Technical University of Crete, Greece

School of Electronic and Computer Engineering

Real-time Stream Data Processing with FPGA-Based SuperComputer

Sofia Maria Nikolakaki

Chania, July 2015

Πολυτεχνείο Κρήτης

Σχολή Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών

Επεξεργασία Δεδομένων Πραγματικού Χρόνου με Υπερυπολογιστή Βασισμένο σε Αναδιατασσόμενη Λογική

Σοφία Μαρία Νικολακάκη

Χανιά, Ιούλιος 2015

Abstract

It is a foregone conclusion that contemporary applications are bounded by massive computational demands. The semiconductor industry has announced that physical constraints restrict the community from surpassing the currently upper frequency limit of modern processors, thus leading to the emergence of multi-core platforms. This thesis explores the recently emergent paradigm of the Maxeler multi-FPGA platform for dataflow computing to efficiently map computationally intensive algorithms on modern hardware. We tackle two challenging problems within this framework, the first being classification by focusing on the kernel computation of the broadly used Support Vector Machines (SVM) classifier, and the second being time-series analysis by focusing on the calculation of the Mutual Information (MI) value between two time-series. Prior art on modern hardware has indicated the parallelism opportunities offered by the SVM method, but mainly for low-dimensional datasets, while no work has contemplated the performance of the algorithm on dataflow processors. Moreover, the problem of MI computation between two time-series on special-purpose platforms has been addressed by the research community for low-precision arithmetic applications, and again the performance of the specific method has not been evaluated on the emerging dataflow platforms. This is the first work to extensively study the pros and cons of using the Maxeler platform, by identifying the most essential dataflow elements and describing how they can be efficiently utilized. Thus, it can be employed as an independent point of reference for similar future endeavors. In terms of results, while the SVM kernel computation reached the same performance as the reference software for high-dimensional data, the know-how acquired during this process was leveraged towards the design of the MI FPGA-based architecture that yielded 9.4x speedup using two parallel cores and 32-precision arithmetic.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor, Prof. Apostolos Dollas, for his trust, enthusiasm, continuous guidance from the very beginning and for our fruitful discussions concerning my future steps. I would also like to thank Prof. Minos Garofalakis and Prof. Ioannis Papaefstathiou for accepting to be in my committee.

I am also deeply grateful to Prof. Stavros Christodoulakis for inspiring me throughout the first years of my studies and for providing me the right incentives to excel.

Furthermore, I would like to thank Pavlos Malakonakis for his assistance whenever needed and for our cooperation. He did not only stand by me as a good friend, but he also offered me fine suggestions regarding my thesis.

My parents, Souzana and Ioannis, as well as my beloved younger brother, Emmanouil for always being my role models in matters of morals and principles, for loving me and for supporting my decisions.

I am thankful to my dear Ioannis Demertzis for encouraging and supporting me unconditionally, in the last few years.

This work was partially supported by EU FP7 project QualiMaster (Contract No. 619525).

Contents

List of Figures

Chapter 1

Introduction

It is a foregone conclusion that contemporary applications are bounded by massive computational demands. However, according to the semiconductor industry the introduced physical constraints restrict the community from surpassing the currently upper frequency limit of modern processors. Yet, this observation led to the emergence of multi-core platforms. In this thesis we explore a multi-FPGA platform to tackle computationally intensive algorithms. More specifically, we study the paradigm of Dataflow supercomputing implemented on the Maxeler systems. Unlike Control Computing, Dataflow Computing focuses on data processing instead of scheduling by configuring lower level elements, such as gates and wires, and thereby it constitutes a setting appropriate for time and energy efficient implementations. The main challenge in Dataflow computing is to identify and modify trending algorithms based on the properties of the Dataflow architecture platform that is being used to yield efficiency. Geophysics, data mining and finance are a few of the scientific fields that have effectively used Dataflow architectures. The potential of Dataflow computing on the Maxeler Supercomputing platform led to the main focus of this thesis. In particular we derived a methodology to efficiently map on hardware dataflow algorithmic parts that comprise thousands of simple computational operations with dependent values in between.

The scientific area of Machine Learning has introduced certain imperative operations, such as semantic text classification and object recognition. A common characteristic that the majority of these algorithms shares is the immense amount of data and the multiple opportunities for data processing parallelism. The aforementioned properties along with the lack of relevant research on the specific topic led to our decision to create an efficient training Support Vector Machines dataflow architecture. In general, the SVM Training phase involves quadratic programming and general-purpose solvers can only handle a relatively small number of elements. Thus, the scientific community turned to special-purpose solvers to address large-scale instances. Given that Dataflow architectures are efficiently utilized when they dedicate several cores for parallel processing of simple computations, we aimed at accelerating the kernel computation in the SVM Training phase which mainly requires matrix-vector operations. In addition kernel functions constitute a significant percentage of the overall execution run-time, and are promising to yield acceleration based on previous findings. However, now we are evaluating a completely different setting, and thereby challenges such as the streaming nature of the data processing need to be tackled. Our main purpose is to investigate, if similar to graphics processors and FPGAs, dataflow architectures can become efficient for kernel computations.

The statistical analysis of financial time series has resulted ins 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 of critical importance, such as in risk analysis. Banks and financial enterprises apply correlation functions on streams to perform such real-time analysis. In this work we aim at accelerating the computation of the Mutual Information statistical measure between two time-series to identify dependence. We considered this computation suitable for a data flow architecture, as similarly to the kernel computation, it involves accumulations and multiplications, and is also a metric that has produced impressive results in real-time problems such as image processing. Yet, we also used this second computation function to evaluate the performance and generic nature of the methodology followed to devise data flow algorithms that involve a big amount of simple operations and also dependent values during computation.

1.1 Thesis Contribution

Dataflow architectures have been proven redemptive for several scientific fields with massive amounts of data to be processed and high computational requirements. Experimental results showed that dataflow computing can reach orders of magnitude better performance compared to other parallel implementations, with significant power savings. Yet, this type of computing lacks detailed implementation descriptions and experimental evaluations in order for interested scientists to exploit its benefits, thus inducing a significant learning curve. We utilized the MAX3A Vectis card, provided by Maxeler technologies to present the basic elements of a dataflow technology and how they can be efficiently utilized. To the best of our knowledge, this is the first work that addresses the whole problem of designing a dataflow architecture from the beginning until the end. We considered two dataflow architectures, the first being the kernel computation component of the Support Vector Machines method and the second being the calculation of the statistic measure Mutual Information. The problem of accelerating the training phase of Support Vector Machines has been tackled by several research groups. In the FPGA community the state-of-the-art approaches produce approximate results with the use of low-precision arithmetic and evaluate their performance in small dimensional datasets. The GPU community achieved better results, and more specifically achieved 32x speedup implementations compared to the LIBSVM reference software. Still once again the final accuracy was not optimal and datasets with small feature space were evaluated. Thus, given that the Maxeler supercomputer system offers a large off-chip memory and high bandwidth we decided to create a system that produces the exact same solution as the widely accepted reference software LIBSVM and evaluates high-dimensional datasets. Our research was not limited to the implementation of a single dataflow system due to the fact that we wanted to evaluate the acquired knowledge on another case. Hence, we considered the problem of calculating the Mutual Information statistic value between time-series and achieved 5x speedup compared to the respective software implementation.

1.2 Thesis Outline

Chapter 2 initially introduces us to the Dataflow Programming concept and then emphasizes on the Maxeler System platform. The goal of these descriptions is to provide a deep comprehension regarding the outcomes of modifying an algorithm based on a dataflow architecture, and specifically based on the properties of the Maxeler platform. Furthermore, we present the theoretical background of the algorithms studied in this thesis, i.e. the Support Vector Machines (SVM) classification method and the Mutual

Information (MI) computation. This overview indicates and analyzes important algorithmic parts that are also considered during the hardware implementation. Chapter 3 briefly presents related works that have been proposed for the acceleration of the SVM and MI computations with the use of hardware-based platforms or multi-core processors. Chapter 4 consists of two parts. The first is algorithmic analysis and profiling from a hardware designer's perspective. The second part presents how the software source codes of the two algorithms were modified in order to become integrated with the Maxeler data flow system, as well as the methodology followed to efficiently map these algorithms to the FPGA platform. In Chapter 5 we evaluate the performance of our hardware-based systems in terms of space and time with the use of differently sized datasets. Finally, in Chapter 6 a conclusion about the presented work is provided, followed by future work directions that are worth considering.

Chapter 2

Background

2.1 Maxeler

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 London-based company Maxeler which from the very beginning focused on what we call data flow computing, a term introduced in the early 1970s. Briefly, the Maxeler platforms are data flow environments, targeted for performance increase in terms of acceleration and energy efficiency. In order to do so, they provide a high-level language which makes them more convenient and easy to use by a wide audience. In addition, even though substantial work has been conducted regarding high-level languages with the most representative works being [1] and [2], none is designed specifically for streams, whereas Maxeler was developed for this purpose.

2.1.1 Data flow Programming Model

The **Data flow Programming Model** is actually a graph execution model and represents an alternative to the conventional control flow programming model. A control flow model depends on task completion. In general, tasks are executed in series if connected based on precedence, or otherwise they are executed in parallel. For example, if the program is sequential task 2 will begin only after task 1 has completed, and only then will the data be allowed to move from one task to the other. More specifically, in software applications the source code is translated to a sequence of instructions for a specific processor which are loaded to the memory. These instructions guide the processor and sometimes allow reading and writing data from and to the memory respectively to complete certain commands. It is well known that contemporary processors do not simply function based on the traditional method we previously described, but have integrated several optimization techniques, including forwarding and prediction logic, as well as caching in order to speed up the process. Nevertheless, regardless of the possible improvements the control flow programming model remains sequential, and thereby its latency depends on the time required for a CPU clock cycle, as well as the number of memory accesses. To conclude, control flow programming implies designating the order in which the individual statements, function calls or instructions are executed or assessed.

On the other hand, data flow computing is a simple, yet powerful model mainly used for parallel computing. In particular, it deviates significantly from typical computer architectures, due to the fact that the order of execution is controlled only by the flow of the data. Unlike control flow programming where instructions are executed sequentially according to a program counter, data flow instruction are performed immediately when the respective resources necessary for the execution are available. More specifically, in a data flow environment any two enabled instructions can be executed in any order, or simultaneously due to the fact that the they do not affect one another. Hence, data flow computing is an efficient tool for parallel programming, as its performance is restricted primarily by data dependencies.

In addition, when thinking of data flow programming it is useful to consider data as streams rather than batches. Usually, we consider a stream of data to be a sequence of instances which are being processed one at a time rather than several together, and which require independent processing. Also, it is often desirable to process streaming data in near-real time. On the other hand batches are a set of data which are processed as a unit, and thereby all the data that belong to the batch are visible to the processing system. That said, unlike control flow modeling, dataflow programming models comprise multiple components that can process input data at the same time. Also, the smallest unit of dataflow models is a component, whereas the smallest unit of control flow models is the task. An analogy that could illustrate the difference between control and data flow is the case of an artisan and a manufacturing enterprise. In the former case the artisan sequentially completes one task after another, whereas in the latter case each worker undertakes a small task so that all workers can work in parallel on different parts of an object.

This section presents an overview of data flow modeling for the purposes of this thesis. The reader is referred to, among many others, the following texts [3], [4], [5], [6].

2.1.1.1 Basic Principles

A data flow model does not contain typical notions such as *variables* and *memory updates*. Instead, it functions in terms of objects, which can either be data structures or scalars. These objects are inserted and processed by an actor whose functionality is similar to the one of an instruction. In particular, an actor may correspond to a single instruction or a sequence of instructions. The computational complexity and size of an actor is designed by the programmer, rather than the system. An actor fires when all the input its expecting is available to him. Then, the produced result object is outputted and passed to subsequent actors. Many actors may be ready to be triggered concurrently, given that the necessary operands are available. Thus, it is useful to consider them as asynchronous concurrent computation events. Note that some data flow architectures offer the functionality of variables and memory updates to be more convenient, applicable and user-friendly. Partially, the Maxeler Dataflow platform satisfies these operations. Also, the replacement of instructions with actors leads to elimination of instruction decode logic, branch prediction and dynamic scheduling, and thereby the resources on the chip are available for computation purposes only. Now that we have presented the principle notions of data flow models, we can further discuss about how such models are represented.

2.1.1.2 Dataflow Graphs

The term data flow architecture appeared in the 1970s [7], [8], [9] along with the term **Dataflow Graph**, since together they led to the exploitation and illustration of parallelism. Data flow program graphs were initially utilized as a description of a machine language. The definition of a Dataflow Graph is as follows:

- A directed graph that indicates data dependencies between a number of functions.

- G = (V,E):

  - V: Nodes which depict instructions with input/output data ports.
  - E: Arcs which represent connections of the output ports with the input ports.

- Semantics:

  - Fire when input data are available.
  - Consume data from input port and produce data to output port.
  - Many nodes may be ready to fire simultaneously.

In general, a data flow graph is a directed graph, *G(V,E)* where nodes V (actors) represent primitive instructions such as arithmetic or comparison operations and arcs E show data dependencies among the instructions. It is useful to envision data flowing along the arcs, as tokens. Thus, the arcs operate as unbounded first-in, first-out (FIFO) queues. Those directed towards a node are the input arcs of the node, whereas those leaving from a node are respectively the output arcs.

Now that we have introduced the notion of data flow graphs we can present the two main structures of data flow architectures that have prevailed. The first concerns static architectures. More specifically, in this type of architecture each arch of the graph can only carry a single result value, called a token, from the source instruction and there can be only one instance of each actor running at any time. The other commonly used architecture, is the dynamic data flow scheme. This scheme introduces tags which are associated with tokens in order to be able to distinguish those who have derived from the same actor but in different firings. Thus, arcs can concurrently contain numerous tokens and data parallelism is favored. Note the convenience and effectiveness of describing parallel computation with data flow graphs. At this point we need to stress out the difficulty of designing data flow hardware architectures mainly because the data flow model is theoretical, which renders it impossible to be carried out in the exact same way in the real world. To begin with, the arcs of the graph cannot be FIFO queues of unbounded capacity, since such thing does not exist. In addition, it also assumes that any number of instructions can be performed concurrently, which actually depends on the available number of processing units. Hence, no hardware data flow models can replicate precisely the theoretical corresponding model.

2.1.1.3 Dataflow Languages

The idea of data flow has been used in a wide range of applications, such as allowing a massive number of computations on data or providing visual languages to facilitate the programming process. In this section we focus on the latter type of application which is data flow languages. Inevitably, the development of data flow hardware would lead to the imperative need of being able to program such machines. Hence, the design of appropriate programming language was necessary in order to exploit the possibilities offered by data flow computing. This triggered the search not only for the specific language, but also for a suitable compiler to produce it, which led to the emerge of the **data flow languages**.

