrieval of the vector *R* times finished, the logic module proceed to the *idle* state again and waits for the system to finish the calculation.

The second in parallel FSM is responsible for the counting of the retrieved elements as described above in the *MC logic 1*. The only difference is that in the *count* state the counter counts how many times the vector retrieved, while in the *MC logic 1* the counter counts the number of elements which retrieved from the memory. When the vector retrieved R times then the function proceed to *idle* state.

- **MC logic 3:** This logic module responds with the elements of the $p(x, y)$ vector. The processes of this logic module are two and the first process is exactly the same with the first process of the *MC logic 1* but without the functionality of the storing state as well as the second process described in the *MC logic 1* with the same number of states.

The figure 4.6 illustrates the functionality which described up to this point for the *MC logic 1*, the figure 4.7 illustrates the *MC logic 2*. The *MC logic 3* is similarly with *MC logic 1* and *MC logic 2* as described above.

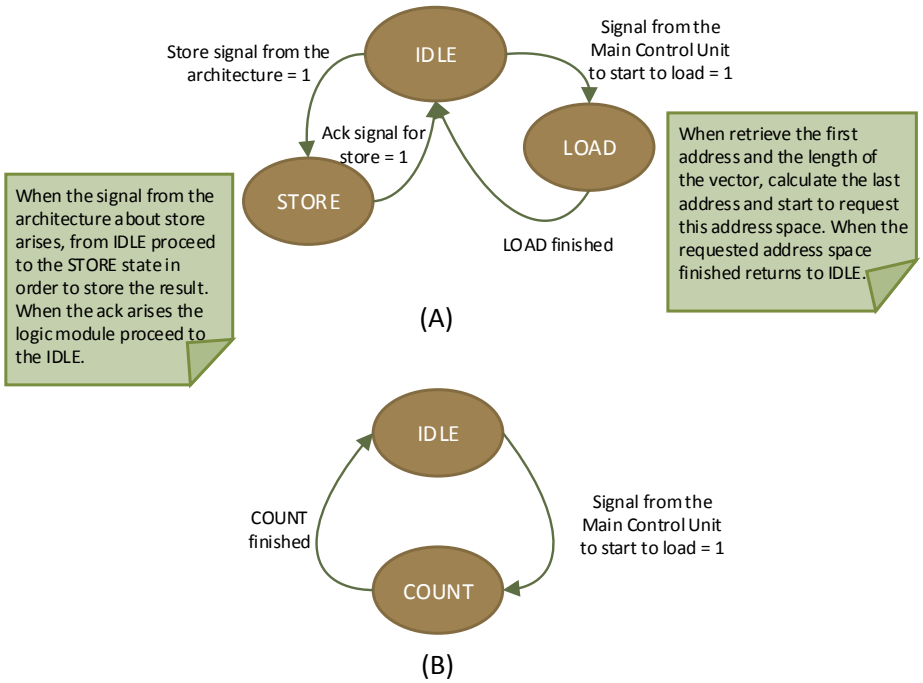

FIGURE 4.6: Finite State Machines for the Memory Logic 1. In (A) illustrated the FSM for the *load* and in (B) illustrated the FSM for the *count*.

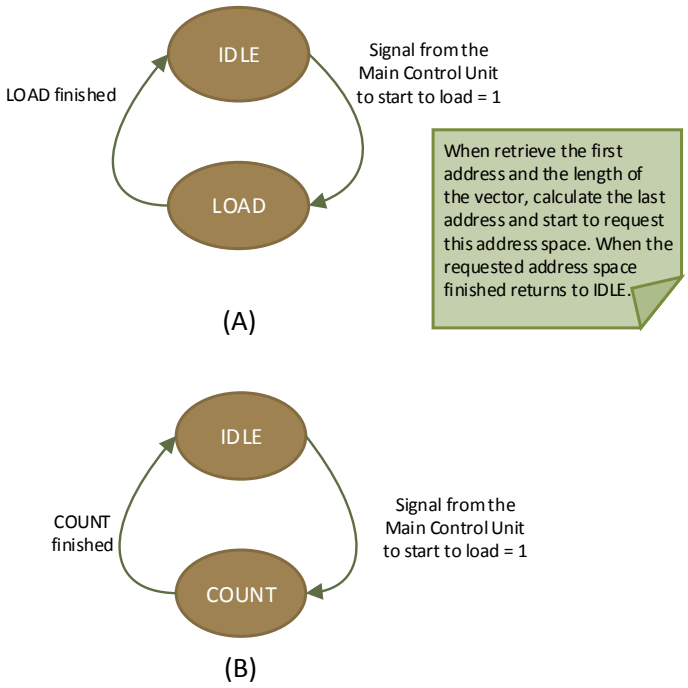

FIGURE 4.7: Finite State Machines for the Memory Logic 2. In (A) illustrated the FSM for the *load* and in (B) illustrated the FSM for the *count*.

4.1.4.3 The Main Control Unit

The Main Control Unit as shown in the figure 4.5 is the main unit which controls all the units in our architecture. This finite state machine has 7 states, all of them control a different part of the system. To begin with we present the functionality of each state in order to provide a sufficient understanding. So the functionality of each state implemented as follows:

- ***idle* state:** This is the first state of the Main Control Unit and it is the state which waits for the *start* signal in order to proceed to the next state and control the reset of the system in case of reset mode. In this state when the *start* signal arises, the *MC logic 1* updated with the address of the length of the $p(x)$ and $p(y)$ consequently, the *MC logic 2* updated with the address of the $p(x, y)$ vector length and the *MC logic 3* updated with the first address of the initialization data block for the mantissa vector and its size in the shared memory. So after that, the Main Control Unit proceed to the *init mantissa* state in order to wait for the two constants (length $p(x)$ and length $p(x, y)$) and the mantissa vector to be retrieved.

- ***init mantissa* state:** This state is responsible for the initialization of the first of the two BRAM's in the *logarithm* unit. This state is responsible for counting how many elements of the mantissa vector have been stored in the BRAM. When the

counting finished and the data stored to the BRAM, the Main Control Unit receives the acknowledgement signals in order to proceed to the initialization of the second BRAM of the *logarithm* unit.

- ***init exponent* state:** This state is responsible for the initialization of the second BRAM in the *logarithm* unit in the pipeline. This state is similarly the same with the previous state with the only difference that the Main Control Unit updates the *MC logic 3* with the first address of the initialization data block for the exponent vector and its size.

- ***wait for non empty fifos* state:** After the initialization of the system, the logic modules updated with the first addresses and their sizes of the vectors of Mutual and joint Probability Density Functions which are allocated in the shared memory and they start to request addresses in the calculated address space of each vector. Instead of *feeding* the inputs of the pipeline, the responded data stored in block RAM's configured as FIFO's. When the Main Control Unit updated that the FIFO's hold a sufficient number of elements in order to start the processing the Main Control Unit proceed to the next processing state.

- ***processing* state:** This is the most important state of the Main Control Unit. This state is the state which controls all the memory elements configured as FIFO's before the pipeline and the pipeline instead. Because the iterations for the MI calculation is known from the lengths of the PDF functions, the number of iterations which the system implements is predefined. So in this state when the FIFO's hold a sufficient number of elements in order to process simultaneously the three vectors, an iteration is done. We note that all individual components of this architecture are finely tuned in order to receive new data for processing once every clock cycle in order to eliminate the need for stalls for the purpose of avoiding possible hazards. So after the proper number of iterations done in order to calculate the MI this states proceed to the next state.

- ***wait for finish signal* state:** After the proper number of iterations this state waits for an acknowledgement signal from the pipeline when the result is ready for store. When this signal arises, the Main Control Unit proceed to the next state but in this state the *MC logic 1* updated with the address of the shared memory in which the result will be stored.

- ***store* state:** This is the state in which the store request implemented from our architecture to the shared memory. This state waits for the *MC logic 1* to implement a store request to the shared memory and when the store is successful proceed to the next state.

- ***finish* state:** This is the last state of the Main Control Unit and this state is responsible to check that there are not any requests from and to the shared memory which are out of date and sends a finish signal to the host processor in order to finish the processing in the co-processor of the system and return the execution of our application in the host side.

The figure 4.8 illustrates the FSM which described up to this point, in order to allowed us to understand the main control of the system.

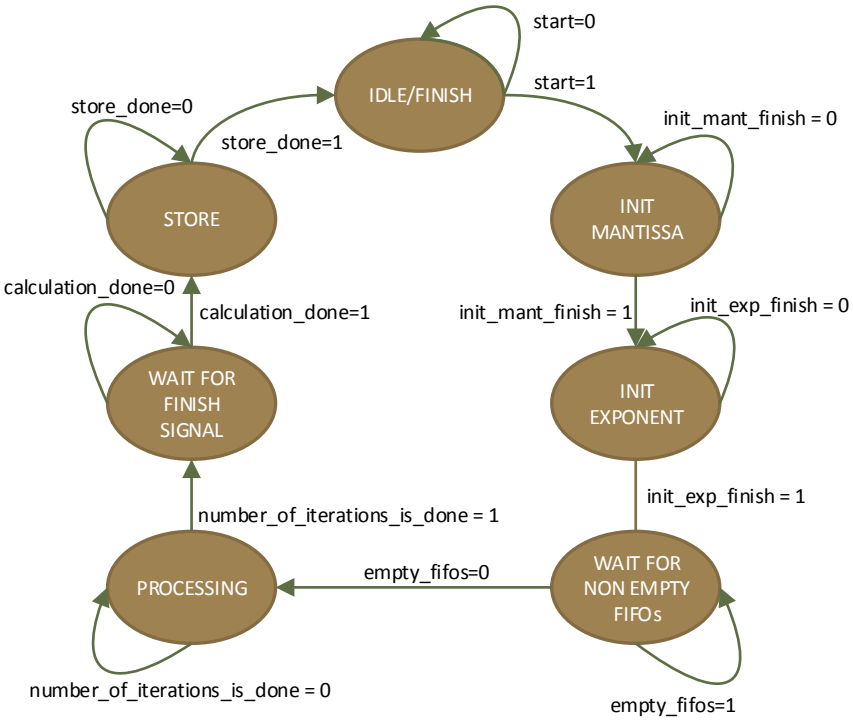

FIGURE 4.8: The Finite State Machine (FSM) of the Main Control of the system

4.1.4.4 Computation Core on Hardware Side

The hardware architecture which calculates the MI is shown in figure 4.9. As we described before, in our implementation we use two parallel special-purpose units to compute the partial MI, in order to divide the number of iterations by two. The figure illustrates one of these cores. Note that, in order to calculate the final MI we need two of these cores in parallel and the partial results of these two cores added through an adder unit which produces the final MI result.

As presented above, when the FIFO's hold a sufficient number of elements in order to start the processing, which means that one element from every PDF vector is ready to inserted to the pipeline, the processing starts and these values are processed in the pipeline. Note that the MI hardware architecture is fully pipelined, which means that if and only if, there are no exist stalls, an iteration of the MI calculation is performed every clock cycle.

More specifically about stalls, a stall requested from the Main Control Unit to the pipeline when one element from a vector can not inserted to the pipeline simultaneously with the elements of the other two vectors. In this case, the iteration can not implemented from the pipeline and all the units of the pipeline wait until the elements of the current iteration processed simultaneously. All the basic arithmetic functions use single-precision floating point and implemented with the Xilinx Core Generator.

The *logarithm* unit and the *Sum* Unit consists of more complex arithmetic operations and are described in detail in the following sections.

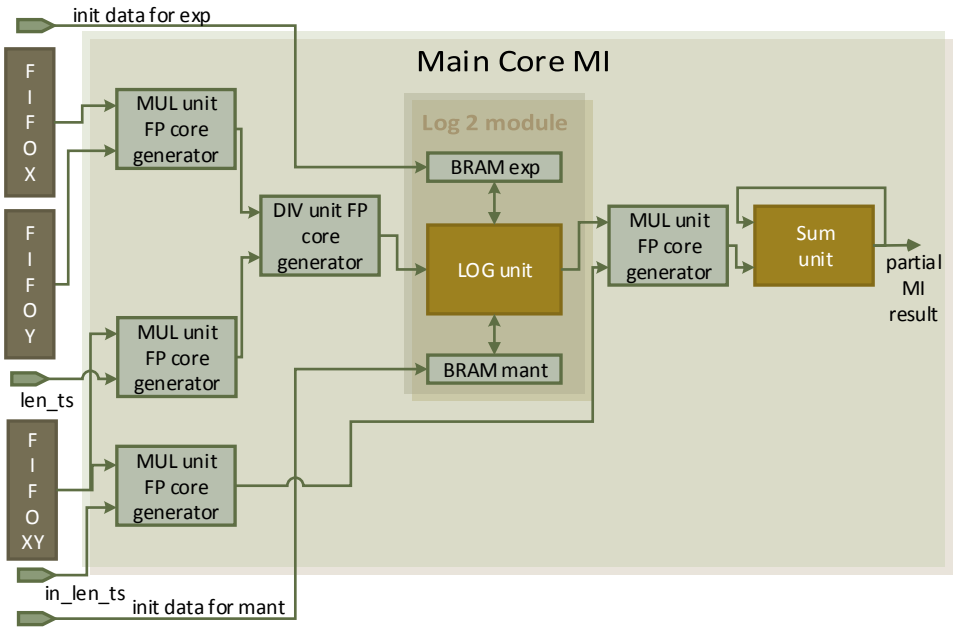

FIGURE 4.9: Mutual Information Hardware Core Architecture

4.1.4.5 Sum Unit

The *Sum* unit, is the unit which is responsible for accumulating the partial results. It consists of three modules, 2 single-precision floating point adders and one FIFO. The main hardware implementation of the *Sum* unit illustrated in the figure 4.10. The 2 single-precision floating point adders are fully pipelined with latency of 11 clock cycles, which means that the first result of the adder after the insertion of the data in the *Sum* unit will be observed after 11 clock cycles. To be more specific the calculation of the MI demands one element for each $p(x, y)$, $p(y)$ and $p(x)$ should be streamed every *"adder latency"* clock cycles. Indisputably, this idea is not optimized because it implements a *"adder latency"* times slower system architecture. In order to overcome this obstacle, an instance per stream is inputted every clock cycle, accumulated and stored in the *"adder latency"*-depth pipeline buffer, but with the first slot of the buffer containing the results of instances 1 + 12 + ... and so on. Similarly the second slot of the buffer contains the partial results of inputs 2 + 13 + .. and so on. This applies for all the 11 slots of the feedback buffer. Afterwards, each slot of the buffer will contain the final 11 partial sums that are streamed to the final adder in order to sum the final 11 partial sums. Also this adder has a latency of 11 clock cycles. All the above functionality controlled by the *Sum* Unit Control which controlled by the Main Control Unit.

In order to yield a better overall understanding the *Sum* Unit Control consists of 5 states. The functionality of the *Sum* Unit Control is as follows:

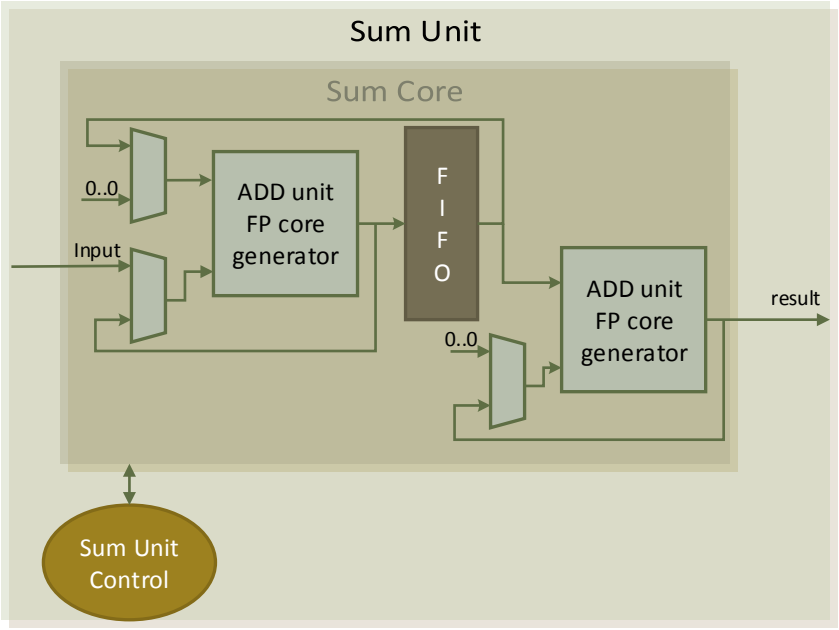

FIGURE 4.10: The Sum Unit of the Mutual Information Hardware core

- ***idle* state :** In this state the *Sum* unit waits for a signal which asserted when the first element of the stream is ready to inserted to the *Sum* unit.