Due to the fact that data flow languages are based on data flow graphs, it seems convenient to express them graphically. However, in contrast to this assumption the majority of languages designed to operate on data flow platforms are not graphical and this is mainly due to two reasons. The first is that when a low-level and complex description of a data flow architecture is required, the process of graphically representing it becomes unnecessarily time-consuming and complicated. Instead, with the use of textual languages the same process becomes much easier and faster [10]. The second reason for which text-based data flow languages have endured lies in the fact that the appropriate hardware for displaying graphics was not available until relatively recently. At this point, we are ready to present what distinguishes Data flow languages from other programming languages. This is not an easy task, as it is common for programming languages to overlap. For example, it is not necessary that a data flow language will only be applied on a data flow environment and vice versa, another type of programming language might be proved to be quite effective in a data flow setting [11], [12]. Conclusively, the boundary that distinguishes data flow languages from others is transparent. Yet, there are certain properties that all data flow languages must exhibit, which are listed in detail in [13]. Briefly, some of these features are the following:

- Insensitive to side effects
- Single variable assignment without possibility of reassigning
- Locality of effects
- Scheduling determined based on data dependencies

- Independence from historical processes

The fact that scheduling is defined based on data dependencies implies the prerequisite that the value of variables remains stable between their definition and their use. Thereby, the possibility of reassignment is eliminated. Also, the single variable assignment changes the way we perceive and handle variables, as now it is more convenient to consider them as values. Furthermore, insensitivity to side effects and historical procedures is crucial in order to guarantee that scheduling is only determined by data dependencies. Data flow languages achieve this feature by not allowing global variables, or even parameter modifications within functions. However, there are cases, such as when we are processing an array, when it is imperative to modify it. Then, each new version of the array is actually considered a new input instance with the respective positions modified. Taking everything into consideration, we might reach the conclusion that data flow languages are basically invariably functional. Some examples of data flow languages are LabVIEW [14], VHDL [15], LUSTRE [16].

2.1.2 Dataflow Engines (DFEs)

There are two broad types of contemporary computer architectures, the first being general-purpose computing, and the second being problem-specific computing. The first class of architectures was originally delivered by John von Neuman and has remained the most prevalent. Yet nowadays, the appearance of challenging computational problems, as well as the abundance of available data has urged researchers to focus on the second class of architectures. The main benefit of special-purpose architectures is that they can be optimally configured based on the requirements and properties of the respective problem, leading to much faster performances. Thus, problems that were previously unresolved due to lack of eligible solutions have been addressed and acceleration of a variety of applications has been achieved. The first special-purpose architectures originated from mapping custom solutions into hardware, by creating application specific integrated circuits (ASICs).

Even though this approach is the most effective one, it is also not amenable to changes and completely hardware specific. Therefore, it is quite expensive to create and not convenient for use. This changed when Field Programmable Gate Arrays were introduced, which can be considered as programmable ASICs. Their main drawback compared to

ASICs is that they cannot reach the same performance in terms of speed and energy efficiency. However, they are still a lot more user friendly and flexible due to the fact that they are programmable, and achieve highly parallel and accelerated implementations. FPGAs are also called reconfigurable computing architectures, as they combine software specific and hardware specific approaches, thus providing the programmer with the liberty to reprogram hardware architectures. Note that although FPGAs can reach extremely high performances by achieving massive parallelism, they also have a major drawback compared to traditional computer architectures which is a slow clock. Hence, either of the two types of computer architectures should be used based on the properties of the application, and usually the most efficient way to benefit is by combining them. Moreover, FPGAs are a fundamental component of the Maxeler technologies, and thereby we will provide a more detailed definition and description. In the previous section we presented a brief description of the functionality of FPGAs. To be more precise, FPGAs are reconfigurable logic chips that can be reprogrammed in seconds to implement customized designs. However, the size of an FPGA, or in other words the amount of resources it offers (BRAM, flip-flops, LUTs, DSPs) determine the designs that can be supported. Moreover, their main advantages compared to other types of conventional architectures (GPUs, multi-core CPU, ASICs) are basically the following:

- Flexibility, which is much greater than the one offered by ASICs.
- Potential to optimize low-level elements
- Power efficiency compared to multi-core CPU and GPUs.

We briefly presented the FPGAs because they play a crucial part in the Maxeler architecture. In particular, they are the key component of the Dataflow Engines which we are about to discuss.

To begin with, each Data flow engine (DFE) is a reconfigurable chip with lots of memory. Even though DFEs can be implemented with the use of any suitable subtrate, Maxeler uses as building blocks FPGAs due to their popularity and flexibility. We stress out from the beginning of the DFE description that Data flow engines do not perform calculations in time, but in space since they are built with the use FPGAs. More specifically, data are streamed on the chip of the DFE from some memory. The interesting part is that the movement of the data is specific and driven from one functional unit to the other without requiring the interference of an external off-chip memory, until the completion of the execution. Recall that in control flow platforms operations executed on the same functional unit should be performed at different time instances. However, in data flow environments, such as Maxeler, space is used to allow concurrent executions. Furthermore, note that the DFE itself is a computational unit whose resources are only used for computation purposes. Note that in order to reach maximum performance in terms of speed and efficiency, both hardware and software are needed. Thus, DFEs are not independent, but they are integrated with the conventional CPU for balanced systems.

Regarding its internal architecture, a DFE implements one or more Kernels and a Manager. A *Kernel* simply contains the computation we intend to perform, whereas the *Manager* controls the data movement within the DFE. Then, the MaxCompiler utilizes the information provided by the aformentioned components to produce the corresponding Data flow implementations. These implementations can be directly called from the CPU by using the *SLiC* (Simple Live CPU) interface. The SLiC interface is automatically built based on the current Data flow program and allows easy access to DFEs from the attached CPU. Furthermore, the DFE has access to two types of Memories. The first one is called *FMem* and it is an on-chip memory, which can store up to several megabytes of data, whereas the second is called *LMem* and is an off-chip memory, which can store several gigabytes of data. The fact that FMem is an on-chip memory implies that the distance between the computation units and the FMem itself is short. Thus, applications exploit the fact that data in memory are close to the computation units. Moreover, FMem offers a broad bandwidth that can reach 21TB/s within the chip. This is a significant advantage compared to traditional CPUs with multi-level caches since there only the fastest cache memory level is close to the computational unit and data are duplicated through the cache hierarchy. Finally, DFEs are controlled by the system manager, who assigns one, or more in case there are more than one DFEs, for a specific application and sets them free whenever they have completed their task.

2.1.3 Data Flow Applications

Indisputably, Data flow computing has substantially affected various scientific areas. The reasons for its wide applicability during the past few decades are mainly due to its ability to radically increase performance, to provide solutions to problems that could not be previously solved, or even because their has been an amazing development of tools.

Some examples of applications which have benefited from the existence of data flow models are Monte Carlo simulations, financia tree-based PDE solvers and finite difference solvers. Specifically for the example of finite difference solvers, we have the following problem. In general, finite difference constitutes a numerical method which is commonly used for wave modeling. More specifically, modeling high wave frequencies such as 70 Hz, may require hundreds of gigabytes of memory. However, with the use of a data flow platform such as Maxeler, which is integrated with a host CPU this can be avoided and an equivalent accelerated mwave model can be constructed. In order to do so, the CPUs can orchestrate the general flow of the application by instructing the DFEs to compute the implementation's steps. On their part the DFEs can work together on the whole problem by splitting it into sub-problems.

We have presented a detailed description of the general notion data flow, in order to introduce the user to the concepts and goal of our work. In case the reader desires to further study data flow notions we refer him to [17], [18], [5], [19].

2.1.4 Maxeler System Description

The Maxeler data flow architecture can comprise up to 8 DFEs described in 2.1.2 which are interconnected with the use of a *MaxRing* high-bandwidth connector. This allows the applications' performance to linearly scale with the number of DFE used, while allowing full overlap of communication and computation. The respective Figure is shown in 2.1.

Experiments were conducted on a specific product of Maxeler Technologies, which belongs to the MPC C Series. The MPC permits standalone development of data flow models with fixed combinations of coupled CPUs and DFEs. More specifically, the DFE used in our application is built on Virtex 6 FPGA. It also provides 4 megabytes of on-chip BRAM located on the FPGA and 24 gigabytes of off-chip DRAM. However, the same Maxeler platform provides four available such DFEs.

To deploy an application on a Maxeler data flow system, one must use *MaxJ*. MaxJ is a high-level language which is build on Java, but which has also imported instructions to define data flow graphs. In order to provide a better understanding of MaxJ we present some representative functionalities it offers.

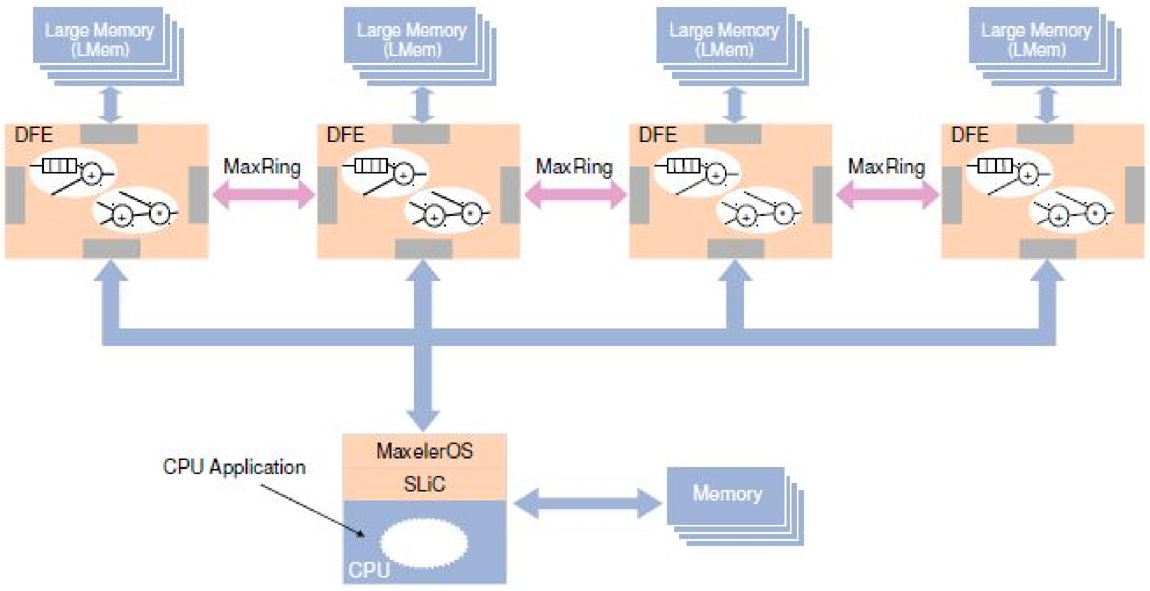


Figure 2.1: Maxeler dataflow system architecture

- The communication between the kernel and the rest of the system is achieved with an I/O interface. In particular, this interface allows the declaration of kernel input and output instances.

- A stream does not only offer access to the current instance, but to previous or following ones too. This is accomplished by using the stream offset, which is actually an offset window stored in the FPGA's BRAM. Thus, it can retrieve at most few hundreds of elements.

- Useful components for data flow designs are also provided. For example, counters are offered by the API since they are very helpful to enumerate loops.

- In order to select between values of streams Maxeler offers multiplexers which are declared by the conditional operator. Other overloading arithmetic operations are also provided.

Note that the MaxJ hybrid language allows instructing the kernel about the operations it will perform.

Another crucial component of the Maxeler data flow platform is the *MaxCompiler*. The MaxCompiler is basically a high-level compiler specifically designed for Maxeler Technologies. Similarly to the I/O interface provided by MaxJ, MaxCompiler consists of a Java-based API with which the user defines the hardware design that he intends to assign to a DFE. This design will initially be compiled and then uploaded to the declard DFE. It also offers a C runtime interface targeted for the part of the application which runs on the host. The above are implemented in the *Manager* which controls the inputs and outputs of the kernel, the relation between different kernels (multi-kernel designs) and the communication of kernels with the DRAM memory, or the CPU interface (via PCI). Thus, the manager creates the complete design by connecting its different pieces (kernels), which is then compiled to a Xilinx bitstream ready to be downloaded to the FPGA.

Moreover, the MaxCompilerRT API that interfaces with MaxelerOS is the host application which enqueues the input streams and runs the hardware design. Finally, each MaxCard features a large external memory called LMem. The LMem memory was studied while implementing our applications, and thereby we will describe this component in more detail in the following section.

2.1.5 LMem

As its name implies, **LMem** is an off-chip memory which stores several gigabytes of data. More specifically, our Maxeler platform contains four FPGAs with 24 GB each, thus large amounts of data can be stored there. These data can be traversed by the DFE but in terms of streams. Also, the parts that need to be iterated over should be declared as part of the hardware design. Thus, random accesses are not applicable. Furthermore, when defining memory addresses the LMem visually appears as a contiguous memory element. The maximum number of streams that can be connected to LMem, which also shows the number of available ports, is equal to 15.

In order to access the LMem memory, the DFE component contains a memory controller which offers the respective interface. The streams connected to the LMem are defined in the manager and the host and each stream has its own command queue and data buffer. We can write to or read from the LMem from the manager, the CPU, or both. The memory controller uses the first of the two structures to read a command.

Then, based on the type of command it either reads a stream of data from the data buffer to write it to a specific memory location or it reads a stream from a specific part of the memory and writes it to the data buffer. The respective LMem controller architecture is illustrated in 2.2.

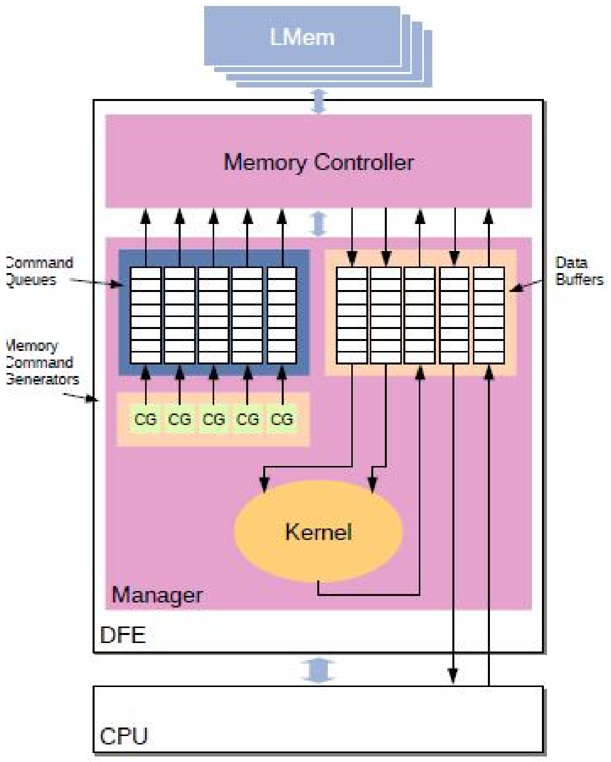


Figure 2.2: LMem Controller Architecture

For further details regarding the Maxeler technologies we refer the user to [20], [21] and the official site of Maxeler.