- ***init* state :** This state waits for *"adder latency"* clock cycles for the adder core in order to produce the first result after the insertion of the first element of the stream. Note that in the first iteration and until the first elements stored in the FIFO the second port of the adder is connected with zero via a multiplexer.

- ***writestart* state :** After the *latency* clock cycles the first result of the adder stored in the FIFO.So after *"adder latency"* clock cycles + *"adder latency"* clock cycles the sum unit control is ready to proceed to the next state. To conclude this state is responsible for filling the FIFO with the first 11 results of the *add* unit.

- ***readwritestart* state :** After the writestart state, the Sum Unit Control proceed to the next state. In the *readwritestart* state, in every clock cycle one element was read from the FIFO and added with the current element of the stream which inserted in the *sum* unit and the current result stored to the FIFO. This state reads one element from the FIFO and stores the current result of the adder for *length of stream* + 22 cycles. Note that 22 is the latency of the adder and the latency of the fifo to fill up with elements.

- ***finalsum* state :** In the end of *readwritestart* state, the FIFO holds the last 11 partial sums in order to calculate the final MI. In this state the 11 slots of the buffer added in *add* unit as illustrated in the figure 4.10.

4.1.4.6 Logarithm Unit

The Logarithm Unit is illustrated in figure 4.11. In this unit we use the architecture for floating-point logarithm calculation with a piece of changes in our implementation, designed by Nikolaos Alachiotis and Alexandros Stamatakis which described in [25]. The open-source VHDL implementation of logarithm approximation unit for FPGAs downloaded from here.The architecture of the Logarithm Unit which proposed, can be easily adjusted to the desired accuracy, and their implementation is based on the ICSILog approximation method. More specifically an IEEE floating point number consists of three fields: the *sign(sgn)*, the exponent(exp), and the *mantissa (man)*. The decimal floating point value of a number *(num)* is represented as follows:

$$num = (+/-)2^{exp} \times man \tag{4.1}$$

In order to calculate the logarithm of *num*, one can use the multiplicative property of the logarithmic function and decompose the computation as follows:

$$\begin{aligned} \log(num) &= \log(2^{exp} \times man) && (4.2)\\ &= \log(2^{exp}) + \log(man) && (4.3)\\ &= exp \times \log(2) + \log(man) && (4.4) \end{aligned}$$

Since the real-valued logarithm is only defined for positive numbers, the sign bit can be discarded. The factor by which exp is multiplied is a constant and only depends on the base of the logarithm. Thus, the calculation of the logarithm for an arbitrary base *x*, only requires the constant $\log_x(2)$ and an appropriately initialized full-size LUT (comprising all values) for the base *x*. As we observe, the calculation has two parts the first part is the $exp \times \log(2)$ and the second part is $\log(man)$. The calculation of the first part of the sum requires the floating point representation for the decimal value of the exponent field. As presented in the figure 4.11 in order to calculate the first part of the equation 4.2, the proposed architecture use a separate LUT that is exclusively used for this conversion.

The implementation is based on the observation that there is a correspondence between the decimal values of the exponent field and the exponents themselves. For single precision the exponent range is from -126 to $+127$. In order to reduce the size of the exp_LUT by 50% the proposed architecture stores only the bits required to represent floating point numbers in the range $0 - 126$. To support the full range uses additional logic as described in figure 4.11. After this transformation, the resulting floating point number becomes the first operand of the multiplication. The second operand is a constant value. The second part of the equation 4.2 calculated by the man_LUT module as illustrated in 4.11. The man_LUT is the standard quantized LUT of the ICSILog algorithm and contains pre-calculated values of logarithms. The most significant bits of the mantissa are used for indexing the man_LUT. Each entry of the table consists of a single precision floating point number. After that the two calculated parts proceed to the add unit in order to calculate the final result. The leftmost module is the *special_case_detector*. As the name suggests, this module assesses whether the input to the LAU is valid or not as defined by the IEEE standard.Both lookup tables are enhanced

by a *construct_sp_fp_value* unit.These units consist of logic gates, registers, and multiplexers which are used to construct the correct floating point representations from the respective LUT entries.

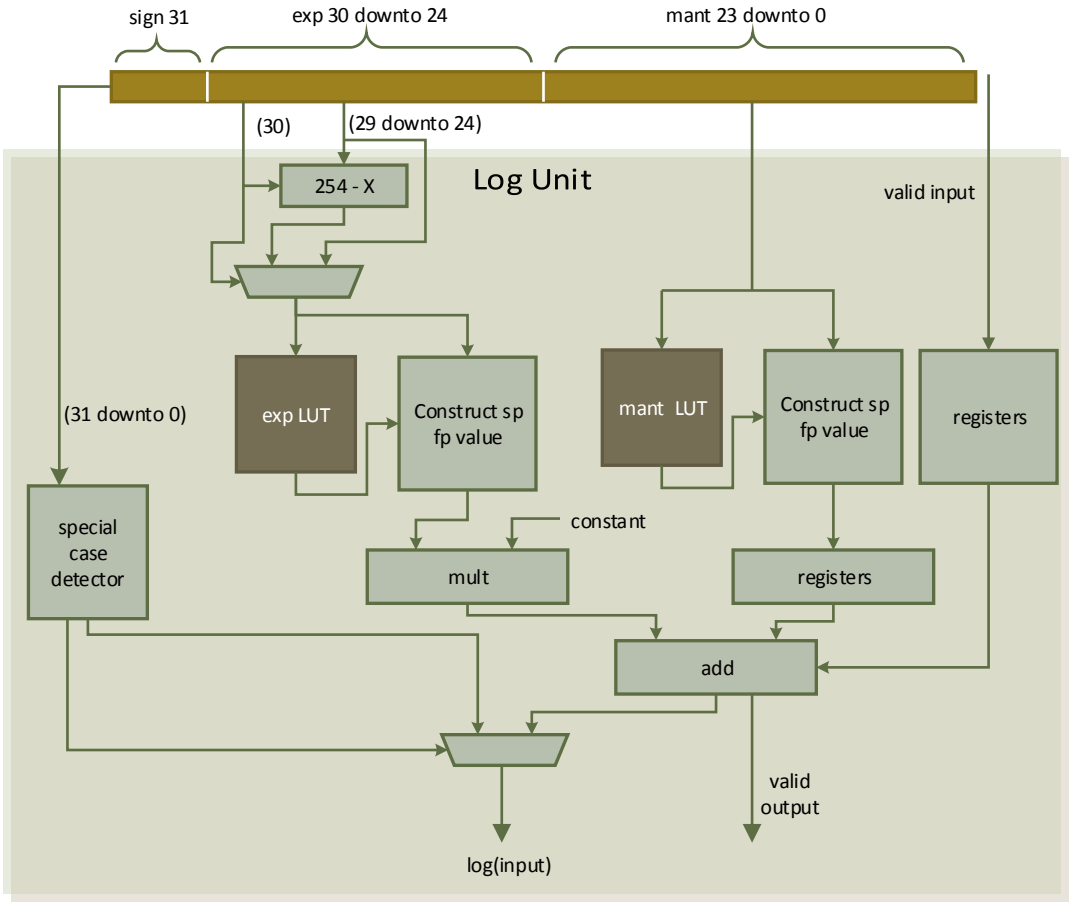

FIGURE 4.11: The Logarithm Unit of the Mutual Information Hardware core

To adapt the architecture of Logarithm Unit to our implementation, some changes in the Logarithm Unit are essential. In order to make our implementation fully parameterized, the initialization of the BRAM's without the use of *COE files* was a matter of necessity. In our architecture, we load the initialization files in the host side from the C-based application, and hold them in the shared memory of the system. Afterwards in the hardware side, in the initialization phase, we fetch the elements for the two BRAM's in order to initialize them. The downloaded version of Logarithm Unit also makes available several *COE files* that can be used to initialize LUTs of various sizes and hence easily adapt the LAUs to the desired accuracy level, so these *COE files* can be loaded in our implementation and in the initialization phase we can control the desired accuracy of our result. In order to implement this functionality in our system, the use of additional logic and the use of multiplexers in the inputs of the Logarithm

Unit were a matter of necessity. It is inevitable result that the size of the COE files are amenable to the BRAM's size. Note that if the LUT size is increased, the speed will only slightly decrease. Also as the authors of the paper present, a LUT with 4,096 entries represents a good trade-off between accuracy and LUT size. In our implementation we use LUT with 4,096 entries and the result for our application was providing a sufficient accuracy.

The authors of the paper proposed specific LUT for base e. So the result of Logarithm Unit is $log_e(input)$. To the best of our knowledge, the only open-source logarithm for FPGAs is the one provided by FloPoCo [6]. One can use FloPoCo [6] to generate the aforementioned logarithm implementation, but in our case because of compatibility issues with the Convey platform, we implemented the Logarithm Unit as proposed in the paper, in order to calculate the $log_e(input)$ and then using the change of base formula we can calculate the $log_2(input)$. The change of base formula implemented as follows:

$$\log_b(x) = \frac{\log_d(x)}{\log_d(b)} \tag{4.5}$$

4.1.5 Multi-core Architecture

In the previous sections we presented the architecture for one core, which means that we calculate one MI value in one FPGA of the Convey HC-2ex.

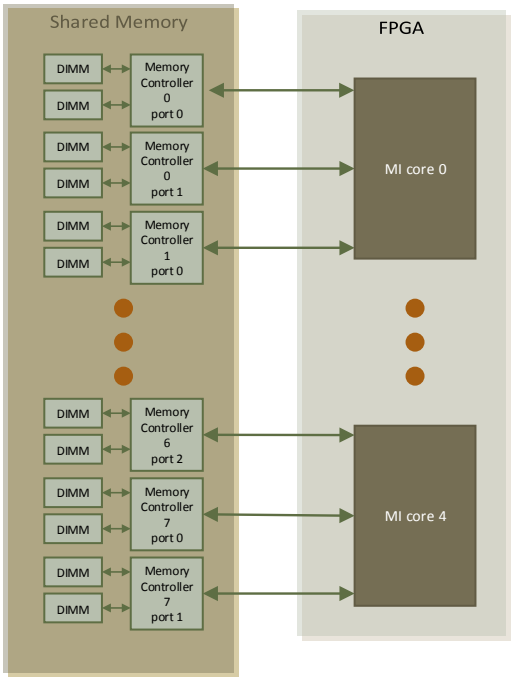

FIGURE 4.12: The Multi-core architecture of the Mutual Information

In terms of parallelism, as long as the FPGA resources allow us to map more than one core and memory controllers are available, we can map more than one core of MI

algorithm in the FPGA. Taking into account all the above, we expand our architecture and we manage to map up to 5 cores per FPGA. The figure 4.12 illustrates the Multi-core architecture of MI algorithm. As we can observe from the figure, every FPGA of the system exploits the 16 total memory controller ports. Each core is connected to three ports of memory controllers. The basic architecture of each MI core illustrated in 4.5. On the Host side with the C-based application, the arrays of constants for each MI core allocated in the shared memory as provided in figure 4.3. On the FPGA side, the top-level of the architecture is responsible for the routing of the first address to the contiguous memory space in which the PDF vectors for each MI core are located, as well as for the routing of the first address of the contiguous memory space that the output data will be stored. Note that in the Multi-core architecture each MI result stored in the corresponding address of a result array. Also the top-level is responsible to fetch the addresses of the *init mantissa array* and *init exponent array* which are the arrays with the initialization data for the BRAM's of Logarithm Unit which are common to each MI core. Furthermore in order to calculate the MI, each core needs the addresses of the constants in order to fetch the lengths of the vectors. After the distribution of the above addresses we proceed to the initialization phase. In the initialization phase as long as the distribution of the addresses has already done, each MI core fetches the lengths of the PDF vectors and the initialization arrays for the BRAM's of each Logarithm Unit and the processing is ready to start. For each MI core the time series lengths are different as well as the PDF vectors lengths are different. As a result some cores produce the final MI result faster than others. Our architecture is designed to wait all the MI cores to finish, store the result back to the shared memory and then the processing returns to the C-based application.The figure 4.12 illustrates the Multi-core architecture of MI algorithm.

4.1.6 Multi-FPGA Architecture

In the previous section we provided the Multi-core architecture of the MI algorithm. This architecture can calculate 5 different MI results. The Convey HC-2ex is equipped with 4 Virtex6 LX760 FPGA. The same bitfile must be downloaded on all FPGAs in order to use them. So in the Multi-FPGA architecture we can produce up to 20 MI results. When the host requests through the *copcall* function, the initial addresses for each memory controller are assigned through a short assembly script, which also routes the constants to the corresponding ports of the architecture. Since only one pointer is provided for the input data memory space, this script implements scalar instructions to determine the correct initial address for each of the two ports of each memory controller through the use of a predefined step depending on the dataset size. The script ends by notifying the co-processor that all data and initial addresses are in order, thus the AEs can start processing the dataset. Note that each assembly instruction may be formed to refer to any number of AEs, therefore the user may choose how many AEs will share the workload. The figure 4.13 illustrates the Multi-FPGA architecture of MI algorithm.

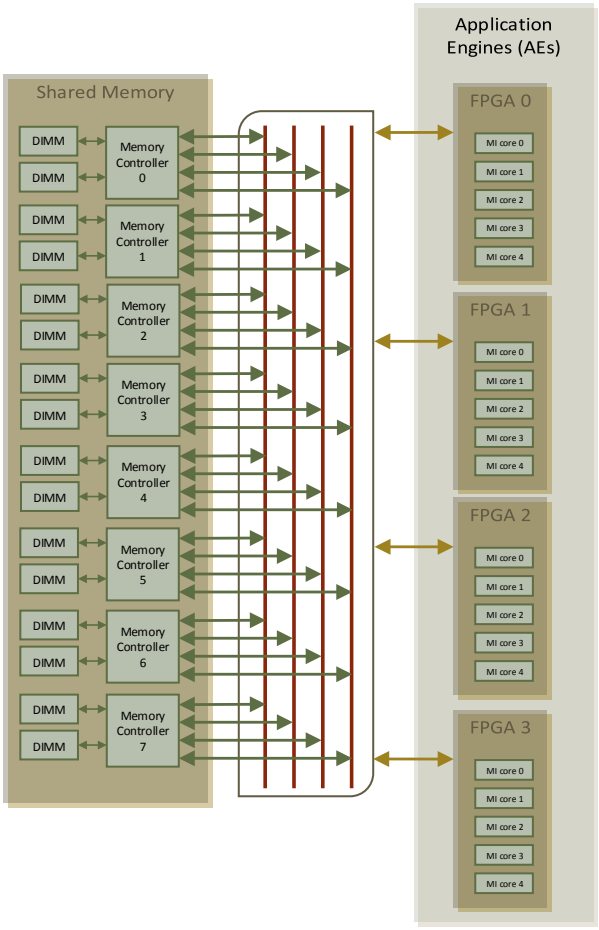

FIGURE 4.13: The multi-FPGA architecture of the Mutual Information algorithm

4.2 Transfer Entropy

As described in the equations 2.21 and 2.22 the algorithm of Transfer Entropy takes as input two random variables in the form of time series and produces one value which is the result of the Transfer Entropy of these two random variables. Transfer Entropy calculation is a computationally intensive algorithm with $O(R^3)$ complexity, making an excellent candidate for implementation on heterogeneous hardware.

4.2.1 Software Profiling of TE Algorithm

In order to optimize the calculation of the Transfer Entropy, it is necessary to profiling the software algorithm as we already done with the Mutual Information algorithm, in order to find the most-computationally intensive part of the algorithm and accelerate it on hardware.

Before to proceed to the calculation of Transfer Entropy algorithm, the estimation of Probability Density Functions (PDF) are necessary. For TE, just like Mutual Information, histograms were used for PDF estimation. The output of the histograms are

$p(x), p(x,y), p(x_{n+1}, x)$ and $p(x_{n+1}, x, y)$. These functions are represented by a 1-D array for $p(x)$, two 2-D arrays for $p(x,y)$ and $p(x_{n+1}, x)$, and one 3-D array for $p(x_{n+1}, x, y)$. The result is the estimation of the PDF's functions given two random variables. The table 4.3 shows the dependency between the execution time and the size of the inputs for the PDF estimation keeping the resolution equal to 1200 because this value is the maximum value for the resolution we can reach in our experiments. As we observe from the table 4.3, we conclude that the execution time for PDF estimation increases linearly with the input size just like the Mutual Information Algorithm.

On the other side, on the table 4.4 we can conclude that increasing the number of bins, can cause the cube increase of the execution time, keeping the size of time series to a fixed value of 100000 elements. Furthermore from the equations 2.21 and 2.22 goes without saying that the time complexity for the TE computation is $O(R^3)$. To provide a sufficient understanding, just like with Mutual Information, for very small number of Bins and very large time series lengths the PDF estimation can be more time consuming but for example for 1200 bins and for size of time series equal to 100000 elements, the TE computation requires 24.5 sec, whereas the creation of the histogram only needs 0.0062 sec which means that the most of the time consumed to the calculation of Transfer Entropy.