2.2 Support Vector Machines - Binary Classification Case

Support Vector Machines (SVMs) were first introduced by Vapnik et al. in the early 90s [22] and [23]. The proposed method has become one of the most influential Machine Learning algorithms of the decade by yielding high classification accuracy, thus finding setting in various scientific areas. Few examples of its wide applicability are presented in [24], [25], [26], [27], [28], [29], as well as countless other applications. In this thesis we will emphasize our study on the case of binary classification using SVMs, which is a simple, yet very effective version of the specific algorithm. Like other binary classifiers, binary SVMs have shown impressive performance in many applications such as speaker identification [30], text classification [31] and face recognition [32]. Significant features which have established the success of SVMs are the following. First, the specific method is based on a theoretical method of learning, and therefore comes with theoretical guarantees concerning its performance. Also, it consists of discrete parts which can be implemented individually and uses the quadratic optimization problem to avoid local minima. Finally, SVMs do not suffer from the curse of dimensionality.

2.2.1 Introduction to SVMs

In general, Support Vector Machines are the best known algorithm of the family of kernel methods. Due to the fact that SVMs are a linear approach, they can only classify data when these are linearly separable. The basic idea is that by using an optimization algorithm, SVMs find the optimal hyperplane which linearly classifies patterns. Specifically, for the case of binary SVMs the hyperplane divides data into two classes. However, SVMs have captured the attention of the research community mainly because they can be seamlessly extended to non-linearly separable input data by mapping these data to a higher-dimensional space. This is achieved with the use of kernels. In order to be able to present the essence of the SVMs method, in this section we provide certain fundamental notions.

The Classification Problem: It is widely accepted that the problem of classification is one of the most significant problems in the field of Machine Learning. Briefly, there is a set of objects $\mathcal{M}$ and each of these objects can be assigned to one of $\mathcal{N}$ distinct classes. The goal of classification is to create a machine that will correctly determine to which class $c \in \mathcal{N}$ will each object $o \in \mathcal{M}$ be assigned when provided with previous observations about this object. The area of Machine Learning provides solutions to this problem by focusing on learning from previous data observations. In order to do so, an initial machine receives training data from a finite set of samples and learns how new (actual) data should be classified correctly.

Note that this approach does not require any prior knowledge regarding the nature of the problem, or the values of the training data.

Linear Models: Linear Models for classification divide input data into classes with the use of decision boundaries. Specifically for two dimensions, this decision boundary is a line. Now, let us assume that we are examining the binary classification problem which only requires two classes, a positive and a negative one. Then, a linear binary classifier defines a plane in the space which separates positive from negative inputs.

Hyperplane: We mentioned that for a two-dimensional linear classification problem the decision boundary is a straight line. However, in a higher-dimensional problem > 2 the decision boundary is called a hyperplane. More specifically, a hyperplane comprises all points that belong in a d-dimensional space which meet the following equation:
$w_1 x_1 + w_2 x_2 + ... + w_d x_d + b = 0$
where each of w_i denotes a weight on the respective feature x_1. From the perspective of geometry the weight coefficients are normal vectors of the separating hyperplane, i.e. they are perpendicular to the plane. Given the above equation about which datapoints belong to the hyperplane we can conclude how the hyperplane can be used for binary classification. More specifically:
For $f(x) = w_1 x_1 + w_2 x_2 + ... + w_d x_d + b$
Then

$$y(x) = sign(f(x)) = \begin{cases} +1 \text{ if } f(x) \geq 0 \\ -1, \text{ otherwise} \end{cases}$$

In other words, if the sign of function $f(x)$ is positive, the input vector belongs to the positive class, while if the sign of function $f(x)$, then it belongs to the negative class. Note that the value of $f(x)$ can also be zero, which means that the datapoint is part of the hyperplane. Furthermore, the hyperplane's slope is defined by coefficients w_i. The bias b shows the perpendicular distance of the hyperplance to the origin and can be included in the weights' vector in order to get:
$f(x) = \sum_{i=0}^{d} w_i x_i = \mathbf{wx}$. Note that for linearly separable data an infinite number of hyperplanes exist. Thus, selecting the optimal one is one of our main concerns.

The Binary Classification Setting: The *input* of the classification problem is a dataset which comprises data samples $x_1, x_2, ..., x_n$, where $x \in \mathcal{X}$ of arbitrary number which can be any type of objects, numerical or non-numerical. For example, these input instances can be string sequences, time-series, previous actions and others. The *output* corresponds to a real number $y_1, y_2, ..., y_k$, where $y \in \mathcal{Y}$ and specifically for the case of binary classification this becomes y_1, y_2, where $y \in \mathcal{Y}$. The values y_1 and y_2 denote the corresponding class of the input instance x_i. A real and quite popular example of binary classification is that of separating emails to spam and non-spam, where x_i corresponds to an email and y_i is the value defining if the respective mail is spam or not. Now, recall that in order to achieve successful data classification for inputs that have not been observed yet, a machine needs to learn how to assign each of them correctly to a discrete class. Thus, it requires a training dataset containing several data instances. Typically there can be lots of training data samples depending on the model complexity and noise ration in the data. A training dataset consists of *input/output* pairs $x, y \in \mathcal{X}\text{x}\mathcal{Y}$, where similarly to the actual classification problem x is an input object and y is its respective label. Note that the training set contains both the input and the answer vectors in order to "train" the machine to correctly predict the output y value for a new input x. The goal of the training process it to configure a satisfying set of weights w, that were presented in the Hyperplane description 2.2.1. Briefly, a weight vector corresponds to each x sample and their linear combination computes the value of y. Hence, the learning phase will utilize the training samples towards this objective.

Support Vectors: The Support Vectors are the essence of the SVM algorithm and are nothing more than the data points located closest to the decision surface. Due to the fact that their correct output may not be straightforward, these instances are the hardest to classify. However, this also renders them the main factors of the optimal location of the classifier. We will describe in detail their functionality to the SVM algorithm in section 2.2.2.

Feature Space: In general, an object is described by one or more characteristics. For example, a car may be described by its price, size, brand, age and others. These numerical representations of an object are called features and determine the length of the n-dimensional input vector x. The fact that the size of the vector is directly associated with its respective features, leads to naming the corresponding vector space, feature space.

2.2.2 Linear Case

This section describes how to find the optimal binary classifier between linearly separable input instances with the use of the SVM method. In section 2.2.3.1 we illustrated the difference between linearly separable and non-linearly separable data. Briefly, data are linearly separable if there exists at least one straight line that divides them in such way that all common points are gathered in the same side of the line. Let us assume that we are trying to predict whether the values of different stocks will rise or fall in the next minute. If a linear binary classifier exists to separate these data to those whose price will rise and to those whose price will fall, then we consider these inputs to be linearly separable. This would lead to perfect classification.

So far, we have generally described the binary classification problem, along with fundamental notions necessary for the SVM method. From now on we will be focusing on the algorithm we are studying, and thereby we expect the reader to be familiar with the terms presented in the previous sections. Support Vectors were concisely described in 2.2.1. Even though these datapoints are the most difficult instances to classify, at the same time they are the most critical ones regarding the computation of the optimal hyperplane. In particular, Support Vectors are the elements of the training dataset that would alter the position of the optimal hyperplane, if removed. Thus, in the SVM method the eventual optimal decision boundary is solely defined by the Support Vectors. We stress out that the overall goal of SVM is to devise a computationally inexpensive approach to discover good hyperplanes in a high-dimensional feature space. In order to do so, SVM utilizes the following techniques.

2.2.2.1 Maximal Margin

labelmaxmargin In the attempt to construct the optimal hyperplane, SVM uses the **Maximum Margin Classifier** as an example of a linear classifier. In 2.2.1 we defined the classification hyperplane as:

$$x_i w + b \geq +1, \mathrm{for} y_i = +1 \tag{2.1}$$

$$x_i w + b \leq -1, \text{for} y_i = -1 \qquad (2.2)$$

Both constraints can be combined into one set of inequalities:
$y_i(x_i w + b) \geq +1$
Consider the limiting cases:
$x_i w + b = +1$
$x_i w + b = -1$
equal to the hyperplanes H_1 and H_2 respectively. We also denote d^+ as the distance between H_1 and H, and d^- as the distance between H_2 and H. In other words, d^+ is the shortest distance of the decision boundary to the closest positive point and similarly d^- is the shortest distance of the decision boundary to the closest negative point. Furthermore, the sum of d^+ and d^- i.e. the distance between the closest positive and negative points is the margin of the hyperplane H. Intuitively, we aim at maximizing the margin of the plane (the distance between the two classes) to avoid making erroneous classifications. That said, it is clear that SVM wants to compute the maximal margin.

Even though we presented the basic idea of the maximal margin we did not define it in computational terms. For this purpose we need to express distances in terms of the input instances x, the weight vector w and the bias b. Recall that the distance between a point (x_0,y_0) and a line $Ax + By + c$ is given by the following formula:
$\frac{|Ax_0+By_0+c|}{\sqrt{A^2+B^2}}$
Then, the distance between H_1 and H is:
$\frac{|wx+b|}{\|w\|} = \frac{1}{\|w\|}$
Note that the lines H_1 and H_2 are parallel, and therefore they share parameters w and b. Hence, the distance between H_1 and H_2 is $\frac{2}{\|w\|}$. Given that $\|w\|$ is in the denominator, one realizes that in order to maximize the above quantity he needs to minimize $\|w\|$. Hence, instead of maximizing the margin we minimize the Euclidean norm of the weight vector w. However, we should also bear in mind that no points should lie between H_1 and H_2. This is a constraint we need to take into account during the process of maximizing the margin. The aforementioned considerations are transformed into a constrained optimization problem which is described just below.

2.2.2.2 Quadratic Optimization Problem

The optimization problem derived from the necessity to maximize the margin can be stated as follows:
$max \frac{2}{\|w\|}$ subject to $\begin{cases} x_i w + b \geq +1, \text{ for } y_i = +1 \\ x_i w + b \leq -1, \text{ for } y_i = -1 \end{cases}$
However, the same problem can be transformed into an equivalent minimization one due to the fact that the Euclidean norm of w is in the denominator. In particular, we have an optimization problem where we minimize function f subject to $g(x) = 0$:

$$\min f : \frac{1}{2} \|w\|^2 \text{subject to } g : y_i(x_i w + b) \geq +1 \tag{2.3}$$

This is a quadratic optimization problem subject to certain restrictions and there is a unique minimum to be found. The solution to this problem is computed using the Lagrangian multiplier method.

2.2.2.3 The Lagrangian Formulation

In the constrained optimization problem described in 2.2.2.2 consists of two parts, the problem and the constraints. More specifically, the quantity we intended to minimize is called a cost function, and specifically in this case it is quadratic and convex. Furthermore, the constraints are linear. Due to the nature of this problem, we can introduce Lagrange multipliers in order to solve it. The number of Lagrange multipliers $a_i \geq 0$ depends on the number of equality constraints as there is a one-to-one correspondence between them. So, in our case there is only one Lagrange multiplier.

In order to formulate the Lagrangian we need to combine f and g to get the following general formulas:

$L(x, \lambda) = f(x) - \lambda g(x)$
$\partial(x, \lambda) = 0$
The two partial derivatives wrt x and λ retrieve the linear constraint and $g(x, \lambda) = 0$ respectively. In general we have the following formula:
$L(x, \alpha) = f(x) - \sum_i a_i g_i(x)$
So, by substituting the variable values with the appropriate ones of our problem we get:
$L = \frac{1}{2} \|w\|^2 - \sum_{i=1}^{l} a_i [y_i(x_i w + b) - 1] =$

$L = \frac{1}{2}\left\|w\right\|^2 - \sum_{i=1}^{l} a_i[y_i(x_i w + b)] + \sum_{i=1}^{l} a_i$, with $a_i \geq 0, \forall i$
Or equivalently:

$$L = \frac{1}{2}\left\|w\right\|^2 - \sum_{i=1}^{l} a_i y_i x_i w - b\sum_{i=1}^{l} a_i y_i + \sum_{i=1}^{l} a_i \tag{2.4}$$

2.2.2.4 The Dual Problem

The above quantity should be minimized with respect to primal variables w and b and maximized with respect to the dual variable a. In such optimization problems the Duality Problem [33] states that the solution to the dual problem sets a lower bound to the solution of the original minimization (primal) problem. So, if the primal-minimization problem has an optimal solution, then the dual-maximization problem will also have an optimal solution. However, it is not necessary for the optimal values to be equal, and their in between difference is called the duality gap. Still in convex optimization problems that satisfy a constraint qualification (our case) this gap equals zero and by solving the dual optimization problem, we find the optimal value of the primal problem. Moreover, we compute the solution for our primal problem by differentiating the primal Lagrangian wrt w and b. Hence, we need to equate the corresponding partial derivatives to 0:

$$w = \sum_{i=1}^{l} a_i y_i x_i \tag{2.5}$$

$$a_i \geq 0, \forall i \tag{2.6}$$

Note that with 2.6 the third term in 2.4 is zero. Now, we are going to apply the Lagrangian trick which means that the constraints will be replaced by equivalent constraints on the Lagrange multipliers and the input data will appear only in the form of dot-products. Recall, the significance of dot-products for the Kernel computation described in 2.2.3.1. Finally, after taking into account 2.5, 2.6, 2.4 and the Lagrangian trick we get the formalization of our dual problem:

$$L_d(a) = \sum_{i=1}^{l} a_i - \sum_{i=1}^{l}\sum_{j=1}^{l} a_i a_j y_i y_j \mathbf{x_i}\mathbf{x_j} \tag{2.7}$$

Now, according to the duality problem we need to maximize quantity 2.7 by computing the optimal Lagrange multipliers a_i subject to the constraints:

$$\sum_{i=1}^{l} a_i y_i = 0 \tag{2.8}$$

$$\sum_{i=1}^{l} a_i y_i = 0 \tag{2.9}$$

At this point we have formulated our final problem, which however cannot be solved with simple mathematical techniques. Instead, we need to use quadratic optimization techniques. In addition, notice that each input vector x_i will correspond to its own Lagrangian multiplier a_i. That said, we can now describe the critical Support Vector datapoints described in 2.2.1 in terms of the unknown Lagrangian multipliers a_i. More specifically, the points for which $a_i > 0$ comprise the set of Support Vectors and are either located in H_1 or H_2 2.2.1. The rest of the datapoints lie either beyond H_1 or beyond H_2 but not between them, so that the strict inequality constraint 2.3 is met. The final step, is to write the equation which finds the optimal hyperplane (maximum margin classifier) 2.1 and 2.2 in terms of the duality problem. Then we get:

$$\mathbf{w_o x} + b_o = \sum_{i=1}^{l} a_(i,o) y_i x_i \mathbf{x} + b_o = 0 \tag{2.10}$$

where o denotes optimal and the bold characters show vectors. Similarly the decision function becomes:

$$sign(\mathbf{w_o x} + b_o) = sign(\sum_{i=1}^{l} a_(i,o) y_i x_i \mathbf{x} + b_o) \tag{2.11}$$

On finding the values of a_i we assume that these are given to us as it is not in the scope of this thesis to explore optimization.