TABLE 4.3: Dependency between the execution time and the size of the inputs for the PDF estimation of the Transfer Entropy Algorithm.

Time series(Elements)	Execution Time(sec)
10000	0.00059
100000	0.0062
1000000	0.065
10000000	0.71
100000000	7.8
1000000000	72.3

TABLE 4.4: Dependency between the execution time and the number of bins for the Transfer Entropy Algorithm.

Number of Bins(R)	Execution Time(sec)
100	0.05
200	0.28
500	3.51
800	12.8
1000	24.5
1200	40.9

To conclude, based on the tables 4.3 and 4.4, we can assume that the most computationally intensive part of the TE algorithm is the final computation of this statistic value, just like the MI algorithm. As a result we have been summoned to accelerate the TE calculation between two random variables.

4.2.2 CPU and FPGA Integrated System of TE Architecture

The splitting of TE Algorithm is the same with the splitting of MI algorithm. So we follow the same steps with the MI algorithm and partitioning the TE algorithm in two parts. The first part is the PDF estimation and the second part is the computationally intensive part of the TE algorithm which is the final computation of its statistic value, which means that the first part of the algorithm was implemented on the host side of the system and the second part was mapped on the hardware side, just like the MI algorithm. The figure 4.14, illustrates the basic System Architecture of Transfer Entropy algorithm. As we can observe from the figure 4.14, the inputs in the PDF stage are the random variables X,Y in form of time series and the outputs are the probability vectors $p(x)$, $p(x,y)$, $p(x_{n+1},x)$, $p(x_{n+1},x,y)$. The flowchart that corresponds to the TE calculation is exactly the same with the flowchart of MI algorithm which illustrated in figure 4.2.

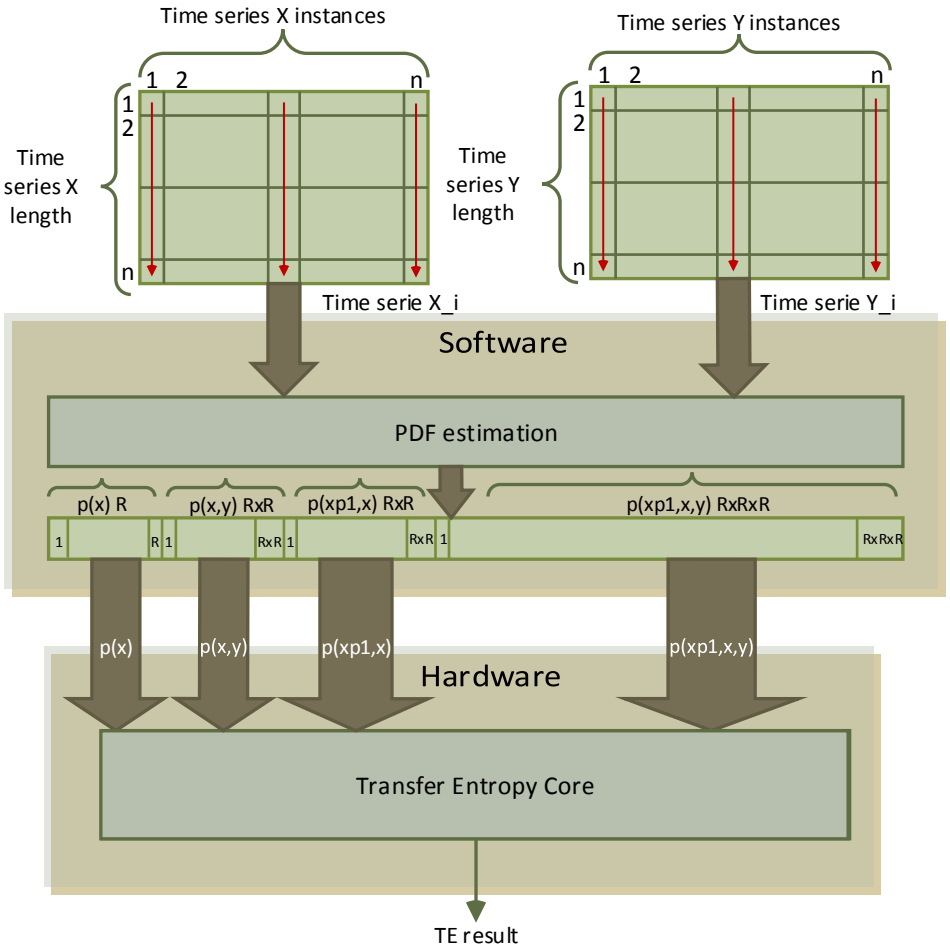

FIGURE 4.14: Basic System Architecture of Transfer Entropy

4.2.3 Pre-processing on Host Processor

Due to the similarity of MI algorithm and TE algorithm, the Pre-processing phase on Host Processor for TE algorithm is similar with the Pre-processing phase of MI algorithm which described in section 4.1.3. So the only difference between the two Pre-processing phases is that the PDF estimation for TE algorithm creates different probability vectors with different lengths. To be more specific as we described in the previous section the TE PDF estimation phase produces 1-D array for $p(x)$, two 2-D arrays for $p(x, y)$ and $p(x_{n+1}, x)$, and one 3-D array for $p(x_{n+1}, x, y)$. After the allocation of the vectors and the initialization constants, a function call to the co-processor is ready. As we described in section 4.1.3 the two first arguments of the function call are the pointers to the PDF vector and the initialization vectors respectively. The rest of the values are constants for each architecture of TE core. After the necessary allocations,the software is ready to call the co-processor so at this point, a function call implemented through the convey-specific $copcall fmt()$ function.The figure 4.15 illustrates the $copcall fmt()$ with the attributes for the TE calculation.

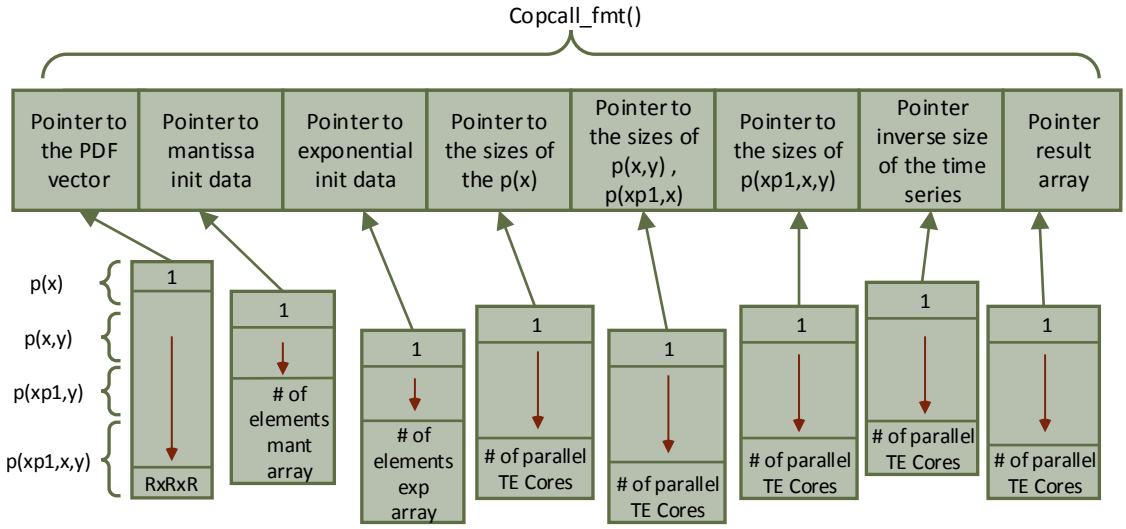

FIGURE 4.15: copcallfmt() function call with attributes of Transfer Entropy

4.2.4 Hardware Architecture Instantiation

After the function call, the co-processor is ready to process the data and calculate the Transfer Entropy value given two random variables. In this section we present the basic architecture which mapped onto the FPGA's. Note that the Transfer Entropy and the Mutual Information algorithms are very similar in their implementation. As a result some sections which are describing the basic architecture of TE, presented as references to the implemented sections of MI architecture.

4.2.4.1 Decomposition of the Problem

From the software profiling, just like with the MI algorithm we conclude that when $p(x)$, $p(x,y)$, $p(x_{n+1},x)$, and $p(x_{n+1},x,y)$ vectors created, the final result is ready after R^3 clock cycles. So taking into account the idea of parallel computing, we can split $p(x,y)$, $p(x_{n+1},x,y)$ into two streams. Then, each half of $p(x,y)$, $p(x_{n+1},x,y)$ is processed simultaneously, as there is no dependence between the different levels of the distribution function. The system uses two parallel units to compute an TE value twice. Therefore, the hardware produces two partial TE, the first produced from the elements in the even positions of the vector and the second produced from the elements in the odd positions of the vector which are summed at the end of the partial calculation of the Transfer Entropy. With this simple consideration we divide the required time of R^3 by a factor of 2, which means finally $\frac{R^3}{2}$ cycles.

The figure 4.16 illustrates the idea of partitioning the TE algorithm in odd and even pipelined cores. We note that the hardware processing is fully pipelined, meaning that to produce correct partial TE values each cycle, the respective $p(x)$, $p(x,y)$, $p(x_{n+1},x)$,and $p(x_{n+1},x,y)$ vectors should arrive simultaneously at each processing unit of the TE calculation.

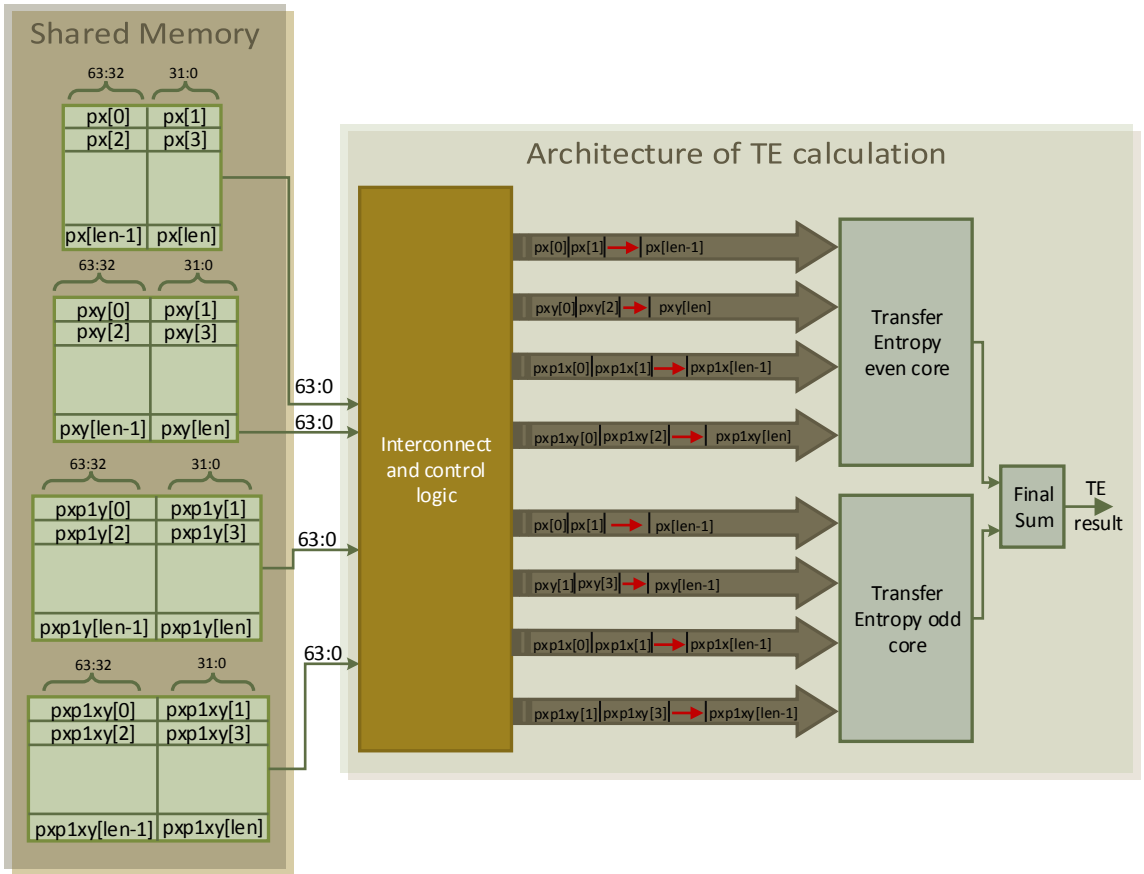

FIGURE 4.16: Partitioning of the Transfer Entropy Algorithm

4.2.4.2 Data Movement to the Hardware Side

This section describes how to retrieve and manipulate the necessary data vectors from the shared memory in order to calculate the TE. Every vector of $p(x)$, $p(x, y)$, $p(x_{n+1}, x)$, and $p(x_{n+1}, x, y)$, uses one memory controller port each. As a result in contrast with the MI architecture, we use four memory controllers, one for each PDF vector. Furthermore the four memory controllers for each special-purpose core are responsible to retrieve the constants which are essential for the computation.

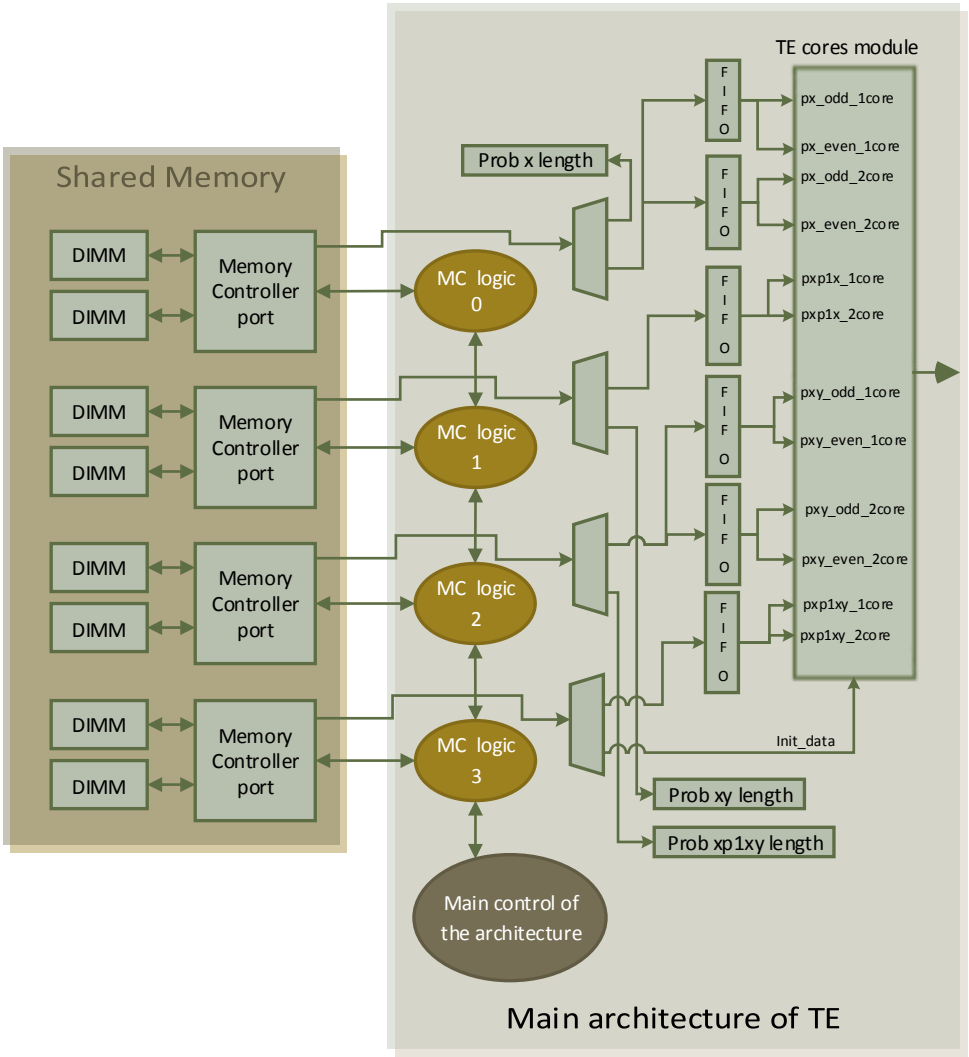

FIGURE 4.17: Connection between the architecture and shared memory of the Convey HC-2ex

Every port to the shared memory is connected to a logic module in our architecture, which acts in order to request data from the shared memory and manages all the responds and control signals from the memory. These logic modules controlled by the Main Control Unit which is responsible to control the system. A detailed description

of functionality of each logic module was provided in section 4.1.4.2. The only difference between the MI architecture and the TE architecture is the inclusion of another one logic module which controls another one memory controller port. The figure 4.17 illustrates the interconnect between the TE architecture and shared memory. Note that all the illustrated FIFO's consist of two 32-bit FIFO's. More specifically in the initialization phase, the *MC logic 1* and the *MC logic 2* has the same functionality with the MI architecture. The *MC logic 3* is responsible to fetch the length of the $p(x_{n+1}, x, y)$ and the *MC logic 4* is responsible for the initialization of the BRAM's for the Logarithm Unit. In the second phase the logic modules acquire the addresses of the PDF vectors. The *MC logic 1* fetches the vector $p(x)$, the *MC logic 2* fetches the $p(x_{n+1}, x)$ vector, the *MC logic 3* responds with the elements of $p(x, y)$ vector and the *MC logic 4* assigned to the $p(x_{n+1}, x, y)$ vector. The basic functionality of each logic module described in section 4.1.4.2. Note that the *MC logic 4* has exactly the same functionality with the *MC logic 3* module of the MI architecture.