2.2.3 Nonlinear Case

Until now we only studied linearly separable input vectors. However, we mentioned that in real datasets researchers encounter high-dimensional and non-separable data. In order to deal with this more complex problem, the idea of Nonlinear SVM was proposed. The general idea is to map the original input data into a higher-dimensional space to gain linearity with the use of kernels. The organization of this section is as follows. First, we will provide a detailed description along with necessary definitions about kernels. Then we will present how do kernels get involved with SVM as well as the final formulation of the SVM problem for the Nonlinear case.

2.2.3.1 Kernels

Kernels a.k.a kernel functions are a key component of significant pattern analysis algorithms due to their effectiveness when operating on real world data and relations. In particular, it is common for an actual data analysis problem to involve non-separable data, thus making it hard to extract even discrete dependencies that would allow successful prediction of properties of interest. However, despite the necessity for efficient non-separable pattern analysis methods, most Machine Learning algorithms have been developed and implemented on linearly separable data, which renders the use of kernels essential. Our ultimate goal is to use kernels in order to learn a decision function which will reasonably classify unseen input data to known classes. In this section we will first introduce the reader to the problem of non-linearly separable data by illustrating the problem. Then, we will describe the approach of kernels in more detail by providing all necessary mathematical notations and definitions.

Prior to posing a formal definition of kernels we illustrate the difference between linearly and non-linearly separable data in Figure 2.3 which stresses out the importance of kernels.

The above figure presents linear and non-linear data that can be divided into two classes i.e. can be binary classified. The former are shown in the left figure, whereas the latter are depicted in the right one. In the very next Figure 2.4 we show some of these separators which are denoted by the letters H_1, H_2, H_3. It is clear from Figure 2.4 that in both cases the data can be classified by an infinite number of classifiers. However, certain issues arise such as finding the best separators from an infinite set, or computing non-linear classifiers when dealing with non-linearly separable data. In this section we will focus on the second question which tackles the computationally difficulty of representing data in a higher space.

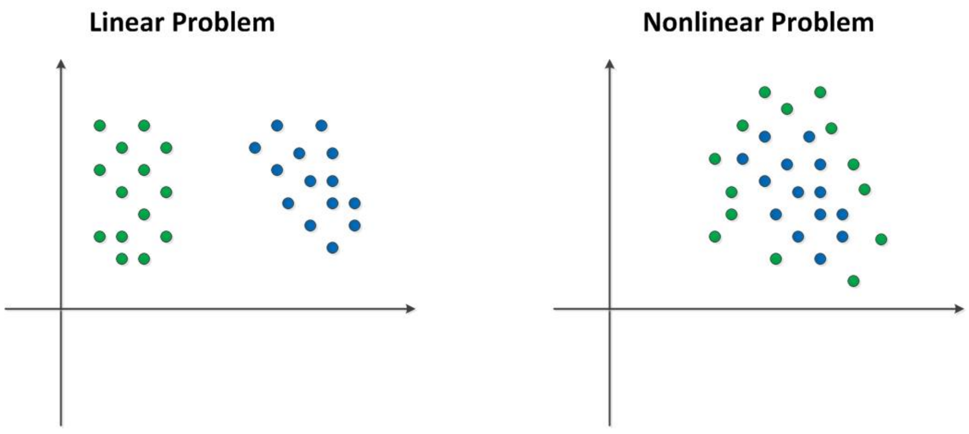


Figure 2.3: Linear and Nonlinear Data

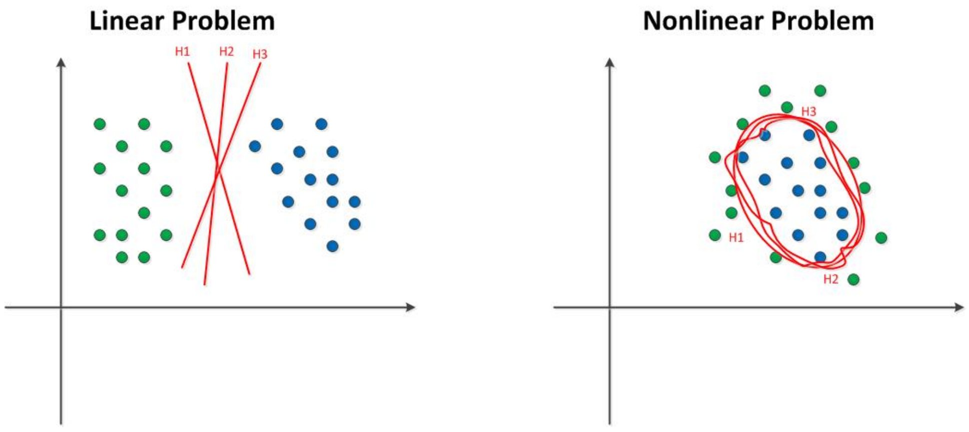


Figure 2.4: Linear and Nonlinear Data Separation

Definitions

In general, kernel functions (kernels) map datapoints to an alternative higher-dimensional vector where linear relations exist and linear separation is feasible. Note that representing data in a higher space is computationally difficult. More specifically, let us assume we have the following setting with $\mathcal{X}$ and $\mathcal{Y}$ being vector spaces usually in $\mathbb{R}^N$ and $\mathbb{R}$ respectively:

$X := \{x_1, x_2, ..., x_i\} \subset \mathcal{X}$

$Y := \{y_1, y_2, ..., y_i\} \subset \mathcal{Y}$

The domain $\mathcal{X}$ is some nonempty set which contains training datapoints and includes inputs x_i, so $x_i \in \mathcal{X}$. The domain $\mathcal{Y}$ denotes target values and includes inputs

y_i, thus $y_i \in \mathcal{Y}$. We also encounter inputs x_i and y_i in the following brief form:

$(x_1, y_1), (x_2, y_2), ..., (x_i, y_i) \in \mathcal{X}\text{x}\mathcal{Y}$

Specifically for the case of binary classification, which is of our main interest the above declaration becomes:

$(x_1, y_1), (x_2, y_2), ..., (x_i, y_i) \in \mathcal{X}\text{x}\{+1, -1\}$

Note that we have not assumed anything about the set $\mathcal{X}$. Given that our initial goal was to address the problem of learning we use the above setting to formulate this problem better. More specifically, in data analysis learning for a new input $x_j \in \mathcal{X}$ we want to predict the respective $y_j \in \mathcal{Y}$, which specifically for the binary classification case is $y_j \in \{+1, -1\}$. From now on we will only address the problem of binary classification, as it is not in the scope of this thesis to study more complex output domains $\mathcal{Y}$. This new x_j datapoint is not a training one as those described in the previous paragraph, but is usually referred to as actual or testing datapoint. That said, the assignment of a y_j value to input x_j should be as similar as possible to the training examples, and therefore we need an indication of similarity in $\mathcal{X}$ and $\{+1, -1\}$. For binary classification, finding the similarity between the target values is easy as there are only two possible values $+1$ and -1, and therefore they can be either identical or different. However, in the former case and for the same purpose we seek a better representation of the data by mapping their respective vectors from the original space to an alternative higher-dimensional space, called the *feature space*. Now, the similarity between x_i and x_j is calculated in the feature space. In other words we perform the following mapping:

$$(x_i, x_j) \longmapsto \Phi(x_i), \Phi(x_j), \qquad \forall x_i, x_j \in \mathcal{X} \tag{2.12}$$

We define $k(x_i, x_j) =< \Phi(x_i), \Phi(x_j) >$, as a kernel function. Note that the functional forms of the mappings $\Phi(x_i)$ and $\Phi(x_j)$ do not concern us because they are defined by the selection of the kernel $k(x_i, x_j) = \Phi(x_i) * \Phi(x_j)$ or by the dot-product in the feature space. We can reformulate the above mapping to the following:

$k : \mathcal{X}\text{x}\mathcal{X} \to \mathbb{R}, \qquad (x_i, x_j) \longmapsto k(x_i, x_j), \qquad \forall x_i, x_j \in \mathcal{X}$
The kernel functions follow the following properties:

- They are *symmetric*,i.e., $k(x_i, x_j) = k(x_j, x_i)$
- They are *positive semi-definite* for every finite set datapoints.
- They are *continuous*.

In particular, we define a kernel function as follows based on [34]: *For any set $\mathcal{X}$ a function K:$\mathcal{X}^2 \to \mathbb{R}$ is a kernel, i.e. it is symmetric and positive semi-definite, if and only if there is a mapping Φ from $\mathcal{X}$ into a Hilbert space $\mathcal{H}$ with a scalar product $h < .,. >$ such that $k(x_i, x_j) = < \Phi(x_i)$,$\Phi(x_j) > \forall\ x_i$,$x_j \in \mathcal{X}$*

Intuitively, one can consider kernel functions as similarity metrics in the feature space Φ, due to the fact that they are equal to dot products. Note that the dot-product $x_i * x_j$ is maximized when $x_i = x_j$ and minimized when the two vectors point to opposite directions. When x_i, x_j are orthogonal the dot-product equals 0. In section 2.2.3.2 we will see that replacing a dot-product with a kernel is very useful and is referred too as the kernel trick.

Examples of Kernels:
Throughout the years several kernel functions have been presented by researchers. However, simply choosing a kernel is not sufficient as the majority of kernel functions also require fine-tuning their respective parameters which can be a time-consuming process. For this purpose several works have proposed solutions to ease this process [35], [36], [37],[38], [39]. Works [40] and [41] by Ali et al. and Li et al. even discuss automatic kernel selection with the use of statistical measures derived from the data sets, as well as with extensive empirical performance outcomes. Still, despite the popularity of kernels this scientific area lacks an effective method to guide the selection of a kernel function and parameters. It is not in the scope of this thesis to study proposed kernel functions and parameter selection techniques. We will however introduce basic kernels, due to the fact that they should be known to anyone interested in deepening his knowledge in kernel methods.

Finding an effective kernel significantly depends on the input data and the problem we are examining. For example, different kernel functions have been implemented for numeric and non-numeric data, such as sequences and structures. Also, depending on the number of features of the data the user might select a less computationally expensive kernel. The following kernels have been proved to be practical for general applications:

Radial Basis Function: The Radial Basis Function (RBF) a.k.a Gaussian kernel function is one of the most popular kernels, broadly used in many kernel-based algorithms and especially in Support Vector Machines. More specifically given two feature vectors x_i and x_j which are actually datapoints in an origin input space, the RBF kernel equals:
$k(x_i, x_j) = exp(-\frac{\|x_i - x_j\|^2}{2\sigma^2})$
where σ^2 is a free parameter called variation and $\|x_i - x_j\|$ is the Euclidean distance between x_i and x_j. Note that tuning σ appropriately is very important for the performance of the kernel. On one hand, if overestimated then the exponential will approach a linear behavior. On the other hand, if underestimated then the decision boundary will be highly sensitive to noise in training data. It is common to encounter the RBF function in the following form too:
$k(x_i, x_j) = exp(-\gamma \|x_i - x_j\|^2), \gamma > 0$
where γ is an adjustable kernel parameter that controls the width of the RBF kernel. The value of RBF ranges from 0 to the limit to 1. The higher the RBF value, the more similar the two input vectors.

Polynomial: The Polynomial function is frequently used in Natural Language Processing. More specifically given two feature vectors x_i and x_j which are actually datapoints in an origin input space, the Polynomial kernel equals:
$k(x_i, x_j) = (x_i^T x_j + c)^d$
where $c \geq 0$ is an adjustable parameter and d is the degree of the polynomial. The degree is usually set to 2, since for greater values it tends to overfit.

Linear: The Linear kernel is the simplest known kernel function as it only consists of a dot-product plus, sometimes, a constant factor. More specifically given two feature vectors x_i and x_j which are actually datapoints in an origin input space, the Linear kernel equals:
$k(x_i, x_j) = x_i^T x_j + c$
Even though using a Linear kernel, instead of using an RBF kernel, solves the

optimization problem much faster, it has been shown in [42] that the Linear kernel is actually a degenerated version of the RBF kernel. Hence, if the model selection of the RBF kernel has been performed optimally, then it is impossible for the Linear kernel to yield more accurate results than the RBF kernel.

Other popular kernels are the Fisher Kernel [43] for statistical classification, the Graph Kernel [44] for computing similarity between graphs and the String Kernel for finding similarity between pairs of strings. The aforementioned description of kernel methods is by no means complete. For further details we refer the reader to the books of ShaweTaylor and Nello Christianini in [45], Smola and Schölkopf in [46] and Joachims in [31]

2.2.3.2 Kernel Trick

In section 2.2.3.1 we presented kernel functions which are efficient methods for computing the similarity between non-linearly separable datapoints. These kernels allow the configuration of the kernel trick; a tool of great importance which links linearity and non-linearity in any algorithm that can be written using only dot-products between vectors. The interest of the kernel trick derives from the fact that by mapping a linear method's input data (vectors) into the feature space, then the same algorithm will operate non-linearly in the original space and linearly in the feature space.

In order to clarify our point we provide the following example. Let us assume we are using the polynomial kernel for input vectors x_i and x_j. We are going to show that the polynomial kernel $k(x_i, x_j) = (x_i^T x_j + c)^d$, with constant value c equal to 0 and degree d equal to 3 for simplicity reasons, yields the same result as the explicit mapping and dot-product.

$\Phi :\quad \mathbb{R}^2 \to \mathbb{R}^3$

$\Phi :\quad (x_{i1}, x_{i2}) \longmapsto (z_{i1}, z_{i2}, z_{i3}) := (x_{i1}^2, \sqrt{(2)}x_{i1}x_{i2}, x_{i2}^2)$

Then the dot-product of the two mappings equals:

$\Phi(x_{i1}, x_{i2}) * \Phi(x_{j1}, x_{j2}) = (x_{i1}^2, \sqrt{(2)}x_{i1}x_{i2}, x_{i2}^2) * (x_{j1}^2, \sqrt{(2)}x_{j1}x_{j2}, x_{j2}^2)$

$= x_{i1}^2 x_{j1}^2 + 2x_{i1}x_{i2}x_{j1}x_{j1} + x_{i2}^2 x_{j2}^2$

This is the same as:

$k(x_i, x_j) = (x_i x_j)^2 = ((x_{i1}, x_{i2})(x_{j1}, x_{j2}))^2 = (x_{i1}x_{j1} + x_{i2}x_{j2})^2$

$= x_{i1}^2 x_{j1}^2 + 2x_{i1}x_{i2}x_{j1}x_{j1} + x_{i2}^2 x_{j2}^2$

Note that in a case where the dimension d is very big the explicit computation of the mapping and the dot-product may not fit in memory. However, the kernel computation will still only require n multiplications, where n is the size of vectors x_i, x_j.