4.2.4.3 The Main Control Unit

The Main Control Unit which illustrated in the figure 4.17 is the module which controls all the signals of the architecture. It is responsible for enabling the logic modules in order to fetch the data from the memory, manage the FIFO's of before the computational cores in order to process the data simultaneously as well as to control the cores in order to produce the partial TE results and finally to be added at the end. It consists of 7 states, all of them control a different part of the system. The detailed functionality of Main Control Unit was provided in the section 4.1.4.3 in which every state described individually.

4.2.4.4 Computation Core on Hardware Side

In this section we describe the basic hardware architecture for the calculation of TE. Because the MI algorithm is very similar with the TE algorithm the implementation of the pipelined core is based on the implementation in section 4.1.4.4. In our implementation we use two parallel special-purpose units to compute the partial TE just like the MI architecture, in order to divide the number of iterations by two. Note that, in order to calculate the final TE we need two of these cores in parallel and the partial results of these two cores added through a adder unit which produces the final TE result.The difference of the two algorithms is the size of the PDF vectors, which for MI is R^2 and for TE is R^3 and the additional vector as input. The figure 4.18 illustrates the basic core architecture of TE algorithm.

4.2.4.5 Sum Unit

The Sum unit of the TE architecture is responsible of accumulating the partial results. It consists of three modules, 2 single-precision floating point adders and one FIFO. The main hardware implementation of the Sum unit illustrated in the figure 4.10. The functionality of Sum unit controlled by the Sum Unit Control which is controlled by the Main Control Unit. The Main Control Unit consists of 5 states. Because of MI architecture and TE architecture use the exactly the same Sum unit a detailed description of this unit was provided in section 4.1.4.5.

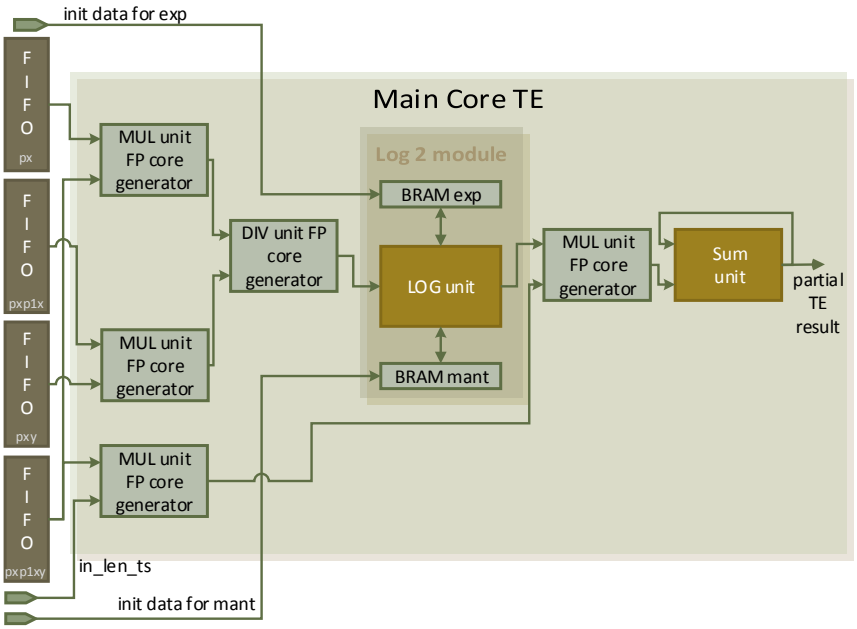

FIGURE 4.18: Transfer Entropy Hardware Core Architecture

4.2.4.6 Logarithm Unit

As we can observe in figure 4.18 in order to calculate TE a Logarithm Unit is needed. We use in our implementation the Logarithm Unit which proposed by Nikolaos Alachiotis and Alexandros Stamatakis and described in [25] as we use in MI architecture. One of the advantages of this architecture is the ability to easily adjusted to the desired accuracy. The architecture of Logarithm Unit is based on the ICSILog approximation method. The basic architecture as well as the changes which are made in order to adapt the architecture of Logarithm Unit to our implementation presented in the section 4.1.4.6 and the basic architecture of the Logarithm Unit illustrated in 4.11.

4.2.5 Multi-core Architecture

Up to this point we presented a detailed functionality of the TE architecture and how every unit of the architecture acts in order to calculate the TE result. In order to increase the throughput of the system we compute up to 4 different TE results in parallel in each FPGA. So we implemented an architecture in which we mapped 4 TE cores in one FPGA. This architecture is very similar with the multi-core MI architecture. The detailed functionality of multi-core architecture was provided in section 4.1.5. The only difference with the MI multi-core architecture is that in the TE multi-core architecture we mapped up to 4 cores in contrast with MI in which we use up to 5. This difference is based on the number of memory controllers which use each independent core. Another difference is the changes in the assembly script which are made in order to distribute the proper addresses to each FPGA and constants which are necessary

for the processing. The figure 4.19 illustrates the Multi-core Architecture for the TE algorithm.

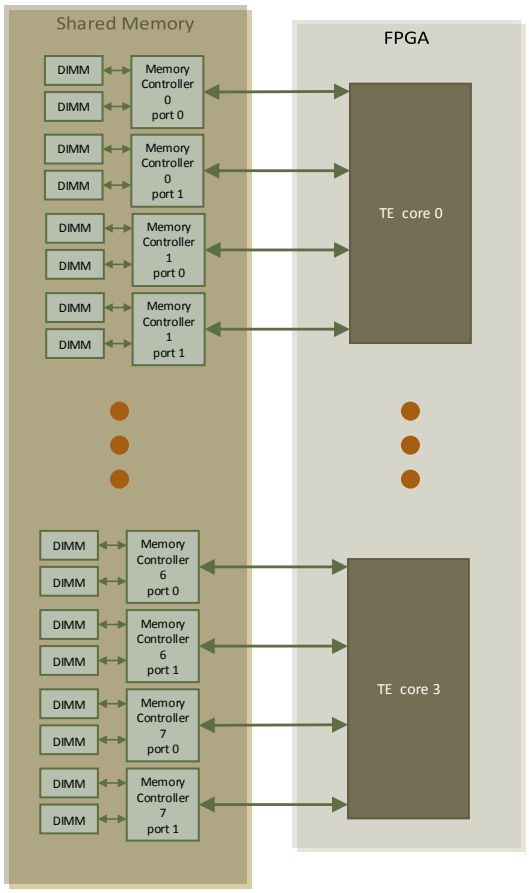

FIGURE 4.19: The Multi-core architecture of Transfer Entropy algorithm

4.2.6 Multi-FPGA Architecture

In order to entrench the 4 FPGAs of the system, we implemented a Multi-FPGA architecture. In order to map the Multi-core Architecture to the 4 FPGA's of the system the bitfile needs to distribute to all the FPGA's. A detailed functionality of this architecture was provided in section 4.1.6. Because the two algorithms are very similar between them, the Multi-FPGA architectures for these two algorithms are very similar too. The basic difference between the two Multi-FPGA architectures is that on the MI architecture we produce up to 20 different MI calculation but on the other side on the TE architecture we produce up to 16. The main reason is related with the number of memory controllers each core use which by extension means the best exploitation of the available bandwidth between the AEs and the shared memory. The figure 4.20 provides the Multi-FPGA architecture of TE algorithm.

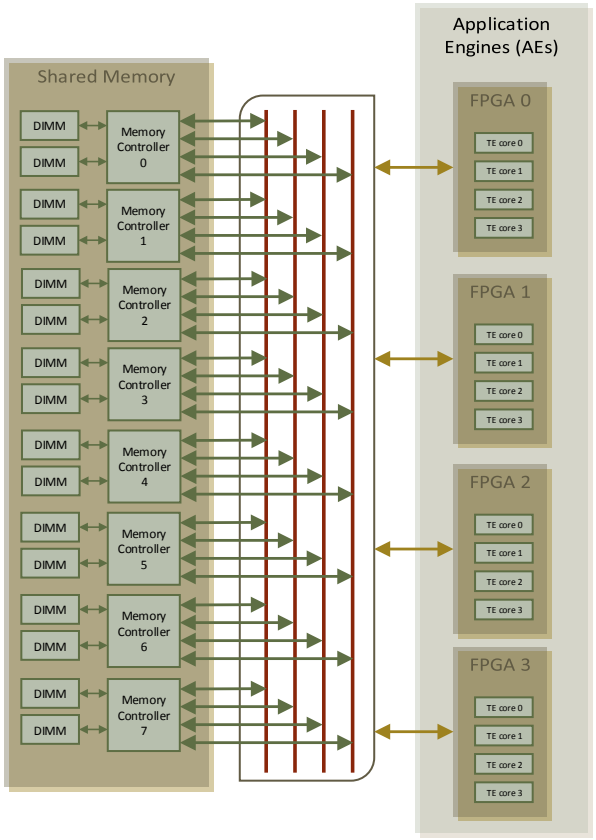

FIGURE 4.20: The Multi-FPGA architecture of Transfer Entropy algorithm

Chapter 5

Experimental Evaluation and System Verification

5.1 Setup

We implemented parallel cores of the computationally intensive algorithms of Mutual Information and Transfer Entropy on the Convey HC-2ex platform. The HC-2ex system, specifically, utilizes two Intel Xeon E5-2642 6-core as its host processor with 64 GB RAM, four Xilinx Virtex LX760 FPGAs on its co-processor unit and a total of 128 GB shared memory capable of providing up to 80 GB/s of memory bandwidth when the four FPGA's of the co-processor are deployed. All the architectures were compiled to run to the predefined from the system 150 MHz. As inputs in the two algorithms we used **artificial** time-series due to the fact that at the time this thesis was written, we wanted to be able to evaluate whether the final MI and TE result was correct. Also all the experiments yielded with 100000 random time-series length.

5.2 System Verification

After the implementation phase, the two architectures needs to be properly verified for correct operation and evaluated based on their performance in comparison with reference software when processing a wide array of datasets. We present the steps we follow for the verification of MI architecture. Because of the similarity of the MI and TE algorithms the verification of TE architecture is exactly the same with the verification of MI implementation.

The development process of the architecture design was accompanied by a number of simulations that verified the correct operation. Each individual module before it was connected to the rest of the architecture tested extensively using the Xilinx ISim 12.4 simulator. After the individual testing of the modules of the system, the fully implemented systems operation was at first verified using ModelSim SE-64 6.5c, along with a simulation infrastructure provided by Convey Computers. By performing extensive simulations of the system and its components, the correct and synchronized operation was assured before attempting to map it onto the actual platform, thus minimizing the chance of the latter malfunctioning. After the ensuring that the simulations of the architecture confirm the proper operation of the design in regards to result correctness, it was ready to mapped onto the AE FPGA's and yield the results. In order to reduce the risk of malfunction due to incorrect operation of the full system, the final verification process was subdivided into 4 stages. The first stage consists of the verification of the

software running on the host processor without a function call to the co-processor. It is a very important step which ensures that the dataset is handled correctly and the correct constants are calculated before attempting to pass them to the co-processor for processing. Note that as inputs in the two algorithms we used artificial time-series. The second verification stage addressed the proper retrieval of data to the AEs. This step was necessary because, an erroneous data management could very easily lead us to incorrect behavior from the architecture and, possibly, even system crash. In the third stage, as long as the retrieval of the data were valid, the processing of the data and the proper result were the main verification goals of this stage. That achieved by cross-referencing the produced results by small datasets with their corresponding known expected values calculated through the software implementation of each algorithm. The fourth and the last stage was the management and the store of result. In order to complete a store a piece of signals must controlled in order to a store request be successful back to the shared memory. So in this stage verified the control of store signals as well as verified the successful storing to the shared memory and the proper finish of the co-processor. Finally, with a working prototype in place, small-scale optimization improvements were implementing and attempting to reach higher operating clock frequencies, the achieved stable clock frequency is 149.5 MHz, with 150 MHz being an upper bound imposed by the vendor.

5.3 Experimental Evaluation

This section presents the performance of the hardware-based MI and TE calculation as well as the multi thread implementation on software. In order to yield a better overall understanding about the behavior of our application on the Convey HC-2ex we provide all the experiments we follow in order to yield the best possible performance.

5.3.1 Mutual Information

5.3.1.1 Multi-thread Software Implementation of MI

In order to explore the performance of the Mutual Information algorithm and yield an overall understanding of the performance of this algorithm we expand the reference software of MI in order to use multi-threading and evaluate this implementation in software. We achieved this transformation using OpenMP API. In order to observe the differences of a multi-thread execution on software only with our Multi-core and Multi-FPGA architectures on the selected hybrid system we created an application which calculates 12 MI cores in parallel on software. The yielded results provided on the table 5.1. As expected, the multi-thread implementation with four cores on software can not achieve a speedup of the theoretical 4x as well as the implementation of twelve cores on software can not achieve the theoretical 12x compared to the reference software because a decisive factor of the overall execution time of the application is the memory bandwidth of the system which is shared to the implemented threads of the application. However, as we can observe from the table 5.1 for a small number of bins the execution time of the application is very small so the performance of the multi-thread implementation is not very obvious. The platform software in which the experiments took place was a two 6core Intel Xeon at 3.2GHz with 50GB RAM.

TABLE 5.1: Performance of Single-core Mutual Information calculation compared with the performance of multi-threaded four-core and twelve -core Mutual Information on software with 12 MI-calls.

Resolution	single core SW	4 core SW	SpeedUp	12 core SW	SpeedUp
100	0.024	0.006	**4**	0.0027	**8.88**
500	0.144	0.042	**3.42**	0.020	**7.2**
1000	0.384	0.117	**3.28**	0.049	**7.83**
2000	1.32	0.42	**3.14**	0.180	**7.33**
5000	7.8	2.67	**2.92**	1.050	**7.42**
10000	30.96	10.7	**2.86**	4.3	**7.2**

5.3.1.2 Multi-thread Histogram Function

As we described in Chapter 4, the Host side runs the C-based application. A function-call to the co-processor proceed to the calculation of the desired result. On the Host side we decided after the software profiling analysis of the algorithm to implement the function which is responsible for the PDF vectors creation. Because our architecture calculates up to 16 MI results in parallel, the idea of parallel creation of PDF vectors on the Host side goes without saying that accelerates our application. So on the Host side, the PDF vectors creation implemented with OpenMP API. Depending on the number of parallel cores architecture, we use on the Host side the proper number of threads in order to accelerate the piece of code which is responsible for the PDF vectors creation. For example if we have an architecture with 16-cores hardware implementation, we use 16 threads on the Host side in order to create the PDF vectors in parallel and accelerate the function. Each thread on the Host side assumes the PDF creation for one core. The table 5.2 represents the execution time only for the PDF creation function in order to yield a better overall understanding. To be more specific the first column represents the Resolution of the Histogram. The other columns represent the execution time in seconds for the PDF vectors creation for 2,4,8 and 16 PDF vectors creation with single execution respectively.

TABLE 5.2: Performance of PDF creation function on the Host side with single-thread execution of Mutual Information Algorithm.

Resolution	**Hist-func(2)**	**Hist-func(4)**	**Hist-func(8)**	**Hist-func(16)**
100	0.006451	0.01268	0.024	0.05
500	0.007	0.0136	0.026	0.055
1000	0.0075	0.015	0.032	0.064
2000	0.011	0.02	0.05	0.1
5000	0.049	0.097	0.2	0.41
10000	0.17	0.3	0.73	1.45

The table 5.3 represents the execution time in seconds of the PDF vector creation with multi-thread execution. In this implementation each thread assumes the instance of one PDF vector creation. In the next sections we will use the multi-threaded architectures for the PDF vectors creation on the Host side in order to evaluate our architectures. As we can observe from the tables 5.2 and 5.3 the achieved speedup of each

implementation depending approximately from the number of different instances of the function for the PDF vector creation.

TABLE 5.3: Performance of PDF creation function on the Host side with multi-thread execution of Mutual Information Algorithm.

Resolution	**Hist-func(2)**	**Hist-func(4)**	**Hist-func(8)**	**Hist-func(16)**
100	0.00326	0.00327	0.0331	0.00341
500	0.00359	0.00361	0.0369	0.00376
1000	0.00378	0.00379	0.004	0.0042
2000	0.0055	0.00568	0.0062	0.00645
5000	0.0247	0.0248	0.025	0.02562
10000	0.0854	0.0855	0.0914	0.0923

5.3.1.3 Performance of MI Implementation

In this section we present the performance of the different implemented MI architectures and evaluate them with other implemented architectures. In the following tables, the column *Resolution* represents the Number of Bins. The column *Software* presents the execution time in seconds for the reference software. The column *Hist-func* represents the execution time in seconds in order to create the PDF vectors from the time-series on the Host processor, the column *Hardware* is the time in seconds required to implement the *copcall* function which means the Hardware time in the FPGA's and the *Memcpy* column is the time required to transfer the data to the shared memory in order to be available for the AEs. The column *overall* is the aggregation between the *Hist-func*, *Hardware* and *memcpy* columns in order to produce the overall execution time of our application. Finally the last column of the tables is the speedup between the reference software and our application.

To begin with, table 5.4 presents the results derived from the comparison between the Mutual Information software implementation and the single core-hardware approach. Note that the architecture which reflects to these results, is our first implemented architecture in which we were using the half bandwidth of the system. To be more specific the Convey HC-2ex can fetch 64-bit words every clock cycle. In our first steps in the exploration of the system and in order to be familiar with it, we were fetching in each FPGA, 32-bit words every clock cycle. Also in this implementation we are not partitioning the PDF vectors in order to split the workload in two parallel cores.

TABLE 5.4: Performance of single-core MI Calculation with half bandwidth usage in single FPGA.

Resolution	**Software**	Hist-func	Hardware	Memcpy	**Overall**	**SpeedUp**
100	0.002	0.0032	0.00015	0.00011	0.003473	0.57
500	0.012	0.00349	0.00176	0.00052	0.005775	2.07
1000	0.032	0.00367	0.0067	0.0017	0.01207	2.65
2000	0.11	0.0053	0.026	0.0066	0.0379	2.90
5000	0.65	0.024	0.16	0.04	0.224	2.90
10000	2.58	0.0843	0.66	0.16	0.9043	2.85

The Table 5.5 provides another comparison which take place between the reference software and a single-core MI of our application with full bandwidth usage which means that in this implementation we exploit all the available bandwidth of the system fetching every clock cycle 64-bit words, and splitting the PDF vectors as provided in Chapter 4 in order to partitioning the workload. The results comparing the tables 5.4 and 5.5 respectively are the expected, because in the second architecture we divide the execution time in FPGA by a factor of two as well as we use the full bandwidth of the system.

TABLE 5.5: Performance of single-core MI Calculation with full bandwidth usage in single FPGA.

Resolution	**Software**	Hist-func	Hardware	Memcpy	**Overall**	**SpeedUp**
100	0.002	0.0032	0.000123	0.00011	0.003438	0.58
500	0.012	0.00349	0.000924	0.00052	0.004936	2.43
1000	0.032	0.00367	0.0033	0.0017	0.00867	3.69
2000	0.11	0.0053	0.013	0.0066	0.00249	4.41
5000	0.65	0.024	0.083	0.04	0.147	4.42
10000	2.58	0.084	0.33	0.16	0.5743	4.49

Up to this point the yielded results provided by the single-core architecture of our MI architecture. As long as we evaluated the architecture that works perfectly with one core we proceed to design new architectures with more than one core in a single FPGA as well as to find the best mapped architecture in one FPGA based on the available resources of the Convey and expanded them to the four FPGA's of the system. The tables 5.6, 5.7, 5.8 present the yielded results from the two-core, four-core and five-core MI architectures respectively. Taking into account these tables we can observe that the yielded speedups from the four-core architecture with the five-core architecture are very close. As a result we assume that the best mapped architecture in a single FPGA is the four-core architecture for Mutual Information. In the next tables we present the multi-FPGA architectures. We expanded the architectures of two-core and four-core architectures in 4 FPGA's. The tables 5.9 and 5.10 provide the yielded results of eight-core and sixteen-core architectures respectively.

TABLE 5.6: Performance of 2-core MI Calculation in single FPGA with Multi-threaded PDF function.

Resolution	**Software**	Hist-func	Hardware	Memcpy	**Overall**	**SpeedUp**
100	0.002	0.003226	0.000123	0.000152	0.00353	1.13
500	0.012	0.00359	0.000924	0.00097	0.00548	4.37
1000	0.032	0.00378	0.0033	0.0034	0.0104	6.10
2000	0.11	0.055	0.013	0.013	0.0315	6.98
5000	0.65	0.0247	0.083	0.084	0.191	6.78
10000	2.58	0.0854	0.33	0.32	0.7354	7.01

The resource summary of all our architectures is shown in table 5.11. As clearly denoted by the table above, only a small fraction of the available resources of the FPGA are utilized for the implementation of the architecture. Thus, any attempt to implement

TABLE 5.7: Performance of 4-core MI Calculation in single FPGA with Multi-threaded PDF function.

Resolution	**Software**	Hist-func	Hardware	Memcpy	**Overall**	**SpeedUp**
100	0.002	0.00327	0.000123	0.000237	0.00363	2.20
500	0.012	0.00361	0.000924	0.0025	0.007034	6.82
1000	0.032	0.00379	0.0033	0.0067	0.01379	9.28
2000	0.11	0.0568	0.013	0.026	0.04468	9.84
5000	0.65	0.0248	0.083	0.16	0.2678	9.70
10000	2.58	0.085	0.33	0.63	1.0455	9.87

TABLE 5.8: Performance of 5-core MI Calculation in single FPGA with Multi-threaded PDF function.

Resolution	**Software**	Hist-func	Hardware	Memcpy	**Overall**	**SpeedUp**
100	0.002	0.00328	0.000123	0.000274	0.00367	2.71
500	0.012	0.00361	0.000924	0.0027	0.007234	8.29
1000	0.032	0.0038	0.0033	0.0084	0.015	10.32
2000	0.11	0.0062	0.013	0.032	0.051	10.74
5000	0.65	0.0249	0.083	0.2	0.307	10.55
10000	2.58	0.0863	0.33	0.81	1.226	10.51

TABLE 5.9: Performance of 2-core MI Calculation in 4 FPGA's with Multi-threaded PDF function.

Resolution	**Software**	Hist-func	Hardware	Memcpy	**Overall**	**SpeedUp**
100	0.002	0.00331	0.000197	0.000277	0.003784	4.22
500	0.012	0.00369	0.000978	0.0023	0.006968	13.77
1000	0.032	0.004	0.0038	0.013	0.0208	12.30
2000	0.11	0.0062	0.014	0.052	0.0722	12.18
5000	0.65	0.025	0.085	0.32	0.43	12.09
10000	2.58	0.0914	0.34	1.2	1.6314	12.65

TABLE 5.10: Performance of 4-core MI Calculation in 4 FPGA's with Multi-threaded PDF functions.

Resolution	**Software**	Hist-func	Hardware	Memcpy	**Overall**	**SpeedUp**
100	0.002	0.00341	0.000197	0.00046	0.004067	7.86
500	0.012	0.00376	0.000981	0.0065	0.011241	17.08
1000	0.032	0.0042	0.004	0.026	0.0342	14.97
2000	0.11	0.00645	0.016	0.1	0.12245	14.37
5000	0.65	0.02562	0.091	0.65	0.76662	13.56
10000	2.58	0.0923	0.358	2.6	3.0503	13.53

the proposed design is bound first by the available memory bandwidth and, if the latter is large enough,only then will the available resources become a hindrance.

Taking into account all the above tables, we can illustrate the performance of the Mutual Information implemented on the Convey HC-2ex.

TABLE 5.11: Resources summary of the Multi-core architectures of Mutual Information

		occupied slices	slice registers	Slice LUT	DSPs	BRAMs
1 core	used	23555	74785	72082	32	120
	percentage	**19%**	**7%**	**15%**	**3%**	**16%**
2 core	used	28178	92371	87745	64	160
	percentage	**23%**	**9%**	**18%**	**7%**	**22%**
4 core	used	38154	128699	119296	128	240
	percentage	**32%**	**13%**	**25%**	**14%**	**33%**
5 core	used	41987	145070	132823	160	260
	percentage	**35%**	**15%**	**28%**	**18%**	**36%**

The figure 5.1 illustrates the different performances of the different architectures for the MI algorithm.

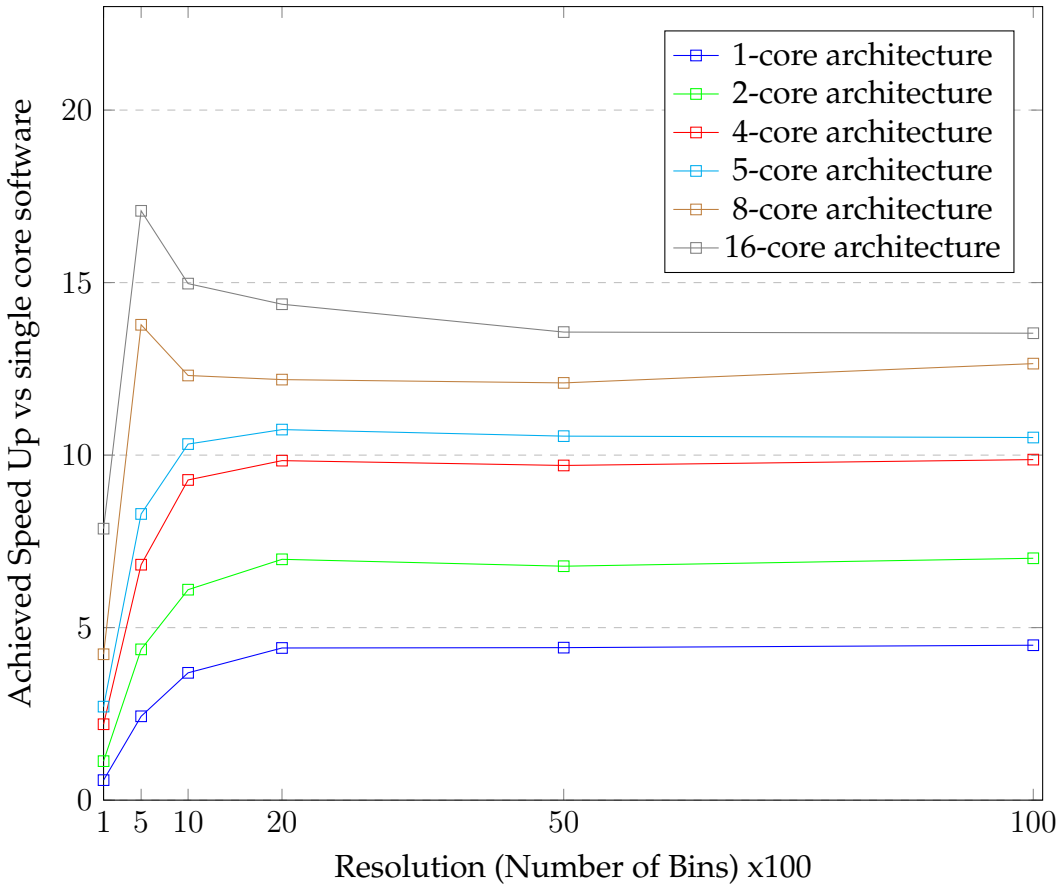

FIGURE 5.1: Achieved Speed Up for all the proposed architectures of MI Algorithm. Note that the scaling factor in x axis is x100 and represents the Resolution (Number of bins) of the MI algorithm.

As we can conclude from figure 5.1 the maximum speedup derived from the the architecture with 4 cores per FPGA when the 4 FPGA's are enabled. Also as observed from the waveforms until the resolution of the PDF vector reach the 1000 bins, all

the architectures achieve the highest speedup of their performance compared to the reference software. After reaching this value the speed up of each architecture settled down. The behavior of these waveforms attests that although we manage to keep the hardware time in low levels as well as we performed a multi threaded implementation in the PDF vector creation the memory copy of the data to the shared memory is a deciding factor of the overall time of our application because as the number of bins getting wide the memory copy time is the most consuming part of the execution time of our application.

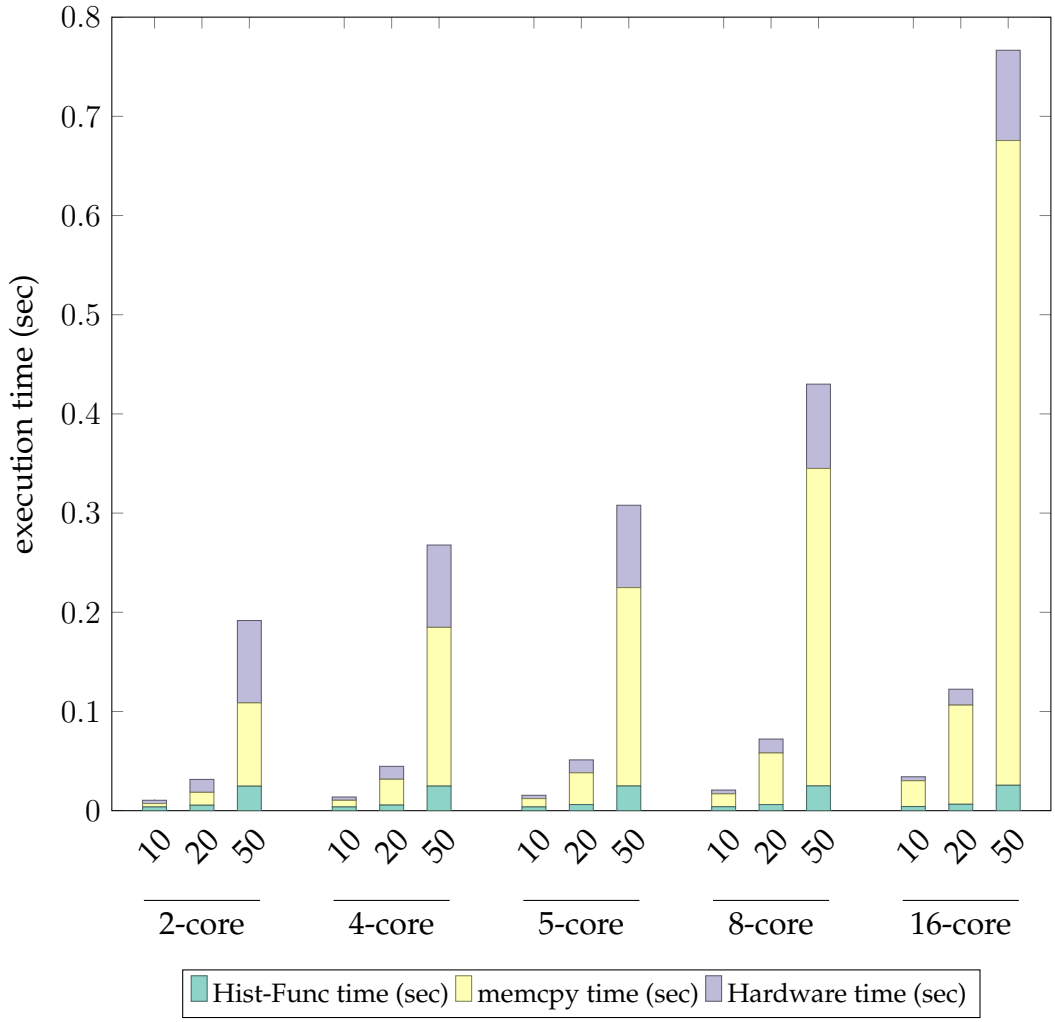

FIGURE 5.2: Time Consuming for all the proposed Architectures of MI Algorithm.The x axis represents the number of Bins of the PDF vector.Note that the scaling factor in x axis is x100

The figure 5.2 illustrates the distribution of the execution time in three main section of the application.The y axis is the execution time in seconds and x axis represents the performance of our architectures with different lengths of PDF vectors. Note that in the figure illustrated only one small range of bins. The first section of the application is the function which creates the PDF vectors.The second section consists of the transferring of the memory blocks to the shared memory in order to the vectors retrieved very fast from the FPGA's. The third section is the function call to the co-processor which

means the hardware execution time. As we can observe from the figure the most time consuming section of our application is the section which transfers the data to the shared memory of the system. In order to yield a better overall understanding, the figure 5.3 illustrates the performance of our architectures with a more wide number of of bins. The y axis represents the execution time in seconds but in *logarithm scale* in order to draw a conclusion about the most time consuming section of our application.