2.2.3.3 Nonlinear Formulation

The significance of non-linearly separable data, the applicability of the key components called kernels to gain linearity and the simplicity of computing dot-products using the kernel trick led to the possibility of extending linear SVM to the non-linear case.

More specifically, by transforming input datapoints of an original space to a higher-dimensional one by mapping 2.12 and by recomposing the Dual Lagrangian Problem presented in 2.7, we get the following optimization problem:

$$L_d(a) = \sum_{i=1}^{l} a_i - \sum_{i=1}^{l}\sum_{j=1}^{l} a_i a_j y_i y_j \Phi\mathbf{x_i}\Phi\mathbf{x_j} \tag{2.13}$$

The optimal hyperplane is 2.10:

$$\mathbf{w_o x} + b_o = \sum_{i=1}^{l} a_(i,o) y_i \Phi x_i \Phi\mathbf{x} + b_o = 0 \tag{2.14}$$

Then, the optimal decision function becomes 2.11

$$sign(\mathbf{w_o x} + b_o) = sign(\sum_{i=1}^{l} a_(i,o) y_i \Phi x_i \Phi\mathbf{x} + b_o) \tag{2.15}$$

Note that equations 2.14 and 2.15 only depend on dot-products between the input data in some feature space. The actual dimensions of the specific feature space, as well as the mapping function are considered indifferent to us due to the kernel function definition $k(x_i, x_j) =< \Phi(x_i), \Phi(x_j) >$. Recall that the kernel function allows us to directly compute the dot-product of the mapped data without explicitly expressing in terms of the map function and the dot-product. Hence, the kernel computation is only dependent of the dimensions of the input space and disregards the dimensions of the feature space.

Finally, with the use of the kernels the dual lagrangian equation 2.13 and the equation of the optimal plane 2.14 become:

$$L_d(a) = \sum_{i=1}^{l} a_i - \sum_{i=1}^{l}\sum_{j=1}^{l} a_i a_j y_i y_j k(x_i, x_j) \tag{2.16}$$

$$\mathbf{w_o x} + b_o = \sum_{i=1}^{l} a_(i,o) y_i k(x_i, \mathbf{x}) + b_o = 0 \tag{2.17}$$

We have describe the method of SVM in order to provide a sufficient understanding of how the specific method works, since it is one of the state-of-the-art classification algorithms. For further details we refer the reader to the following works [47], [48], [22].

2.2.4 LIBSVM

As its name implies, **LIBSVM** is library for Support Vector Machines, i.e. an integrated software tool targeted for Support Vector classification (two-class and multi-class), regression and distribution-estimation. The specific project began in 2000 and ever since it has gained wide acceptance by machine learning applications. At the time this thesis is being written, the LIBSVM publication written by Chang and Lin [49] has 22028 citations and more than 250, 000 downloads, whereas the next most popular SVM tool called SVM Light [50] has 6685. In addition to the high number of citations, there are also several representative works in various domains which show the applicability of LIBSVM. Some of these works are [51] in Computer Vision, [52] in Natural Language Processing, [53] in Neuroscience and [54] in Bioinformatics.

The LIBSVM package is based on the two basic steps that any supervised learning method follows, the first being the training phase and the second being the classification phase. In brief, first we insert a training set with data samples to create the classification or regression model, and then we use the created model to predict information about actual data. Moreover, the structure of LIBSVM is as follows. There is a main directory composed by C/C++ source codes and the required datasets, a sub-directory which comprises tools to guarantee that the input datasets have the appropriate format, as well as other sub-directories with pre-built binary files and interfaces for integration with other software environments and languages. This thesis focuses on Support Vector Classification. Thus, we will only present technical details concerning this part of LIBSVM.

2.2.4.1 Data Preprocessing

Prior to executing the training phase, the user has to format his datasets based on the requirements of the LIBSVM package. More specifically, LIBSVM requires that each input data is depicted on a corresponding vector. Note that the input vectors can only contain real numbers, and thereby categorical types of instances must be mapped to numbers. An informative description of a categorical data would use m numbers, for m respective attributes. In particular, the creators of LIBSVM recommend that number 1 denotes the actual value of the data instance and the rest of the values have value equal to 0. For example, a five-category instance with attributes (tourist, standard, comfort, fist class, luxury) can be represented as $(1, 0, 0, 0, 0)$, $(0, 1, 0, 0, 0)$, $(0, 0, 1, 0, 0)$ and so on. They also mention that in cases where the number of categories is not too big, then the aforementioned categorization may yield better results. Moreover, the precise LIBSVM training dataset format is the following:

$$<label> \quad <index_1> : <value_1><index_2> : <value_2>...<index_n> : <value_n> \tag{2.18}$$

where *label* indicates the feature's corresponding class, each of the indexes $index_1, index_2, ..., index_n$ show the attribute id of the instance with respective values $index_1, index_2, ..., index_n$. The reason for which LIBSVM uses indexes in addition to values, is because this allows efficient representation of sparse datasets. For example, let us assume that we have the following two data instances of an object:

$$-1 \quad 5 : 0.4 \quad 9 : 1.5 \quad 20 : -0.34 \quad 49 : -1.25$$

$$+1 \quad 5 : 1.7 \quad 12 : 0.35 \quad 14 : -1.4 \quad 36 : 0.1 \quad 49 : -1.25$$

Note that in the first case only indexes, i.e. attributes, 5, 9, 20 and 49 have a value. This implies that the rest of the indexes 1 to 4, 6 to 8 and so on have a value equal to 0 and this is represented with little information. Similarly, the second data instances has nonzero values for attributes 5,12,14,36,49 which means that it shares with the previous data instance nonzero values only for attributes 5 and 49.

Another factor that could improve performance is scaling the data samples during the preprocessing phase. The most significant advantage of scaling is that it balances bigger numerical values with smaller ones, thus none of two dominates the other. Furthermore, scaling renders computation easier. In particular, the fact that kernel computations require producing the inner product between vectors, indicates that large numbers could hamper this process. In order to overcome this difficulty, the creators of LIBSVM suggest linear scale of each attribute to the range $[-1, +1]$ or $[0, 1]$. Note that if a model is trained on scaled data instances, then it can be applied only to corresponding data. Thus, both the training and actual dataset should be scaled. For instance, if we scale the training attribute from $[-10, +10]$ to $[-1, +1]$, then an actual sample in the range $[-12, +5]$ needs to be scaled to $[-1.2, +0.5]$.

2.2.4.2 C-Support Vector Classification

As we mentioned when introducing LIBSVM, this package allows utilizing various SVM formulations for classification, regression and distribution estimation. The formulation that we have considered is the **C-Support Vector Classification** (C-SVC) which is based on the Support Vector Machines method that we described in 2.2.1. More specifically, LIBSVM solves the following dual problem:

$$\min \quad \frac{1}{2}\alpha^T \mathcal{Q} \alpha - \mathbf{e}^T \alpha \tag{2.19}$$

$$\text{subject to } y^T \alpha = 0 \tag{2.20}$$

$$0 \leq \alpha_i \leq \mathcal{C}, i = 1, ..., \mathrm{l} \tag{2.21}$$

where $\mathbf{e} = [1, ..., 1]^T$ denotes the all-ones vector, $\mathcal{Q}$ is an l by l positive semi-definite matrix, $Q_{ij} \equiv y_i y_j K(\mathbf{x_i}, \mathbf{x_j})$, and $K(\mathbf{x_i x_j}) \equiv \phi(\mathbf{x_i})^T \phi(\mathbf{x_j})$ is the kernel function 2.2.3.1. The transformation of the above problem to the primal-dual relationship yields the following optimal $\mathbf{w}$:

$$\mathbf{w} = \sum_{i=1}^{l} y_i \alpha_i \phi(\mathbf{x_i}) \tag{2.22}$$

and therefore the decision function of the C-SVC formulation is:

$$\text{sign}(\mathbf{w}^T \phi(\mathbf{x}) + b) = \text{sign}(\sum_{i=1}^{l} y_i \alpha_i K(\mathbf{x_i}, \mathbf{x}) + b) \tag{2.23}$$

The program needs to store values $y_i \alpha_i \forall i$, b, as well as all label names, support vectors, and kernel parameters.

2.2.4.3 Code Organization

Now that we have described the exact problem that C-SVC solves, we are briefly going to describe the source code's organization. All of the implemented LIBSVM methods are included in the **svm.cpp** file, which has two basic sub-routines, **svm_train** and **svm_predict**. The training phase is the most complex among the two as it is responsible for finding the support vectors with the use of an optimization method. More specifically, in the case of classification **svm_train** calls the **svm_train_one** function multiple times. Then, **svm_train_one** subsequently calls the function that corresponds to the SVM formulation that the user desires to use. For example, for classification **svm_train_one**, either calls **solve_c_svc** or **solve_nu_svc**. All of the sub-routines that correspond to specific SVM formulations, simply initialize certain parameters with suitable values for the next phase, which is the phase of solving. As implied by its name, the function *solve*, comprises of several steps which are the execution of the quadratic optimization problem with the use of two implementation tricks called Shrinking and Cashing to improve performance. However, this thesis neither focuses on the algorithm of optimization, nor on shrinking and caching. For further details regarding the implementation of the aforementioned components we refer the reader to [49]. However, we are particularly interested in the dot-product computation during the optimization process, which was described in 2.2.3.1.

2.2.4.4 Performance Measure

Like every other prediction algorithm, the performance of the LIBSVM package has been evaluated based on certain performance metrics. In particular, recall that once the optimization problem which solves the primal-dual problem has been completed, the outputted decision function can be applied to newly observed data (testing data) to predict their respective target labels. In particular, let us assume that $x_1, ..., x_l$ are the unseen input instances and $f(x_1), ..., f(x_l)$ are the predicted labels. Then, given that $y_1, ..., y_l$ are the actual true labels we appraise the success of predictions with the following metric of accuracy:

$$Accuracy = \frac{\text{\# of correctly predicted data}}{\text{\# of total testing data}}\text{x}100\% \tag{2.24}$$

Accuracy is computed only for classification tasks. In the case of regression LIBSVM uses other metrics which are the mean squared error [55] and the squared correlation coefficient [56].

2.3 Mutual Information

In this section we present a brief tutorial on **Mutual Information** (MI) as formulated in [57]. It is beyond the scope of this thesis to thoroughly study the specific term and its variations. However, it is worthwhile to provide a comprehensive background, and therefore we will also make short references to other fundamental notions of Information Theory closely related to MI, in order to yield a better overall understanding. In case the reader is interested in detailed descriptions, then they are referred to [57], [58], [59], [60] from which much of the content of this section is derived.

Indisputably, information is a very broad and abstract notion. In the attempt of formalizing it is crucial to quantify certain concepts such as how meaningful or relevant is a piece of information. Such concerns were not part of Information Theory initially. More specifically, the founder of Information Theory, Claude E. Shannon [61] emphasized on efficient data transmission rather relevance of data content. The approach that Shannon followed regarding Information Theory led to the misconception that the specific field is unrelated to denoting meaningful information, and is only targeted to the area of communication. Yet, we will describe why this is a mistaken belief based on the fact that the single information theoretic principle has been applied as a particular case in a variety of problems, such as prediction, filtering and learning.

2.3.1 Entropy

This section presents the notion of entropy, which quantifies uncertainty of a random variable. It is considered the most fundamental notion of Information Theory. Prior to formally defining Entropy we are going to present a quantified description of information. In particular, let E be an event which occurs with probability $P(E)$. Then, if event E has been observed we say that:

$$I(E) = \log_2 \frac{1}{P(E)} \tag{2.25}$$

bits of information have been received. Note that the base of the log function is 2 because we are transmitting and receiving bits. For example, the outcome of a flipped coin is $\log_2 2 = 1$ bit, whereas for a rolled dice it is $\log_2 6 \approx 2.585$. If the base changes to b, then entropy is denoted as $H_b(S)$, whereas if we have the natural logarithm with base equal to e then the entropy is computed in nuts. Moreover, information is additive, so for k fair coin tosses we get:

$$I(k) = \log \frac{1}{\frac{1}{2^k}} = k \text{ bits} \tag{2.26}$$

Thus, a random word belonging to a $100,000$ word alphabet requires $I(symbol) = \log_2 100,000 = 16.61$ bits of information. Based on 2.26, a 1000 word piece of information produced from the same source requires $16,610$ bits. Moving to another domain, a $480x640$ pixel, 16-gray scale picture needs $I(picture) = 480 * 640 \log_2 16 = 1,228,800$ bits.

Now, regarding entropy suppose we have a source S with no memory. Also, let random variable X with corresponding alphabet $\mathcal{X}$. If this variable is discrete then we have, $p_1(x_1) = prob(X = x_1), x_1 \in \mathcal{X}$, $p(x_2) = prob(X = x_2), x_1 \in \mathcal{X}$, and so on, probability mass functions. So the source S emits statistically independent symbols $s_1, ..., s_n, s_i \in \mathcal{S}$ with probabilities $p_1, ..., p_n$ respectively. Then, we define Entropy as:

$$H(S) = \sum_i p_i I(s_i) = \sum_i p_i \log \frac{1}{p_i} \tag{2.27}$$

which shows the average amount of observed information at the output of source S. Note that entropy is a functional of the distribution of X, and therefore does not depend on actual values but on possibilities. Also, notice that information reduces uncertainty and should not be confused with knowledge, since it does not reveal meaning. Let us assume that an observer is about to watch the result of a flipped coin, or a rolled dice. Then, once he has observed the final outcome the uncertainty of the result has become zero, given that he is aware of what was sent. Moreover, note that usually the logarithms in Information Theory have base 2, and thereby entropies are measured in bits. Recall that the complete characterization of source S includes probabilities $p_1, ..., p_n$ of corresponding symbols. The domain to which these symbols belong to, i.e. what they actually are, is indifferent to entropy, as this notion clearly shows the average uncertainty in the probability distribution of the symbols in source S. Therefore, equation 2.27 can be re-written as:

$$H(S) = H(P) = H(p_1, ..., p_n) = \sum_i p_i \log \frac{1}{p_i} \tag{2.28}$$

The above equation can be similarly formulated for continuous distributions. Even though an axiomatic definition of entropy has been formulated, we will discuss it towards the end of this section. Instead, we will approach the specific concept based on certain quantities that are directly correlated to it. More specifically, given the formulation of entropy 2.28 we can derive useful quantities which are the following:

- The average amount of information provided by sending a single symbol.
- The average amount of unexpectedness when observing the output of a symbol.
- The uncertainty an observer has prior to seeing the sent symbol.
- The average number of bits required to communicate a symbol. Then, the maximum efficiency can approach $H(S)$ but cannot be $< H(S)$ bits/symbol

At this point we present certain important properties of entropy:

- Entropy is always a non-negative value, i.e. $H(S) \geq 0$. Note that $0 \leq p_i \leq 1$, and therefore $\log \frac{1}{p_i} \geq 0$. It is zero only when the random value is certain to be predicted.
- Invariant with respect to permutation of its inputs
- Given any other probability distribution $q_1, ..., q_n$:
  $H(S) = \sum_i p_i \log \frac{1}{p_i} < \sum_i p_i \log \frac{1}{q_i}$
- The further the probability distribution is from the uniform distribution, the lower the entropy.
- $H(S) \leq \log k$, with equality iff $p_i = \frac{1}{k}, \forall i$
- In order to change the base of the entropy multiply the original entropy with the appropriate factor: $H_b(S) = (\log_b \alpha) H_\alpha(S)$

At this point we are going examine an actual example of entropy in a real setting. More specifically, we are interested in finding the true entropy of the English language, which has an alphabet of 27 characters. Then, a document of $100,000$ words with 5.5 average characters each, we can compute the following. Assuming independence from one character to the next and that these characters follow a uniform distribution, then the entropy required for each character equals $-\log \frac{1}{27} = 4.75$ bits/character. However, the above entropy is computed under the assumption that all characters are equally likelly and this is the zero-order model of English which provides an approximation of the English entropy but is false. In particular, some English characters are used much more frequently that others. For example, the character e has frequency 0.12702, whereas the character z has frequency 0.00074. Hence, we need to assume that all letters are independent but the actual probabilities are used. This is called the first-order model of English which produces entropy equal to 4.219. However, this number is also false as assuming independence between characters is non-realistic. More specifically, certain letters follow others with high probability. Such sequences are for example, "ON", "AND", "TH", and others. The last English model that was presented is called the higher-order model, which comprises the second-order model, the third-order model, and so on, based on whether we take into account the likelihood of digrams, or trigrams respectively. A third-order model produces 2.77 bits/symbol and this number reduces as the order of the model increases.

2.3.2 Joint Entropy and Conditional Entropy

The previous section presented the entropy of a single random variable X. The same formula 2.28 can be extended to pairs of random variables (X, Y), as pairs can actually be perceived as single random variable vectors. That said, the **joint entropy** $H(X, Y)$ of a pair of discrete random variables (X, Y) with joint distribution $p(x, y)$ is defined as:

$$H(X,Y) = \sum_{x\in\mathcal{X}} \sum_{y\in\mathcal{Y}} p(x,y) \log \frac{1}{p(x,y)} \tag{2.29}$$

For example, let us assume that we have two random variables the first being T depicting temperature with values *hot*,*mild* and *cold*, and the second being N denoting humidity with values *low* and *high*. Furthermore, for random variable T, we have probabilities $P(T = hot = 0.3$, $P(T = mild) = 0.5$, $P(T = cold) = 0.2$, whereas for random variable

N the probabilities are $P(N = low) = 0.6$ and $P(N = high) = 0.4$. In addition we also need the joint probabilities in order to compute the joint entropy. These are $P(T = hot, N = low) = 0.1, P(T = hot, N = high) = 0.1, P(T = mild, N = low) = 0.4, P(T = mild, N = high) = 0.1, P(T = cold, N = low) = 0.1, P(T = cold, N = high) = 0.2$. Then, the entropies are $H(T) = H(0.3, 0.5, 0.2) = 1.48548$, $H(N) = H(0.6, 0.4) = 0.970951$ and the joint entropy in the space of (t, n) is:

$$H(T, N) = \sum_{t,n} P(T = t, N = n) \log \frac{1}{P(T = t, N = n)} \tag{2.30}$$

or

$$H(0.1, 0.4, 0.1, 0.2, 0.1, 0.1) = 2.32193 \tag{2.31}$$

Note that $H(T, N) < H(T) + H(N)$. Also, recall that two events T, N are independent if the joint probability mass function satisfies $P(T = t, N = n) = P(T = t)P(N = n)$. Thus, one realizes that in our example T, N are not independent.

In addition to the joint entropy of two random variables X and Y we also define the **conditional entropy** of the first given the second. Conditional entropy is 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:

$$H(Y|X) = \sum_{x \in \mathcal{X}} p(x) H(Y|X = x) = \sum_{x \in \mathcal{X}, y \in \mathcal{Y}} p(x, y) \log \frac{1}{p(y|x)} \tag{2.32}$$

Once again, consider the previous example with the two random variables of temperature T and humidity N. However, now we need the conditional probabilities in order to compute the conditional entropy. Thus, let us assume that we are given the following conditional probabilities $P(N = low|T = hot) = \frac{1}{2}, P(N = high|T = hot) = \frac{1}{2}, P(N = low|T = mild) = \frac{4}{5}, P(N = high|T = mild) = \frac{1}{5}, P(N = low|T = cold) = \frac{1}{3}, P(N = high|T = cold) = \frac{2}{3}$. Then substituting formula 2.32 with the appropriate values results the following conditional entropies $H(N|T = cold) = H(\frac{1}{3}, \frac{2}{3}) = 0.918296$, $H(N|T = mild) = H(\frac{4}{5}, \frac{1}{5}) = 0.721928$, $H(N|T = hot) = H(\frac{1}{2}, \frac{1}{2}) = 1.0$ as well as the average conditional entropy:

$$H(N|T) = \sum_{t} p(T = t) H(N|T = t) = 0.3H(N|T = cold) + 0.5H(N|T = mild) + 0.2H(N|T = hot) \tag{2.33}$$

$$\approx 0.8364 \tag{2.34}$$

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:

$$H(X, Y) = H(X) + H(Y|X) \tag{2.35}$$

In addition, note that $H(Y|X) \neq H(X|Y)$, but $H(X) - H(X|Y) = H(Y) - H(Y|X)$. Prior to moving to the next section where Mutual Information is introduced, recall that Shannon's definition of entropy measured the information that was transmitted in a quantitative but not context related way. So, it only showed the uncertainty that was inserted in the message. The notion of meaningful context appears in the next section.

2.3.3 Relative Entropy and Mutual Information

Once again, we stress out that the entropy of a random variable is a metric that computes the amount of the average information required to represent the random variable. Now that we have presented entropy we are going to describe the notion of relative entropy.

More specifically, **relative entropy** $D(p||q)$ is a numeric indication of the distance between two distributions. In statistics, the respective formula equals the expected logarithm of the likelihood ratio. The descriptive explanation of this measure is that it shows inefficiency of considering that our data follow distribution q when they actually follow distrubtion is p. Making a correct assumption about the true distribution of a random variable would allow to create a corresponding encoding with average descriptive information $H(p)$. However, if we associate distribution q with the random variable, then we would need on average $H(p) + D(p||q)$ bits of descriptive information. In particular the definition of relative entropy is:

$$D(p||q) = \sum_{x \in \mathcal{X}} p(x) \log \frac{p(x)}{q(x)} \tag{2.36}$$

where for the extreme cases $0 \log \frac{0}{q} = 0$ and $p \log \frac{p}{0} = \inf$ 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, i.e. $p = q$. Still, we cannot regard relative entropy as a true distance between distributions due to the fact that it neither satisfies the triangle inequality, nor it is symmetric. Yet it is convenient to contemplate this measure as a distance measure. At this point we are going to present mutual information, which is our main interest.

Even though conditional entropy 2.32 shows when two random variables are completely independent, it does not sufficiently describe dependency. If $H(Y|X)$ equals to a relatively small value, this either implies that X contains a lot of information regarding Y, or that $H(Y)$ was from the beginning too small. In brief, **mutual information** $I(X;Y)$ computes the amount of information a random variable includes about another random variable, or in terms of entropy it is the decrease of uncertainty in a random variable due to existing knowledge about the other. For example, suppose discrete random variable 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. On the other hand, if we have a third discrete random variable Z denoting the role of another dice, then variables X and Z or Y and Z do not share mutual information. More formally, 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:

$$I(X;Y) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x,y) \log \frac{p(x,y)}{p(x)p(y)} = D(p(x,y)||p(x)p(y)) \tag{2.37}$$

Note that mutual information is symmetric in the arguments, that is $I(X;Y) = I(Y;X)$, but $H(Y) \neq H(X)$ and $H(X|Y) \neq H(Y|X)$. Furthermore, it is a non-negative measure, which yields zero $I(X;Y) = 0$ if and only if random variables X and Y are independent. The above formula 2.37 can be easily used for continuous random variables, by substituting the summation with an integration.

2.3.4 Relationship between Entropy and Mutual Information

Note that we can re-write the definition of mutual information 2.37 as follows:

$$\begin{aligned} I(X;Y) &= \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log\frac{p(x,y)}{p(x)p(y)} && (2.38)\\ &= \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log\frac{p(x|y)}{p(x)} && (2.39)\\ &= -\sum_{x\in\mathcal{X}y\in\mathcal{Y}} p(x,y)\log p(x) + \sum_{x\in\mathcal{X}y\in\mathcal{Y}} p(x,y)\log p(x|y) && (2.40)\\ &= -\sum_{x\in\mathcal{X}} p(x)\log p(x) - (-\sum_{x\in\mathcal{X}y\in\mathcal{Y}} p(x,y)\log p(x|y)) && (2.41)\\ &= H(X) - H(X|Y) && (2.42)\end{aligned}$$

Formula 2.38 shows that mutual information is the decrease of the uncertainty of discrete random variable X given knowledge about the discrete random variable Y. With the use of the property of symmetry:

$$I(X;Y) = H(Y) - H(Y|X) \tag{2.43}$$

Furthermore, equation 2.35 yields the following:

$$I(X;Y) = H(X) + H(Y) - H(X,Y) \tag{2.44}$$

Finally, note that:

$$I(X;X) = H(X) - H(X|X) = H(X) \tag{2.45}$$

Note that mutual information $I(X;Y)$ is the intersection of information in X with the information in Y.

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:

$$H(X_1, ..., X_n) = \sum_{i=1}^{n} H(X_i|X_{i-1}, ..., X_1) \tag{2.46}$$

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:

$$D(p(x,y)||q(x,y)) = D(p(x)||q(x)) + D(p(y|x)||q(y|x)) \tag{2.47}$$

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

$$I(X_1, ..., X_n; Y) = \sum_{i=1}^{n} I(X_i; Y|X_{i-1}, ..., X_1) \tag{2.48}$$

Chapter 3

Related Work

3.1 Support Vector Machines

Research efforts towards developing efficient parallel Support Vector Machines implementations span FPGA, GPGPU and cluster technologies. In this section we present a brief overview of these works.

3.1.1 FPGA

Among the first FPGA-based SVM implementations was the work of Anguita et al. in [62] who implemented the Fibs algorithm. More specifically, they presented a novel algorithm for the SVM Training phase targeted for special-purpose platforms. Recall that the training phase requires solving a constrained quadratic optimization problem (CQP) in order to find the necessary optimal values 2.2.2.2. Furthermore, the proposed algorithm is based on the digital SVM algorithm which they claim has been proved effective for real-world settings. In general, the DSVM algorithm solves the CQP by considering the threshold to be fixed, but this neither follows the general SVM principle, nor is practical for the general class of kernel functions. Thus, in addition to implementing the SVM Training phase on an FPGA, they also extend the initial DSVM version to a more general by adding another step to the algorithm. In particular, initially they also solve the CQP by assuming that the threshold is a priori known. Hence, the inputs of the first step are the kernel matrix Q, a fixed parameter b and an initial value α_0, and the output is an intermediate α' value. This leads to the second step of their algorithm which is to find b^*.

So they substitute the α parameter in the decision function 2.11 which yields the range $[b_{down}, b_{up}]$ where the optimal $b*$ belongs to. Then, an iterative bisection process begins and in each iteration of this process the values of b_{down} and b_{up} are updated, again based on function 2.11. This process continuous until the range is smaller than an error e. From the perspective of hardware, they present an initial architecture of *Fibs* specialized for the RBF-SVM Training phase. Their architecture comprises three main stages load, learn and output and each of these stages has its own controller. More specifically, the loading stage stores the values of the target vector y and the kernel matrix Q, and subsequently the learning stage runs the Fibs algorithm until it finishes. Finally, the output phase outputs the values of $b*$ and a_i*. The most significant computing blocks are the counter blocks required for indexing and counting, the dvsm block which contains the memory required for the Q matrix, as well as the digital components for Fibs, the bias block which comprises the necessary registers and logic to update values b_{down} and b_{up}, and finally the s-blocks which consist of registers that store target values y and logic components to compute the decision value. Regarding their experimental results, the evaluate their performance on the Sonar dataset and on channel equalization with the use of a Xilinx Virtex II FPGA. They mention that they achieve accuracy close to the one yielded by the SMO algorithm, however they do not provided the exact numbers. Furthermore, the maximum number of support vectors found by their architecture was 32, a relatively small number compared to real-world datasets. Thus, the overall conclusions of this work was that they achieved effective resource utilization, but convergence was not compared with the one of SMO and it was only tested for small problems.

A more recent work was presented in [63] where Cadambi et al. proposed an FPGA-based architecture to address the massive parallelism opportunities provided by the SVM Training phase, as well as the requirements to be able to support vast amounts of data. Due to the fact that contemporary datasets do not fit into on-chip caches, high-bandwidth communication between the processor and off-chip memories is needed. Another way to address this problem is with the use of Embedded systems. In this particular work, Cadambi et al. focus on accelerating the standard SMO SVM algorithm with the use of an FPGA platform. However, note that in order to gain acceleration they significantly reduce precision which yields their implementation impractical. Still, they propose an efficient solution for large problems which constituted the basis for our architecture as we were also interested in handling big datasets but with high precision. In brief, they created an FPGA-based co-processor to achieve SVM Training hardware acceleration with low precision arithmetic. The computationally expensive and precision tolerant part of the algorithm is executed on the FPGA, whereas the rest and sequential part is performed by the host CPU. In particular, the host runs the original SMO algorithm and the FPGA co processor computes the kernel functions i.e. dot-products. In each iteration of the optimization algorithm, the training data are inputted in the FPGA with a host DMA and the output is the kernel dot-products. Since SMO is a gradient-descent algorithm it stores both certain initial values, as well as the required updated ones. These values are the α values and the gradients are on the host side, whereas the training instances and support vectors on the FPGA. Regarding the computations performed during the execution of the algorithm, the host produces the next working set, the α updates and the gradients, whereas the FPGA is responsible for implementing the kernel functions. In particular, their infrastructure consists of four vector processor clusters each one containing P chained VPE arrays with N parallel functional units, two data inputs called *DataA* and *DataB* from which the first is stored in an off-chip memory and the second in a cache, an instruction bus and a registered output. The two input streams are in different memories because *DataA* contains the large training dataset which is a big and invariable matrix, while *DataB* is dynamic and small so it is efficient to keep it in cache. Regarding the algorithm, *DataA* is distributed among the different clusters by the bus, whereas *DataB* is inserted to all clusters. Each cluster uses the VPE arrays to pipeline the calculation of dot-products and final outputs are stored in a single output bus. Furthermore, they conceal the FPGA stalling by creating non-blocking functionality. This means that while a chunk is being prepared to be sent to the FPGA, the host side undertakes dot-product computation, and therefore latency is hidden. For their experiments they used an FPGA from the Xilinx Virtex 5 family. They tested their implementation with 4-bit, 8-bit and 16-bit precision and used the RBF SVM Kernel. Furthermore, the software reference was the MiLDE library and the dataset was MNIST containing 60K and 2M data instances. For small numerical precision (4 and 8 bit) they achieve double clocking i.e. each memory element is multiplied with two cache instances in a single clock cycle. Briefly, in the training phase their architecture achieved 18.2x for 4-bit precision, whereas for higher precision they mention that the FPGA is slower. However, they claim that even though their implementation is not as fast as other GPU implementations, still the respective energy per watt that is required is much less.