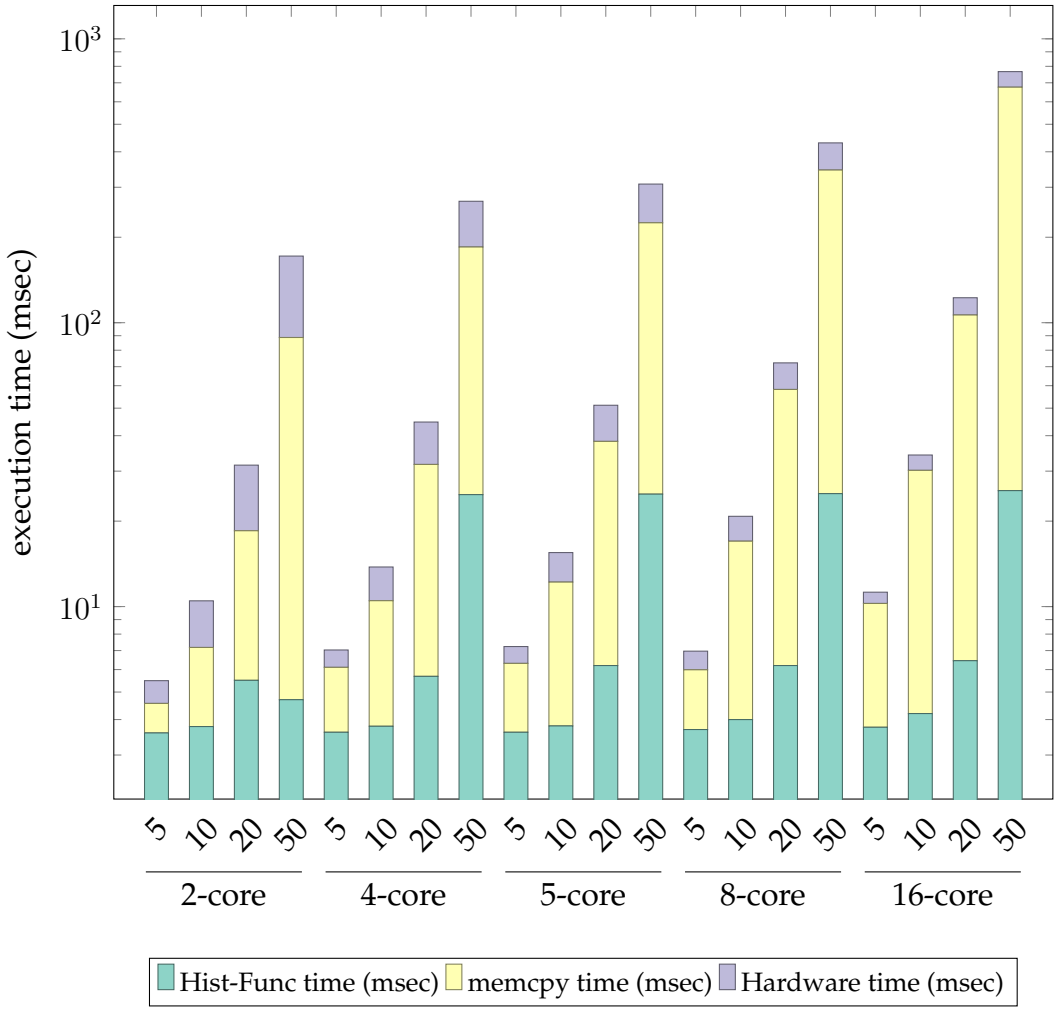

FIGURE 5.3: Time Consuming for all the proposed Architectures of MI Algorithm in logarithm scale.The x axis represents the number of Bins of the PDF vectors. Note that the scaling factor in x axis is x100

In order to reduce the time of memory copy to the shared memory a number of different implementations took place. We re-designed our application in the Host processor in order to implement it with threads. Our first idea was to partitioning the memory blocks and assign the transferring of these memory blocks in threads in order to transfer them in parallel. After the above implementation we conclude that the system could not transfer the memory blocks with the use of threads. Another idea was to partitioning the wide memory block with the PDF vector and processing smaller memory blocks. With this idea we were creating a pipeline and we were transferring a memory block. After the transferring of the first memory block we were processing

this memory block and simultaneously we were transferring the second memory block and etc. We came up with many obstacles with this implementation because the processing time was a drop in the bucket in relation with the transferring time. Also the required time to synchronize the pipeline of transferring and processing each memory block was very time consuming. So this implementation would be a very fruitful alternative but under circumstances. Last but not least we came up with the idea of transferring the memory block at once. To be more specific we wrote code in order to transfer a very wide array with all the PDF vectors at once and then using the assembly script to route all the necessary addresses to each FPGA. That happened because the transferring of a wide array over the PCIe is more efficient than to partitioning this wide array and transfer it in blocks.

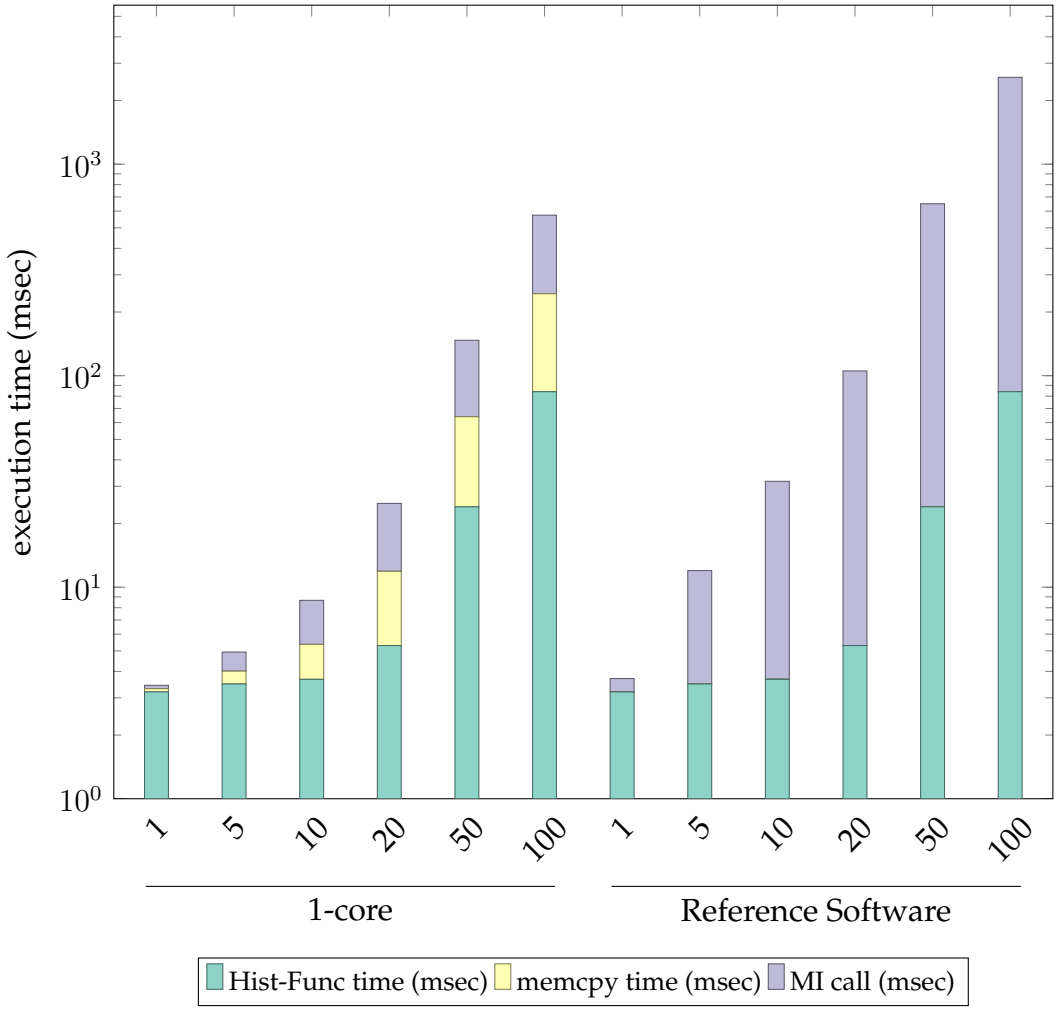

FIGURE 5.4: Time Consuming for the single core on single FPGA of MI Algorithm and the proposed reference software on logarithm scale. The x axis represents the number of Bins of the PDF vector.Note that the scaling factor in x axis is x100

The execution time of partitioning the memory block compared with the transferring this block at once are very close. That happened because from the scratch the vector partitioned in four wide memory blocks so the idea to transfer a very wide memory

block would be more efficient if from the scratch, the transferring memory blocks were much more than four.The difference between the partitioning these kind of wide arrays which used for our experiments, from the transferring them at once was insignificant. So taking into account all the above ideas, our effort to reduce the transferring time of the memory blocks to the shared memory was impossible. Nevertheless the execution time of our application was achieved a significant speed up compared to the reference software.

The figure 5.4 presents the time consuming parts of MI calculation implemented on Convey HC-2ex with one core on a single FPGA and the time consuming parts of MI in software in logarithm scale. As we can observe the execution time for the MI calculation is insignificant compared with the MI calculation in the software implementation. However as the number of bins and cores getting more wide, the most time consuming part of our application becomes the transferring of the memory blocks to the shared memory. We can observe that in this experiment with the one core calculation the most time consuming part of our application is the PDF estimation although in Multi-core and Multi-FPGA implementation the PDF estimation compared with the transferring time of the memory block to the shared memory is insignificant.

The main motivation of the implementation of these algorithms in a system like Convey HC-2ex, it was to push to the limits the system with computationally intensive algorithms and to evaluate its performance in relation with the same algorithms in other platforms. Based on the implementations of the MI algorithm which described in the Qualimaster project [16] in the deliverables [17] , [18] and [19].

The figure 5.5 illustrates the achieved speedups compared to the reference software. The architectures of MI algorithm mapped on a MAX3A Vectis PCI Express card version of a Maxeler Dataflow Supercomputing System. The MAX3A Vectis card is inserted into a base workstation with an Intel XEON 2-6 core processor running at a clock rate of 3.2 GHz and with 50GB RAM. MAX3A Vectis card is equipped with a Xilinx Virtex-6 FPGA device and 24 GB of DDR RAM. At the other end of the spectrum, another implementation of the MI algorithm mapped on a newer generation of Maxeler, the Maia platform. Maxeler MPC-X consists of 8 Altera Stratix V FPGA devices and 48 GB RAM. The Maia card has the same maximum PCIe bandwidth of 2GB/s and the same 24 GB internal memory (LMEM) as the Vectis platform. The increased resources, provided by the Altera FPGA which allows the implementation of up to 6 parallel memory controllers,which improve the LMEM bandwidth. The derived results from the platform of Maia and from the MAX3A Vectis are *estimated* results as derived from the deliverables of the Qualimaster Project.

Our implementation with the implementation on the MAX32 Vectis are very close based on their features. Also if we evaluate the two platforms we conclude that they are very close in performance. Furthermore the FPGA's which are equipped the two platforms they belong in the same family devices of Xilinx. As we can observe from the figure the Maxeler platform presents a higher overhead on MI call compared to the Convey HC-2ex which yields a higher speedup on smaller number of bins. As the number of bins getting more wide the transferring time to the shared memory of Convey HC-2ex getting the decisive factor of the overall application time. In terms of evaluation the Maia platform is a new device so it has a significant advantage compared to the other platforms and also there is a main difference in the scope of architecture. The MI as well as the TE architectures in Maia streamed a piece of the input vectors from a

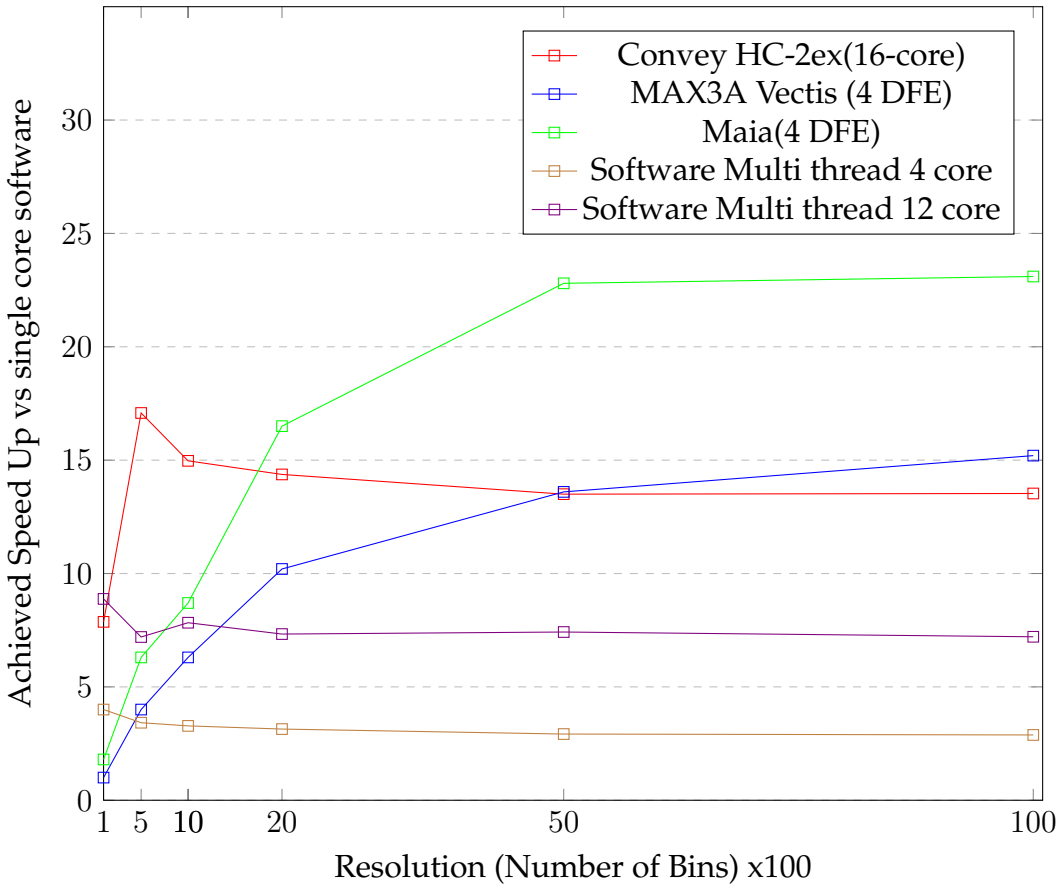

FIGURE 5.5: Achieved Speed for the proposed architectures on different platforms of MI Algorithm. Note that the scaling factor in x axis is x100 and represents the Resolution (Number of bins) of the MI algorithm.

internal DDR memory instead from PCIe which gives a significant advantage on this architecture in the design section.

5.3.2 Transfer Entropy

5.3.2.1 Multi-thread Software Implementation of TE

In order to explore the performance of the Transfer Entropy algorithm and yield an overall understanding of the performance of this algorithm just like for the MI algorithm we expand the reference software of TE in order to use multi-threading and evaluate this implementation in software. We created an application which calculates 12 TE cores in parallel on software. The yielded results provided on the table 5.12. As expected the multi-thread implementation on software can not achieve a speedup of the theoretical 4x and 12x respectively, compared to the reference software because a decisive factor to the overall execution time of the application is the memory bandwidth of the system which is shared to the implemented threads of the application as we already presented on the multi-thread implementation of MI algorithm. However, as we can observe from the table 5.12 for a small number of bins the execution time of the application is very small so the performance of the multi-thread implementation is

not very obvious. Note that from 1000 number of bins the execution time of the application is very high. That happened because the systems runs out of memory and starts to use the memory of the disk as well as in the 1200 number of bins. So typical for the TE calculation we can reach to the 800 number of bins for 12 TE calls in parallel because of the system limitations. The platform software in which the experiments took place was a two 6core Intel Xeon at 3.2 GHz with 50GB RAM.

TABLE 5.12: Performance of Single-core Transfer Entropy calculation compared with the performance of multi-threaded four-core and twelve core Transfer Entropy on software with 12 TE-calls.

Resolution	single core SW	4-core SW	SpeedUp	12-core SW	SpeedUp
100	0.6	0.15	**4**	0.065	**9.23**
200	3.36	0.87	**3.86**	0.34	**9.88**
500	32.12	11.61	**3.62**	4.32	**9.75**
800	153.6	47.88	**3.2**	17.95	**8.55**
1000	294‡	90.5‡	**3.24**	53.14 ‡	**5.53**‡
1200	out of mem	out of mem	out of mem	out of mem	out of mem

5.3.2.2 Multi-thread Histogram Function

As we described in section 5.3.1.2, we decided to implement the PDF vector function on the Host processor with threads as we already done with the MI algorithm. The tables 5.13 and 5.14 provides the reason for choosing the multi-thread implementation of the PDF function. As we can observe the execution time of the PDF vector in parallel with the use of OpenMP API is insignificant in relation with the single thread execution.

TABLE 5.13: Performance of PDF creation function on the Host side with single-thread execution for Transfer Entropy Algorithm.