Another work that combined software with hardware was described in [64]. More specifically, similarly to [63] Pedersen et al. utilized the Sequential Minimal Optimization (SMO) method for the training phase. They also move the time-consuming kernel computation part of the algorithm to the FPGA, but in contrast to [63] they study 32-bit integers using fixed-point arithmetic. Apart from that they use multiple and accumulate processing units to compute the kernel function. Their experimental results, which ran on an Cyclone FPGA showed that for certain test cases the hardware kernel computation required the same execution time as the software kernel computation and for other test cases even less. However, they do not provide any information about the software reference. Also, the fixed point resolution provoked non-tolerable errors in finding all of the support vectors, which respectively affected classification accuracy.

All works of Bouganis et al. [65], [66] and [67] accelerate the classification phase of the SVM algorithm by implementing the Cascade SVM classifier on an FPGA. In order to do so they take advantage of the property of FPGAs to be reconfigurable. However, their main contribution is that they have created an architecture that adapts to heterogeneous datasets, i.e. datasets with different types of attributes, based on their precision requirements. Thus, they have implemented a fully-customized processing unit targeted for the heterogeneous resources of the FPGA. The combination of applying the cascade classification method with custom precision arithmetic units, led to an at least x7 speedup compared to other FPGA and GPU implementations. They attribute this success primarily to the fact that GPUs cannot be configured based on the exact precision requirements, and thereby they always assume floating point precision. More specifically, regarding their design they start by loading the support vectors in an internal memory and the classification dataset in RAM memories between the FPGA board and the host. Then they encode the attributes according to the minimum required precision that is required to represent them. The values of these attributes are normalized to limit the effect of outliers and subsequently they are sorted in descending order based on the required bits so that they can be inputted to precision targeted adders. The size of each adder depends on the minimum precision bit of the decimal part of the two inputs of the adder. Once this is found the tree adder is constructed which outputs the dot-product. They also compute the initial ratio between the DSPs and the LUTs, since they support that by preserving a balance between the two they achieve maximum performance. Thus, they calculate this ratio continuously to guarantee balance. Now, in order to accelerate classification, they follow the cascade scheme which basically assumes the existence of multiple SVM classifiers. These classifiers combine their results to yield the output of the classification decision function. For their experiments they used an Altera's Stratix III FPGA board and their designs achieved operating frequency between 200-250 MHz. Regarding accuracy the FPGA-based cascade classifier reached good accuracy, but still remains an approximate method. In terms of acceleration they compared their implementation to other hardware-based SVM classification applications (FPGA and GPU) and achieved at least an x7 speedup, whereas compared to software implementations they accomplished 2-3 orders of magnitude better performance.

At this point we will briefly present other works that assigned some part of the SVM algorithm to an FPGA platform. However, similarly to works [62], [66] and [67] all of the following works reach approximate solutions. For specific datasets these solutions may be very close to the precise ones, but they still remain sub-optimal. Moreover, the authors of [68] of Khan et al. use the Logarithmic Linear System (LNS) to achieve compression in arithmetic computations, thus efficiently using the hardware resources. Using the LNS is an alternative of fixed point arithmetic to avoid multiplications and divisions. However, there is an overhead to convert the original number to and from the LNS. In addition, they also created an architecture specifically for the linear kernel classification due to the fact that the linear kernel is simply a dot-product. Hence, it is not required to store both the support vectors and the values of α as storing the weight vector is sufficient. Also, in order to produce the decision about an input instance they compute the kernel function using multiply and accumulate (MAC) processing units. Yet, they use as many MAC as the number of attributes. For the experiments they used a Xilinx Spartan 3 FPGA. The accuracy yielded by this application was similar to the one produced by the identical LNS software implementation. Nevertheless, when the word precision increased to 20-bit that caused a prohibitive error and the design could not fit in the FPGA. In [69] Irick et al. present a Gaussian Radial Basis classification implementation in order to achieve real-time extraction of objects from grey-scaled images. In order to perform classification they compute the euclidean norm of the support vectors with the input data. In addition, they perform computations in the Logarithmic Number System (LNS) for the same reasons that the authors of [68] did. The experimental results showed that their architecture yields 88.6% accuracy when the testing dataset is a 30x30 image window with 8-bit arithmetic precision. In particular, in 1 sec they support that they can classify 1,100 30x30 pictures but they do not mention the FPGA platform that they are using. Ramos et al. in [70] describe they implementation of a whole speaker identification system implemented on a special-purpose hardware platform. In particular, they perform binary classification to distinguish whether a speaker is male or female. Their whole algorithm is executed on the FPGA and comprises two phases, the first being feature extraction based on mel-frequency coefficients and the second being the SVM classification phase. In the proposed architecture they consider fixed point numbers which provokes discrepancy between the real and fixed value. In their experiments they used a small FPGA, the Xilinx Spartan 3 platform with operating frequency 50 MHz. Their execution time and the classification accuracy was comparable to the one of an Intel Pentium IV processor running at 1.5GHz but the testing dataset was small with few samples and only 26 attributes. Finally, the work proposed by Llata et al. in [71] also addresses deploying the svm classification phase to a low-cost small FPGA, but they focus on multi-classification and regression. More specifically, they claim that the same hardware can be used for both algorithms. The kernel of the decision function is hardware-friendly, i.e. a not commonly used one. It contains several sv blocks whose number is equal to the number of support vectors. Each sv receives as input a support vector and a sample, and outputs the estimated y target. Regarding their experimental results, they test 8-bit precision images with the use of Cyclone II FPGA. No acceleration comparisons are made, and their work presents a certain error rate which they presume will become zero for a big number of SVs and floating point arithmetic.

3.1.2 Graphical Processing Unit

In this section we present the most significant implementations of SVM Training and classification, implemented with the use of GPUs. Due to the fact that our architecture is deployed on an FPGA we will not describe each work in detail. We will provide however their main contribution and experimental results as Grapical Processing Units (GPUs) have been proven to be more effective for general SVM computations.

To begin with, until recently the fastest GPU-based training and classification approach was proposed by Catanzaro et al. in [72]. More specifically, in their work parallelism derives from creating one thread for each data sample to compute a value that reflects the impact of the optimization step on the optimality conditions of the remaining data instances. For both the training and classification stages they used the Map Reduce model. In particular, in the former case the Map function computes the optimality condition vector based on the Karush-Kuhn-Tucker condition of the SMO algorithm. Then, the Reduce function summarizes these results and derives the final boundaries. Reduction is performed in a tree-structure manner to yield maximum parallelism. Note that for each data point the optimality condition is computed, and this task is undertaken by a dedicated thread. Regarding the classification step, the final decision is also computed based on the Map Reduce model. Results showed very good performance compared to the fastest LIBSVM software, as they reached 32x speedup in training and 154x in testing. For their experiments they used an Nvidia GeForce 8800 GTX graphics processor. However, the overall results of their implementation are not the exact same ones with those of LIBSVM.

Furthermore, in work [73] Carpenter provides a software package called cuSVM that accelerates the training and classification steps of the SVM method. Among their contributions was that they modified the standard SMO decomposition method, by integrating the second-order working set selection heuristic which requires a reduced number of iterations to produce results. In addition, they neither used single-precision floating point arithmetic in order to optimize the results of classification, nor did they use double-precision floating point arithmetic which could be pointless. In fact, they used a mixed precision arithmetic to reach maximum performance and as good accuracy as possible. Moreover, similarly to the previous GPU-based implementations, Carpenter also performs the batch processing kernel computation on the hardware side based on Volkov's highly optimized CUDA matrix multiplication algorithm. Finally, they use kernel caching, i.e. kernel computations that are probable to be repeated are stored in the GPU cache memory. The cuSVM package was executed on an NVIDIA GTX 260 GPU. Experimental results showed that it could achieve 12-73x faster performance for the training phase and 22-172x acceleration for the classificaton phase compared to the LIBSVM package. In terms of classification accuracy they achieved an outcome close to the optimal one.

In [74] Herrero-Lopez et al. describe an multi-class SVM classifier on a GPU. The main contribution of this work besides yielding similar speedups with other fast GPU-based applications, is that it performs multiple cooperative binary PSMO SVM instances at the same time. This approach is called Parallel-Parallel SMO (P2PSMO) and as implied by its name it is based on the PSMO algorithm. Once again the basis of the proposed implementation is the SMO algorithm, due to its popularity and also because it supports data re-usability patterns. Regarding the mapping of their method to the GPU, given P subsets and N binary tasks a grid including PxN blocks is utilized to address the most computationally demanding portion of the SVM Training step. Then, each thread withing the block works only for a single dimension of the block. The vertical dimension denotes specific tasks that are processed by the block, while the horizontal dimension depicts an occurrence of the PSMO algorithm. So, the concurrent interconnected execution of rows results to the eventual P2PSMO method. Their experiments were carried out in an NVIDIA Tesla C1060 GPU card using single-floating point arithmetic and the software reference was LIBSVM. In terms of classification accuracy, they achieved the exact same accuracy as the software benchmark but there were differences in the number of support vectors and the offset value. Furthermore, in terms of performance acceleration they reached x32 improved speed for binary training and x57 for multi-class training.

The recent work of Cotter et al. presented in [75] provides a convenient to use library for kernalized SVM Training on GPUs. Their main contribution compared to previously proposed GPU-based implementations is that they have designed a n algorithm particularly efficient for sparse datasets, which are often encountered in real-world applications. This is achieved with the novel *sparsity clustering* method. Similar to other GPU-based works the time-consuming parallel computation is performed on hardware, whereas the sequential steps are carried out on software. More specifically, the CPU calculates the Gram matrix and solves the constrained optimization problem, while the GPU carries out the kernel matrix computation for a small part of the working set (16 working sets). In order to address sparse datasets, they perform coarse clustering by sparsity pattern with the use of a greedy heuristic so that memory accesses are coalesced. Regarding experimental results they tested their performance on an NVIDIA Tesla C1060 graphics card. Compared to the performance of LIBSVM they achieved an at most x78 acceleration speedup and compared to the fastest GPU implementation an at most x3 acceleration speedup, depending on the dataset.

Moreover, in work [76] Do et al. present an extended parallel version of the fast Least Squares SVM (LS-SVM) method. In general, LS-SVM substitutes the standard SVM optimization inequality constraints with equalities in least squares error, and thereby the training task only requires solving a system of linear equations instead of a quadratic program. This transformation allows having very short training time. The proposed extension of the original LS-SVM method follows two directions. The first is towards creating an incremental approach that scales up to allow large dataset classification (billions of data instances) by loading only subsets of data in memory and updating the solutions of the growing training set. The incremental version of the LS-SVM approach yielded the exact same accuracy as the original one. The second direction focuses on creating a parallel extension of the incremental algorithm. They split the dataset into small blocks. For each incremental step, a data block is loaded into the CPU memory. A data transfer task copies this block from CPU to GPU memory and then GPU computes all matrix multiplications in parallel. The results are transferred back to the CPU memory where the linear equation system is solved. The experiments were executed on an NVIDIA GeForce 8800 GTX GPU and the results showed that the parallel incremental implementation is up to 70 times faster than the respective non-parallel one.

In addition to the aforementioned works, others have also been proposed which present similar or worse results. Some of these are the approaches are described in [77], [78], [79].

3.1.3 Multi-Core

This section presents multi-core approaches for the Support Vector Machines algorithm. Due to the fact that this thesis studies the parallelism offered by the SVM method from a hardware-based perspective we will not be providing detailed descriptions.

To begin with, a broadly used parallel SVM approach is the Cascade SVM [80] proposed by Graf et al. The novelty of this approach is that it considers SVMs as filters in the following way. It initializes the problem with a number of independent, optimization sub-problems and combines the partial results in further stages in an hierarchical manner. In addition to a description of their approach, they provide a formal proof of convergence. More specifically, in the case of binary Cascade SVM the sets of support vectors produced by two SVMs are combined and the optimization proceeds by finding the support vectors in each of the combined subsets. This continues until only one set of vectors is left. If the global optimum has to be reached, the result of the last layer is fed back into the first layer. Each of the SVMs in the first layer receives all the support vectors of the last layer as inputs and tests its fraction of the input vectors, if any of them have to be incorporated into the optimization. If this is not the case for all SVMs of the input layer, the Cascade has converged to the global optimum, otherwise it proceeds with another pass through the network. The experimental results were tested on both a single processor and a cluster of processors. A good indication of the inherent efficiency of the proposed method is obtained by counting the number of kernel evaluations required for one pass. For example, a single pass requires only about 30% as many kernel evaluations as a single SVM for 100,000 training instances. Thus, a simulation on less than 8 processors can produce a speed-up of 10x or more, depending on the available memory size.

In [81] Cao et al. propose a parallel version of SMO for training SVM, developed with the message passing interface (MPI). The main idea is that they divide the total training dataset into equal subsets each of which is assigned to a CPU processor. Then, the CPU processors perform independent tasks in parallel, which in particular are updating a different subset of training data patterns, and calculating b_up, b_low as well as the Duality Gap at each step. Therefore, assuming that the sequential SMO algorithm requires t time to perform the same task, the time reduces approximately to $\frac{t}{p}$, where p depicts the number of available processors. Their experiments were conducted on 1.3 GHz processors and the accuracy achieved was the same as the one yielded by the sequential SMO algorithm. For this specific setting their approach reached approximately x23 speedup when 32 processors where used, but efficiency decreased with the increase of the number of processors. Thus, they concluded the proposed parallel version of SMO is useful for large datasets.