Resolution	**Hist-func(2)**	**Hist-func(4)**	**Hist-func(8)**	**Hist-func(16)**
100	0.011	0.023	0.047	0.09
200	0.021	0.057	0.11	0.21
500	0.24	0.48	0.97	1.83
800	0.92	1.85	3.89	6.8
1000	1.78	3.56	7.15	13.1
1200	3.073	6.16	12.28	25.1

5.3.2.3 Performance of TE Implementation

In this section we present the performances of our implemented architectures. Firstly we present the multi-core architectures mapped in a single FPGA. The tables 5.15, 5.16 and 5.17 present the results of the architectures for one,two and four cores respectively. As we can observe from the tables the most time consuming part of our application is the transfer of the memory blocks to the shared memory. The obstacle of achieving better speedup of memory transferring in our implementation is obvious. In relation with the MI algorithm because the length of PDF vectors in order to calculate the TE

TABLE 5.14: Performance of PDF creation function on the Host side with Multi-thread execution for Transfer Entropy Algorithm.

Resolution	**Hist-func(2)**	**Hist-func(4)**	**Hist-func(8)**	**Hist-func(16)**
100	0.0055	0.0058	0.00589	0.0061
200	0.014	0.0148	0.0151	0.016
500	0.129	0.132	0.139	0.142
800	0.471	0.473	0.483	0.491
1000	0.91	0.916	0.926	0.928
1200	1.538	1.542	1.558	1.568

statistical value are more wide than the PDF vectors length of MI algorithm, the problem of transferring is come into focus. About the execution time of histogram function as well as the execution time of our hardware implementation the results are expected.

TABLE 5.15: Performance of 1-core TE Calculation in single FPGA.

Resolution	**Software**	Hist-func	Hardware	Memcpy	**Overall**	**SpeedUp**
100	0.05	0.00551	0.0034	0.0017	0.0106	4.71
200	0.28	0.014	0.026	0.013	0.053	5.28
500	3.51	0.12	0.41	0.2	0.73	4.80
800	12.8	0.46	1.69	0.83	2.98	4.29
1000	24.5	0.89	3.31	1.62	5.82	4.20
1200	40.9	1.53	5.7	2.81	10.04	4.07

TABLE 5.16: Performance of 2-core TE Calculation in single FPGA with Multi-threaded PDF function.

Resolution	**Software**	Hist-func	Hardware	Memcpy	**Overall**	**SpeedUp**
100	0.05	0.0055	0.0034	0.0035	0.0124	8.06
200	0.28	0.014	0.026	0.026	0.066	8.48
500	3.51	0.129	0.41	0.4	0.939	7.47
800	12.8	0.471	1.69	1.67	3.831	6.68
1000	24.5	0.91	3.31	3.25	7.47	6.55
1200	40.9	1.538	5.7	5.62	12.858	6.36

TABLE 5.17: Performance of 4-core TE Calculation in single FPGA with Multi-threaded PDF function.

Resolution	**Software**	Hist-func	Hardware	Memcpy	**Overall**	**SpeedUp**
100	0.05	0.0058	0.0034	0.0069	0.0161	12.42
200	0.28	0.0148	0.026	0.053	0.0938	11.94
500	3.51	0.132	0.41	0.81	1.352	10.38
800	12.8	0.473	1.69	3.33	5.493	9.32
1000	24.5	0.916	3.31	6.56	10.786	9.08
1200	40.9	1.542	5.7	11.24	18.482	8.85

TABLE 5.18: Performance of 2-core TE Calculation in 4 FPGA's with Multi-threaded PDF function.

Resolution	**Software**	Hist-func	Hardware	Memcpy	**Overall**	**SpeedUp**
100	0.05	0.00589	0.0035	0.013	0.02239	17.86
200	0.28	0.0151	0.0285	0.1	0.1436	15.59
500	3.51	0.139	0.43	1.63	2.199	12.76
800	12.8	0.483	1.73	6.67	8.883	11.52
1000	24.5	0.926	3.35	13.08	17.356	11.29
1200	40.9	1.558	5.74	20.9	28.198	11.60

As we can observe from the tables, the best mapped architecture in a single FPGA in terms of achieved speedup is the implementation with four TE cores. This architecture achieved the best speedup in a single FPGA. So in order to expand our architectures we attempted to mapped eight cores and sixteen cores in four FPGA's respectively. Our architectures implements two and four cores in each FPGA of the system. The tables 5.18 and 5.19 present the yielded results from the Multi-FPGA architectures.

TABLE 5.19: Performance of 4-core TE Calculation in 4 FPGA's with Multi-threaded PDF function.

Resolution	**Software**	Hist-func	Hardware	Memcpy	**Overall**	**SpeedUp**
100	0.05	0.0061	0.0038	0.027	0.0369	21.68
200	0.28	0.016	0.0292	0.21	0.2552	17.55
500	3.51	0.142	0.438	3.38	3.96	14.18
800	12.8	0.491	1.747	13.2	15.438	13.26
1000	24.5	0.928 ‡	3.39	25.2 ‡	29.518 ‡	13.28 ‡
1200	40.9	1.565 ‡	5.78	35.4 ‡	42.748 ‡	15.38 ‡

The resource summary of our multi-core architectures are presented in table 5.20. As we can conclude from the table 5.20, just like the MI implementations, under the average fraction of the available resources of the FPGA are utilized for the implementation of the architecture in each FPGA. Thus, any attempt to implement the proposed design is bound first by the available memory bandwidth similarly with the MI implementations.

TABLE 5.20: Resources summary of the Multi-core architectures of Transfer Entropy.

		occupied slices	slice registers	slice LUT	DSPs	BRAMs
1-core	Used	24241	76649	74514	32	142
	Percentage	**20%**	**8%**	**15%**	**3%**	**19%**
2-core	Used	29388	96021	92080	64	204
	Percentage	**24%**	**10%**	**19%**	**7%**	**28%**
4-core	Used	40841	138647	128951	128	328
	Percentage	**34%**	**14%**	**27%**	**14%**	**45%**

In order to yield a better overall understanding the figure 5.6 illustrates the performance of our architectures for TE algorithm in relation with the produced speed

up compared to the reference software. The y axis represents the achieved speedup of our implementations compared to the reference software and the x axis represents the range of PDF vectors. As we can conclude from the figure the best achieved speed up produced by the architecture with the 16 cores. Also the produced speed up starts from a peak value for each architecture and as the PDF vector be more wide the speed up of the architecture decreased. The problem of our architectures is located in the transferring of the PDF vectors.

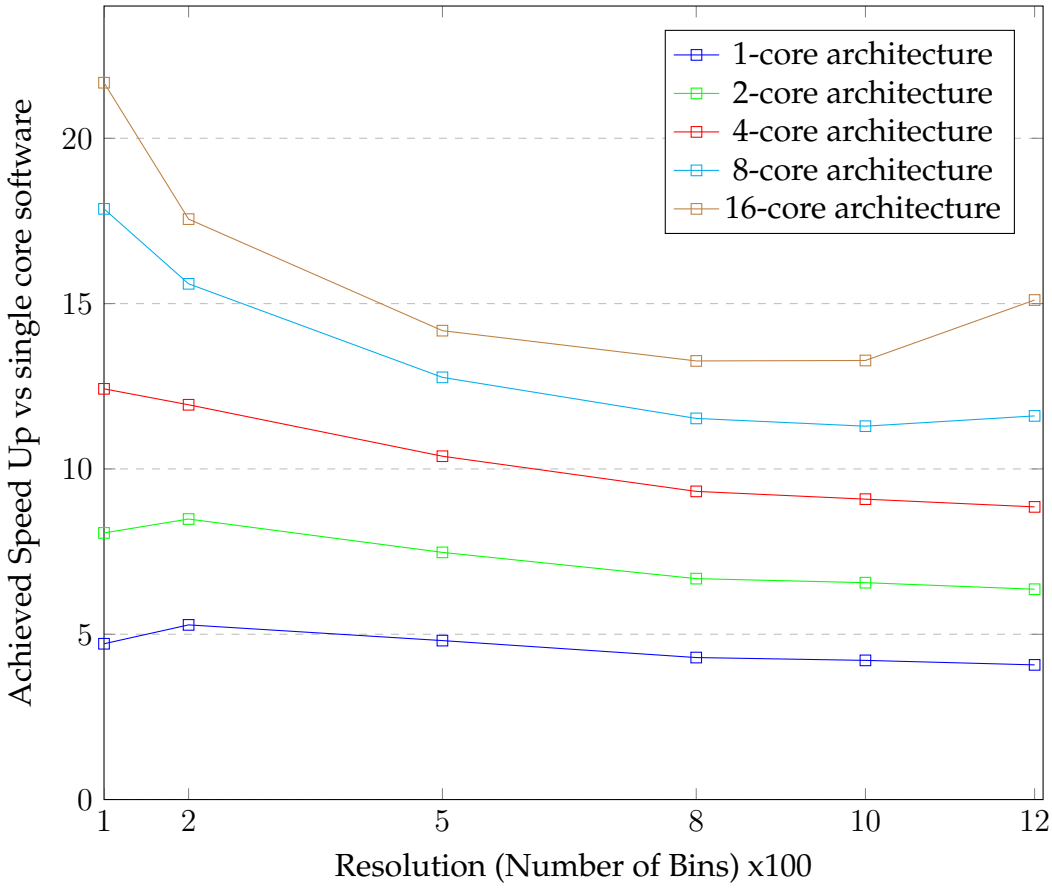

FIGURE 5.6: Achieved Speed for all the proposed architectures of TE Algorithm. Note that the scaling factor in x axis is x100 and represents the Resolution(Number of bins) of the TE algorithm.

As seen and from the figure 5.6, the performance of our architectures affected by the factor of the transferring time, and the achieved speed up decreased. Although we manage to keep the hardware time in low levels as well as we performed a multi threaded implementation in the PDF vector creation, the memory copy of the data to the shared memory is a deciding factor of the overall time of our application.

The figure 5.7 illustrates the distribution of the execution time in three main section of the application. As we described in the MI section, the y axis is the execution time in seconds and x axis represents the performance of our architectures with different lengths of PDF vectors. Note that in the figure 5.7 illustrated only one small range of bins and more specifically from 200 to 1200 number of bins which affects the overall time of our application. The first section of the application is the function which creates

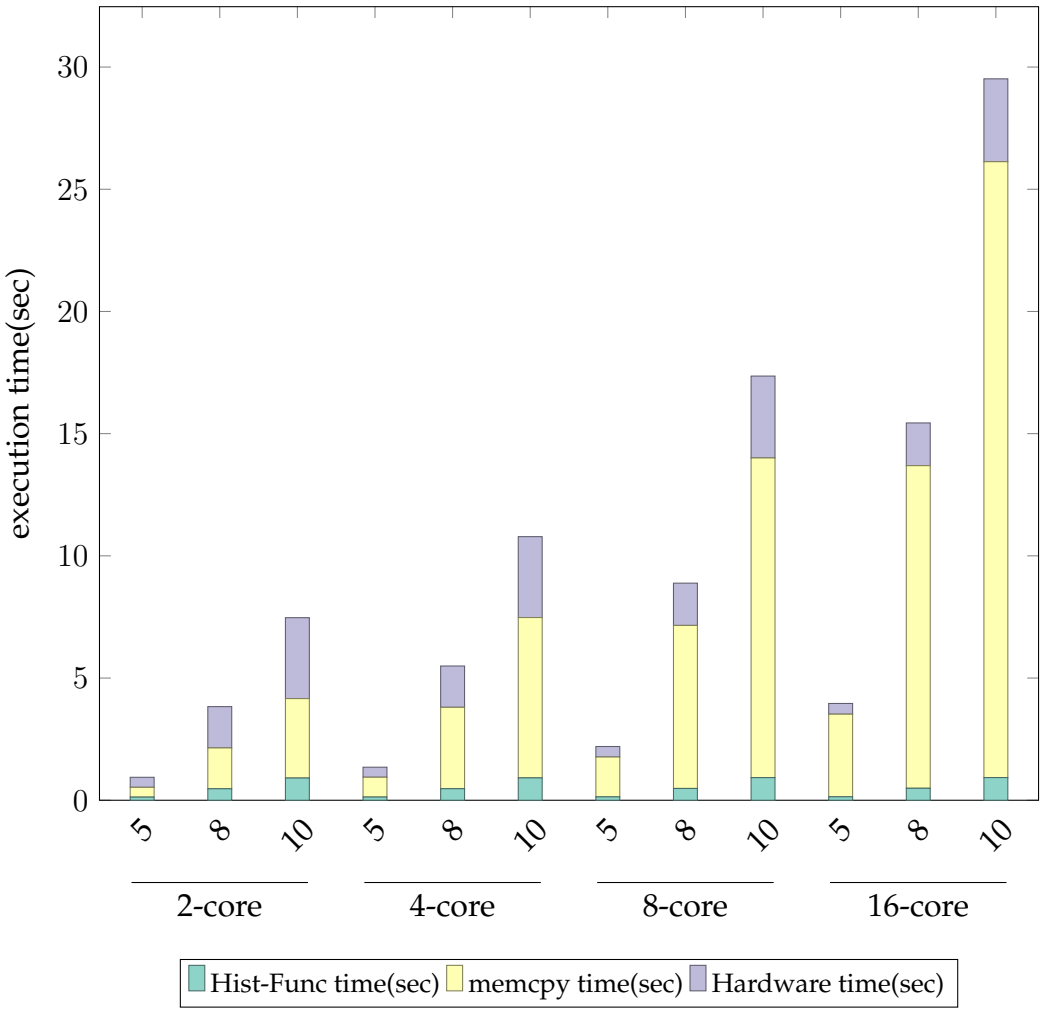

FIGURE 5.7: Time Consuming for all the proposed Architectures of TE Algorithm.The x axis represents the number of Bins of the PDF vector. Note that the scaling factor in x axis is x100

the PDF vectors. The second section consists of the transferring the data to the shared memory. The third section is the function call to the co-processor which means the execution time of our hardware implementation. The lengths of the PDF vectors for the TE algorithm are more wide than the lengths of the PDF vectors in MI algorithm, so the system in order to transfer the data to the shared memory requires more time, so for the TE implemented architectures the most time consuming part is more obvious.

In order to yield a better overall understanding the figure 5.8 presents the time consuming parts of one core implementation on a single FPGA of the Convey HC-2ex compared with the time consuming parts of the reference software. As we can observe the TE call (TE calculation on hardware) yields an important decrease of the execution time compared with the TE call on reference software. Also because we calculate one TE result in this experimental application the transferring time of the memory blocks to the shared memory is not the most time consuming part of our application. From this figure the most time consuming part seems to be the PDF estimation as well as in

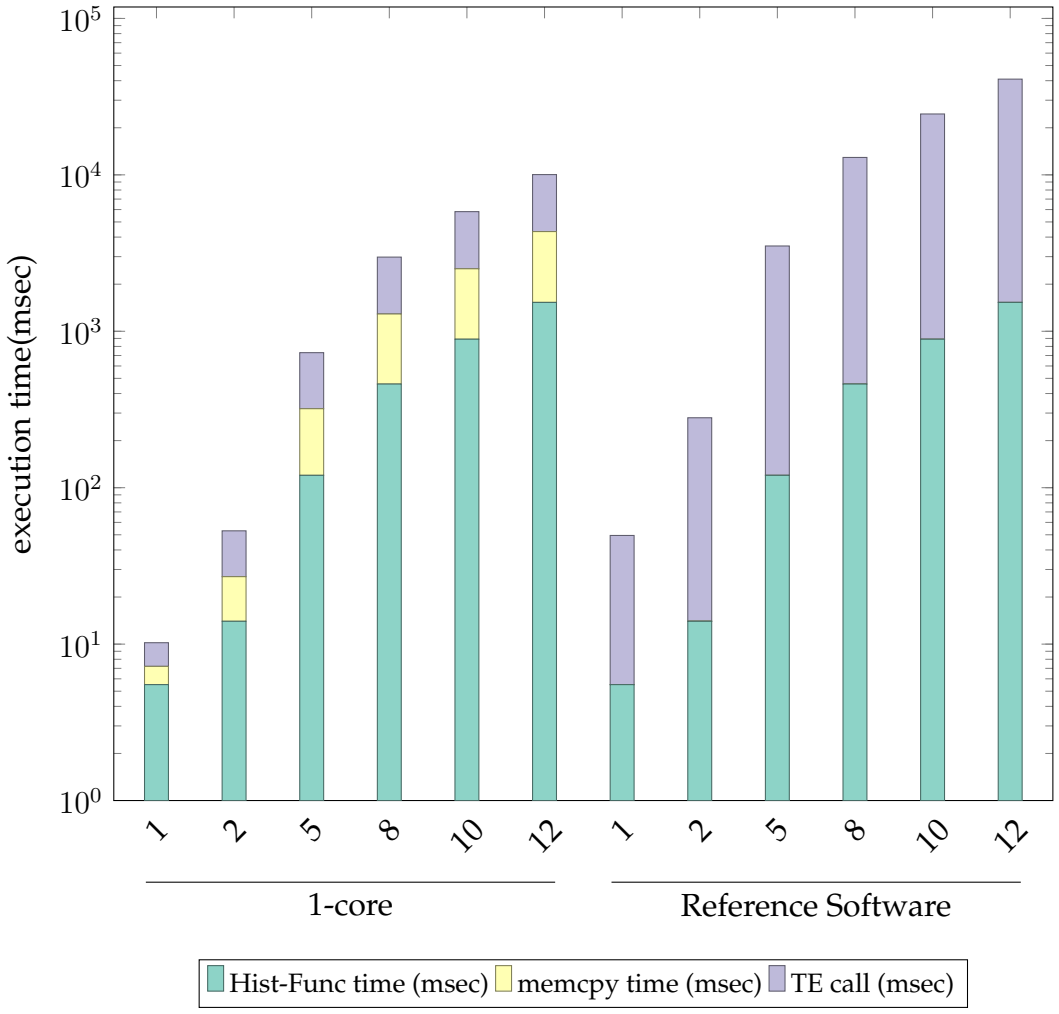

FIGURE 5.8: Time Consuming for the single core on single FPGA of TE Algorithm and the proposed reference software on logarithm scale. The x axis represents the number of Bins of the PDF vector.Note that the scaling factor in x axis is x100

the software implementation, although in the Multi-core and Multi-FPGA implementation because of the significant increase of the transferring time to the shared memory, the PDF estimation on those experiments is a drop in the bucket compared to the transferring time.

Just like the MI algorithm, the most time consuming part of our application is the transferring of the data. As described in section 5.3.1 in order to reduce the time of the data transferring a piece of different reworks implemented in the C-based application on the Host side and applied also in the TE implementation. A detailed functionality of these reworks described in section 5.3.1. The yielded results were the expected, after the apply of these reworks in the TE implementations. The figure 5.9 presents the achieved speedup of our implementation compared to other platforms as we already presented in the MI algorithm. The experiments of the other platforms yielded from the deliverables of the Qualimaster Project [16]. Note that the experimental values from the other platforms are the estimated results of the implementations. The platforms in

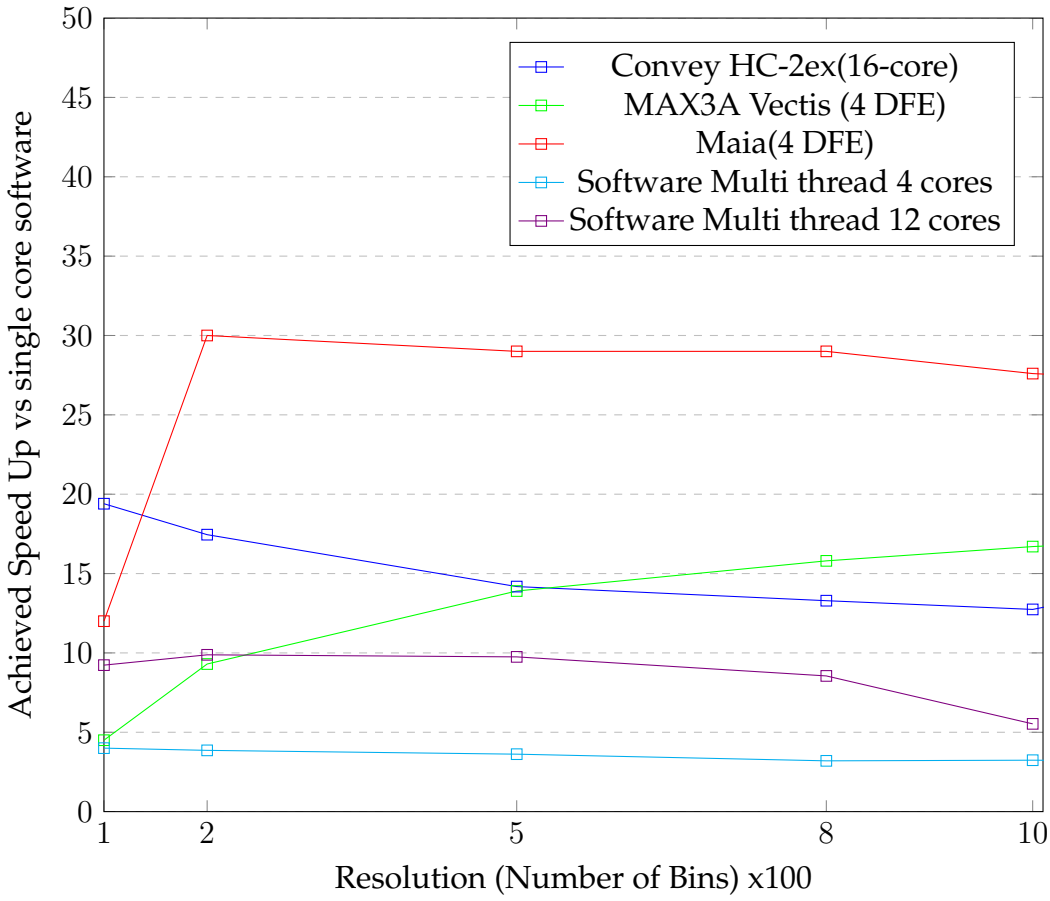

FIGURE 5.9: Achieved Speed for the proposed architectures on different platforms of TE algorithm. Note that the scaling factor in x axis is x100 and represents the Resolution (Number of bins) of the TE algorithm.

which the comparison took place were a MAX3A Vectis PCI Express card version of a Maxeler Dataflow Supercomputing System and a next generation Maxeler platform, the Maia. The detailed features of the platforms described in 5.3.1.

In conclusion, we can extract that our implementation compared with the Maxeler Vectis MAX3A are very close. Although we tried to keep the hardware and histogram creation execution times in low levels, the transferring of the data consumed a very wide percentage of the execution time of our application. This observation goes without saying as long as the two platform specifications are very similar. As we can observe from the figure, the Maxeler platform just like on the MI implementation, presents a higher overhead on TE call compared to the Convey HC-2ex which yields a higher speedup on smaller number of bins. As the number of bins getting more wide the transferring time to the shared memory of Convey HC-2ex getting the decisive factor of the overall application time. At the other side of the spectrum the yielded speedup of Maia platform is better than the implementations in MAX3A and Convey HC-2ex. Note that the implementation of TE in Maia as well as the implementation of MI algorithm deployed the internal internal memory (LMEM) of the platform so the achieved speed up was expected. Also the Maia platform has a key strength compared to the specifications of our platform Convey HC-2ex and MAX3A.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The evolution of computer systems has been driven by the exponential increase in logic density. Performance has increased exponentially as clock rates increased, and soaring transistor counts are utilized in a wide variety of architectural innovations to increase performance. However, in recent years single-core performance is falling out of flavor because of the increasing of system complexity and of clock frequency. As a result companies have turned to heterogeneous computing architectures that employ semiconductor gates in more efficient configurations. Specifically, computing elements like general-purpose graphics processing units (GPGPUs) and field programmable gate arrays (FPGAs) are used to perform application-specific functions directly in hardware. The resulting combination of a custom architecture based on reconfigurable logic is a more efficient alternative compared with conventional CPU.

In this thesis we mapped two computationally intensive algorithms on Convey HC-2ex, a hybrid supercomputer. The implementation process entailed the gradual design of the architecture following a top-down approach. The first steps in order to design and map the hardware architectures in the system were the detecting of the individual key calculations which led us to compose them and produce the final result of the two algorithms.

The two algorithms designed, mapped and evaluated on the Convey HC-2ex, a hybrid supercomputer in order to fully exploit its capabilities for the parallelization of the operations required for the estimation of the statistical value of MI and TE algorithms. The combination of a high-bandwidth memory interface with an architecture featuring multiple levels of computational parallelism proved to perform better than state-of-the-art software running on high-end CPUs. Even thought the design was not tailored for extreme performance efficiency, it still managed to process large datasets 13 times faster than its current software counterparts for MI and 15 times faster for TE results.

The conclusion of this work would have to be that implementing, such as computationally intensive algorithms consisting of a large amount of repetitive simple calculations, highly benefit from the development of custom application-specific pipelines that exploit the capacity of reconfigurable devices for cheap and efficient parallel implementations with custom-width arithmetic addressing exactly the application's requirements. However, the difficulty in using Convey platform lies in the fact that the studied algorithm needs to be reconsidered from the beginning based on the properties of the tool and not based on related works, as well as that the Convey platform

presents unpredictable behavior in certain simple functionalities that cannot be known in advance.

As a result this thesis is a small contribution to the fields of reconfigurable computing that is to be improved upon, as hybrid computing comes to the front as the demands of the applications getting overburdened.

6.2 Lessons Learned

This thesis focused on two directions. The first direction was the implementation of computational intensive algorithms on reconfigurable hardware and the second was the comparison between the two multi-FPGA platforms (MAX3A from Maxeler Technologies and the Convey HC-2ex). The basic difference between the two platforms is that Maxeler platform initiates streams from memory to the DFEs. Data transfer and processing is done simultaneously when PCIe streams are used (Multiplexing I/O). On the other hand for the Convey platform the data have to be copied from the host memory to the FPGA memory, which takes a significant amount of the total execution time. So even if the Convey system offers greater memory bandwidth (80GB/s) vs. the Maxeler PCIe (2GB/s per FPGA), the performance is very similar. In other words, the Convey system can offer memory bandwidth of 80GB/s to the 4 FPGAs with the **important assumption** that the data first have to be copied from the *hosting* shared memory subsystem to the *coprocessor* memory subsystem via PCIe which is very consuming and **then** the coprocessor memory subsystem transfers the data with 80GB/s memory bandwidth to the FPGAs.

Concerning the parts of the processing between the two multi-FPGA platforms, MAX3A Vectis as well as Convey HC-2ex can solve very easily and efficiency algorithms which are in the class of systolic arrays problems, despite the fact that Convey HC-2ex is even better in processing compared to the Maxeler MAX3A Vectis and this observation goes without saying from the fact that in small range of input data MAX3A Vectis seems to have a significant overhead compared to Convey HC-2ex.

We can only be hopeful for the future of hardware design, as more interesting advances in high-performance computing are underway. For example with the new cooperation between Intel and Altera in the near future we will enjoy new hybrid architectures in which the reconfigurable co-processor will be directly attached on the host processor. With this way the memory copy between the memory subsystems via PCIe will be distinguished and this progress with many other new features in heterogeneous architectures will lead us to a blissful yearning for further experimentation around novel ideas and designs.

6.3 Future Work

The time for completing a diploma thesis is always too short for implementing all ideas that arise during the work. At the end, three of them are outlined as outlook for future work.

Although in our architectures for the MI and TE we managed to keep the hardware execution time as well as the PDF vector creation in low levels, we yielded that the transferring time was the most time consuming part of our application. We tried to

decrease the transferring time of our application using the concept of pipelining in the data transferring and we implemented a piece of other ideas as presented in this thesis. The idea of implementing a pipelining between the memory blocks was a very good idea if the processing time was close to the transferring time. As we presented the processing time was a drop in the bucket compared to the transferring time. So as future work include an effective way to reduce the transferring time of the data to the shared memory of the system.

Furthermore the PDF vector creation in our implementations for MI as well as for TE are based on the matrix representation. More specifically histograms are commonly represented by an array or a matrix of counters. Although simple and intuitive, this representation is quite inefficient in terms of memory, for large resolution, especially when the PDFs are skewed (i.e., sparse) and not uniform. So as future work include the sparse representation of the PDFs using *Hashmap* as described in [15].

Finally, it would be useful to study other algorithms especially in the fields of finance, geophysics and data mining to further evaluate our acquired knowledge and achieve higher speedups.

References

[1] Kraskov A., Stogbauer H., Andrzejak R.G., and Grassberger P. "Hierarchical clustering using mutual information". In: *EPL (Europhysics Letters)* (2005).

[2] Carlos R. Castro-Pareja and Raj Shekhar. "Hardware acceleration of mutual information based 3d image registration." In: *Imaging Science and Technology.* (2005), pp. 105–113.

[3] Convey Computers. *Convey Personality Development Kit Reference Manual*. Convey Computers, April 2012. URL: http://www.conveysupport.com/alldocs/ConveyPDKReferenceManual.pdf.

[4] Convey Computers. *Convey Programmers Guide*. Convey Computers Corporation, June 2012. URL: http://www.conveysupport.com/alldocs/ConveyProgrammersGuide.pdf.

[5] Convey Computers. *Convey Reference Manual*. Convey Computers Corporation, May 2012. URL: http://www.conveysupport.com/alldocs/ConveyProgrammersGuide.pdf.

[6] FloPoCo. *FloPoCo,Circuits computing just right*. URL: http://flopoco.gforge.inria.fr/.

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

[8] *Histograms*. URL: https://en.wikipedia.org/wiki/Histogram.

[9] Shengjia Shao, Ce Guo, Wayne Luk and Stephen Weston. "Accelerating transfer entropy computation." In: *Field-Programmable Technology (FPT), 2014 International Conference on, IEEE.* (2014), pp. 60–67.

[10] Tebmann M., Eisenacher C., Enders F., Stamminger M., and Hastreiter P. "Gpu accelerated normalized mutual information and b-spline transformation." In: *VCBM.* (2008), pp. 117–124.

[11] Maxeler. *Maxeler MPC-X Series*. URL: https://www.maxeler.com/products/mpc-xseries/.

[12] Thomas M.Cover and Joy A. Thomas. *Elements of Information Theory*. John Wiley and sons, 2012.

[13] *Mutual Information*. URL: https://en.wikipedia.org/wiki/Mutual_information.

[14] Sofia Maria Nikolakaki. *Real-time Stream Data Processing with FPGA-Based Supercomputer*. July 2015. URL: http://purl.tuc.gr/dl/dias/14890E6A-5050-49F7-A104-1FF844601572.

[15] Pavlakis Nikolaos. *Scaling out streaming time series analytics on Storm*. March 2017. URL: http://purl.tuc.gr/dl/dias/802769EF-66BD-411D-9E6E-BA00A61F2B6F.

[16] Qualimaster Project. URL: http://qualimaster.eu/.

[17] Qualimaster Project. *Qualimaster deliverables 3.1*. URL: http://qualimaster.eu/wp-content/uploads/2014/01/QualiMaster-WP3-D3-1.pdf.

[18] Qualimaster Project. *Qualimaster deliverables 3.2*. URL: http://qualimaster.eu/wp-content/uploads/2015/09/QualiMaster-WP3-D3-2.pdf.

[19] Qualimaster Project. *Qualimaster deliverables 3.3*. URL: http://qualimaster.eu/wp-content/uploads/2015/01/QualiMaster_WP3_D3-3.pdf.

[20] Shams R. and Barnes N. "Speeding up mutual information computation using nvidia cuda hardware." In: *Digital Image Computing Techniques and Applications, 9th Biennial Conference of the Australian Pattern Recognition Society on, IEEE.* (2007), pp. 555–560.

[21] Shams R., Sadeghi P., Kennedy R., and Hartley R. "Parallel computation of mutual information on the gpu with application to real-time registration of 3d medical images." In: *Computer methods and programs in biomedicine.* (2010), pp. 133–146.

[22] T. Schreiber. "Measuring information transfer". In: *Phys.Rev.Lett.,vol. 85* (2000), pp. 461–464.

[23] C.E Shannon. "A Mathematical Theory of Communication". In: *ACM SIGMOBILE Mobile Computing and Communications Review* (2001), pp. 3–55.

[24] Shengjia Shao. *Accelerating Transfer Entropy Computation*. 2014. URL: http://www.doc.ic.ac.uk/teaching/distinguished-projects/2014/s.shao.pdf.

[25] N. Alachiotis, A. Stamatakis. "Efficient Floating-Point Logarithm Unit for FPGAs." In: *RAW workshop, held in conjunction with IPDPS 2010, Atlanta, Georgia.* (2010).

[26] N. Alachiotis, A. Stamatakis. "FPGA Optimizations for a Pipelined Floating-Point Exponential Unit". In: *7th International Symposium on Applied Reconfigurable Computing* (2011).

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

[28] *Transfer Entropy*. URL: https://en.wikipedia.org/wiki/Transfer_entropy.

[29] Xilinx. *LogiCORE IP Floating-Point Operator v5.0*. Xilinx Corporation, March 2011. URL: https://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf.

[30] Xilinx. *Virtex-6 Family Overview*. Xilinx Corporation, August 2015. URL: https://www.xilinx.com/support/documentation/data_sheets/ds150.pdf.

[31] Xilinx. *Xilinx ISE design suite*. Xilinx Corporation. URL: http://www.xilinx.com/products/%20design-tools/ise-design-suite/.

[32] Lin Y. and Medioni G. "Mutual information computation and maximization using gpu." In: *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on, IEEE.* (2008), pp. 1–6.