One

Moreover, in [82] Chang et al. proposed the Parallel SVM implementation (PSVM) which is available open source. The first step of the algorithm is based on the parallel row-based ICF (PICF) algorithm, which loads training instances onto parallel machines and performs factorization simultaneously on these machines. Then, once n training data instances are distributed on m machines and the size of the kernel matrix has been reduced via factorization, the Interior-Point method is solved on parallel machines concurrently. The experiments were carried out on 500 machines with each of these machines having a CPU faster than 2 GHz and memory bigger than 4 GBytes. Results showed that the accuracy of PSVM can approach the one achieved by LIBSVM for higher ranks of the ICF matrix, but is still worse than other approximate solutions. Nevertheless, in terms of acceleration when PSVM is executed on 500 machines, it can reach up to x170 speedup for certain datasets.

A more recent work presented in 2011 [83] by Zhao et al. utilizes the Map-Reduce parallel framework and applies decomposition to configure a parallel version of SVM. Their goal is to particularize the general Map-Reduce framework for Machine Learning methods proposed in [84] for SVM. That is achieved as follows. The variables in the working set B and the computation of the appropriate function f are partitioned into parts in the same manner. Each of these parts is assigned to a map process, and subsequently each map process calculates the value of Q_{ij}. The distributed values yield the optimal value f^* in the reduce phase. In addition, they cache the kernel elements as much as possible which significantly improves the method's performance. Regarding the experimental setting, they used two computers with different hardware architectures to evaluate different hardware effects. The first had 4 cores with a shared L2 cache and memory, whereas the second was a dual-processor system with 4 cores in each processor. Furthermore, results showed that the proposed implementation was at most 4 times and at least 2 times faster than LIBSVM, while it always outperformed a similar MPI-based implementation presented in [85]. In terms of accuracy it yielded results quite similar to those of LIBSVM, with the difference lying in the number of support vectors. However, results also showed that the specific approach worked efficiently for smaller datasets due to locality and overheads of reduction.

3.2 Mutual Information

Unlike SVM, few hardware-based approaches have been proposed for Mutual Information which are presented in this section

3.2.1 FPGA

The first FPGA-based approach for Mutual Information computation was presented in [86] by Castro-Pareja and Shekhar. The proposed architecture was called FAIR-II and achieved hardware acceleration of mutual information-based image registration. More specifically, they aim at real-time computation of image registration but without the use of supercomputers. Furthermore, their architecture consists of two discrete steps both of which are carried out on hardware. In the first step hardware creates the mutual and individual histograms based on an algorithm that transforms the floating image's coordinates to the respective reference ones with the use of Partial volume interpolation. Subsequently, during the second part of their approach the partial joint histogram values are sent to the accumulator and the total mutual information calculation is derived. Their system was tested on an Altera Stratix EP1S40 FPGA and it was able to process 50 million reference image voxels per second. Compared to an optimized software implementation on a 3.2-GHz Xeon workstation with 1 GB of 266 MHz DDRAM FAIR-II delivered x30 speedup for linear registration and x100 speedup for elastic registration.

Furthermore, in [87] Shao et al. propose the first reconfigurable computing solution to accelerate transfer entropy computation. Even though we are not studying transfer entropy in this thesis, it is worth describing this architecture as transfer entropy is a measure quite similar to mutual information and also the Maxeler tool is being used. The main difference between mutual information and transfer entropy is that the second captures the amount of uncertainty reduced in a time series', Y, future values by knowing the past values of another time series, X, given the past values of Y. During the pre-processing phase of the transfer entropy calculation and based on Laplace's rule of succession, they associate each possible pattern between two time series with an imaginary count to obtain the necessary probability estimates. Note that the aforementioned process is carried out on software. Regarding their hardware implementation they achieve optimized memory allocation by mapping small and medium-sized tables to the on-chip BRAM and by streaming large tables to the FPGA at runtime. Furthermore, they utilize the bit-width narrowing technique to guarantee efficient memory allocation and to reduce I/O overhead. The computation of the transfer entropies of both X to Y and Y to X are carried out on the FPGA in parallel based on the available resources. The experiments were conducted on a Xilinx Virtex-6 FPGA and was compared to a single and 6-core Xeon CPU. Their implementation achieved x100 and x18 speedup respectively.

To the best of our knowledge, no other FPGA-based approach has been proposed for the computation of Mutual Information.

3.2.2 GPU

To begin with, in 2007 Shams and Barnes presented an efficient method for mutual information computation between images for NVIDIA compatible devices [88]. The execution flow of their approach is as follows. First, as a pre-processing step they transform the 2D joint histogram calculation to a 1D code. Then, the probability mass function calculation is distributed to L thread blocks each with N threads. Each block maintains 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 multithreaded reduction function. The experimental results of the aforementioned implementation were carried out on an NVIDIA 8800 GTX platform. Moreover, results indicated that in the case of a 3D image with approximately $7x10^6$ voxels and 256 threads the GPU-based registration was around 25 times more efficient.

Another approach was presented in [89] by Lin and Medioni, who proposed a GPU implementation to compute both mutual information and its derivatives. More specifically, in order to estimate the probability density for the mutual information computation they use the Parzen Window method, which directly utilizes the samples drawn from an unknown distribution and applies the Gaussian Mixture model to estimate the probability's density. Furthermore, they address the image registration problem by estimating the transformation T that best aligns two images. In order to do so they maximize mutual information by approximating its derivative with respect to T. The opportunity for parallelism is offered by the inner summations in the required equations since the statistics associated with each element are independent from the ones of the others, as well as parallel shared memory access. The experiments of the aforementioned work were conducted 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.

In addition to the aforementioned implementations, other works have also been proposed for hardware-based Mutual Information computation [90], [91]. However, apart from [87], all these approaches focus on accelerating specifically the image registration problem.

Chapter 4

Implementation

The two following algorithms, Support Vector Machines and Mutual Information were profiled with respect to computational characteristics, available parallelism, Input/Output (I/O) requirements, and suitability for hardware implementation. This leads to the optimization of the computationally intensive part, or computationally equivalent mathematical transformations for more hardware parallelizable versions of the algorithm. SVM proved to be less suitable for translation to FPGA whereas Mutual Information proved to be highly suitable.

4.1 Support Vector Machine

Data Classification aims at categorizing data objects into distinct classes with the use of labels. In particular, statistical classification receives new data inputs and identifies their respective classes. An example would be assigning a post derived from the social media into ”relevant” or ”irrelevant”. It is confirmed by the bibliography that the Support Vector Machines (SVM) algorithm is a commonly used approach for data classification and is also highly parallelizable 3.1. Furthermore, traditionally SVM have been used for binary classification scenarios, but it can be used for multiclass cases as well, with an extension of the binary case. In our work, we built binary classifiers using the SVM methods from the LIBSVM package [49]. We integrated parallelizable LIBSVM functions to our hardware implementation and used the LIBSVM tool as a point of reference for SVM translation to hardware. Note that quadratic programming optimization problems are computationally expensive. In cases where the datasets are high-dimensional and voluminous, such as in text classification, the kernel and inner product computations require a massive number of matrix-vector operations. On hardware however, these operations can be performed in parallel and produce the same outcomes much faster.

However, it is of essential importance to study the properties of an algorithm from the hardware designer's perspective before implementing it to a special-purpose platform. A thorough analysis shows the acceleration opportunities and bottlenecks provided by each of the algorithms.

4.1.1 Modeling for Hardware

It is common practice for hardware designers to analyze the hardware-friendly properties of an algorithm, prior to mapping it to hardware. More specifically, from a designer's point of view the most significant characteristics are the inputs and outputs of the algorithm, performance issues and the basic data structures and operations that constitute the respective algorithm. In addition, the communication overhead for the synchronization of the portions of the algorithm which run in specialized hardware with the aspects that run in software has to be considered, as excessive fragmentation may lead to poor performance. This analysis is performed in the specific section.

4.1.1.1 Inputs and Outputs

The SVM Training algorithm receives as input a two dimensional structure and outputs the SVM model. More specifically, the input of the LIBSVM library is a file containing training data, with a specific format. The goal of our implementation was to apply the classification phase of SVM on streaming data, if the algorithm yielded significant acceleration. However, the training phase which precedes classification was applied on training datasets. In these datasets each row represents a data instance and each column denotes a feature. The only exception is the first column of each data instance that depicts the target class of the data instance defined as the label of the data instance. In binary classification, this label can value 1 or -1, whereas in multi-class classification the number of possible labels depends on the number of classes. A detailed description of the input format of the training dataset was provided in 2.2.4.1 as it is the same with the one required by LIBSVM. In brief the input file contains *a:b* expressions, where *a* denotes the number of the feature and *b* represents the value of the respective feature.

Note that it is not necessary for all data instances to share the same features. Absence of a feature implies that it has zero-value, thus allowing an efficient representation of sparse datasets.

The output of the SVM Training phase is a file that contains the SVM model. The SVM model comprises certain variables computed by the SVM Training algorithm which are necessary to determine the final classifier. More specifically, it contains the optimal objective value of the dual SVM problem, the bias term in the decision function, as well as the number of support vectors. In addition, following the aforementioned parameters are the support vectors that are ordered based on their target labels. Specifically for the binary classification case, the support vectors belonging to the first class are grouped first and those corresponding to the second class, follow.

4.1.1.2 Algorithm Profiling

In this section we present critical points of the LIBSVM software code that indicate hardware opportunities. In order to do so, we performed profiling of the source code using the Linux GNU GCC profiling tool (gprof) so as to detect potential parallelism.

Quadratic programming optimization problems, such as the SVM classification algorithm are expensive. In cases where the data sets are high-dimensional and large, the kernel and dot-product computations require a massive number of matrix-vector operations. This can be observed in table 4.1 since all functions that appear assume matrix-vector operations. A brief description of each function is given in the specific table and a more detailed one follows.

Dot-Product function (1)

This function receives as input two equal-length vectors x and y and outputs a single number which denotes the dot-product of the two vectors. A dot-product is defined as the sum of the products of the corresponding entries of the two sequences of numbers. Moreover, in the SVM algorithm these vectors x and y represent data instances or in other words rows of the training dataset file. In software, computing the dot-product of two vectors of length l implies an lxl number of computations and produces l dot-products. Nevertheless computing simply the product of two entries is independent of computing the product of two other entries. Therefore the multiplication task could be performed in parallel and as a result we would have

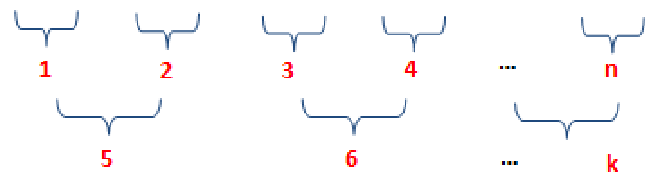
Description	**Time Percentage**	**LIBSVM Function**
Computes the dot-product between two data instances, i.e. $x_i * x_j$	72%	*dot_product()* **(1)**
Computes the kernel function, i.e. $K(x_i, x_j)$	8%	*kernel_computation()* **(2)**
Finds sub-problem to be minimized in each iteration	8%	*select_working_set()* **(3)**
Computes part of the dual lagrangian equation $\sum_{i=1}^{l}\sum_{j=1}^{l} y_i y_j k(x_i, x_j)$	6%	*get_Q()* **(4)**
Solves the optimization problem.	4%	*solve()* **(5)**

Table 4.1: SVM profiling analysis

l multiplications performed concurrently. Once we have produced the products we can add them in pairs and in parallel. Note that the computation of addition is independent of computing another addition and this allows parallelization. In particular, the sum of products 1 and 2, the sum of products 3 and 4 and so on can be performed concurrently. For example, in our l length vectors $\frac{l}{2}$ addition pairs are formed, so that $\frac{l}{2}$ additions are produced simultaneously. Then, the outcomes that are produced can also be added in pairs and at the same time and this procedure continues until we are left with a single number, which is the final dot-product outcome.

An illustration of the aforementioned dot-product computation follows. Given two vectors A and B with length n $A = [\ A_1,\ A_2,\ ...\ ,\ A_n]$ and B = $[B_1,\ B_2,\ ...\ ,B_n]$ a parallel version of the dot-product is defined as:

$$\boldsymbol{A \cdot B = \sum_{i=1}^{n} A_i B_i = A_1B_1 + A_2B_2 + A_3B_3 + A_4B_4 + \cdots + A_nB_n}$$



The parallel version of the above dot-product computes in parallel all products from *1* to n. Once these are produced, the sum of 1 and 2 (5) is calculated in parallel with the sum of 3 and 4 **(6)** and so on.

Furthermore, we know that in the LIBSVM software implementation every sequential step, i.e. loop, of the optimization solver requires the computation of the dot-product between two selected data instances (working set) with the rest of the data instances of the input file. It is worth mentioning that producing in parallel the maximum allowed number of possible dot-products is much more efficient than only computing a single dot-product.

Kernel Computation function (2)
The dual formulation of the SVM optimization problem introduces the notion of the kernel 2.2.3.1. Recall that in the case of linearly separable data, the kernel of two vectors is equivalent to the dot-product of these two vectors, whereas for non-linearly separable data the kernel can be selected from a variety of kernel functions 2.2.3.1. Note that the efficiency of a kernel function is determined by the nature of the training and tested input data, as well as other factors such as speed and accuracy. That said, we selected the radial basis function which according to the equivalent software implementation of this function is computed based on the following formula:

$$k(x_i, x_j) = exp(-\gamma \left\| x_i - x_j \right\|^2), \gamma > 0 \tag{4.1}$$

which in terms of the LIBSVM functions in table 4.1

$$k(x_i, x_j) = exp(-\gamma(dot\_product(x_i, x_i) + dot\_product(x_j, x_j) - 2dot\_product(x_i, x_j)), \gamma > 0 \tag{4.2}$$

In the above formula γ denotes a constant selected by the user and function $dot\_product()$ yields the dot-product of the given vectors. What can be derived from the above formula is that the three parallel dot-products can be computed in parallel too given that we know indicators i and j. However, the result of $dot(x_i, x_i)$ has constant value during the dot-product computation of data instance i with the rest of the data instances, and thereby we do not need to compute this dot-product every time. The same does not apply for indicator j since it receives all values from 0 up to the last data instances for a specific value i.

- **Select Working Set function (3):**
The Q matrix of the dual optimization problem which is defined in LIBSVM as:

$$\sum_{i=1}^{l}\sum_{j=1}^{l} y_i y_j k(x_i, x_j) \tag{4.3}$$

is usually dense and too big to be stored. Therefore, decomposition methods have been proposed to effectively process this matrix. In general, optimization methods update the whole vector a in each iteration. However, with the use of decomposition methods only a subset is processed and modified. This subset is called a working set and allows handling sub-problems in each iteration instead of the whole vector. This LIBSVM approach is based on the sequential minimal optimization algorithm [92] which considers as a sub-set two Lagrange multipliers a_i, a_j in each optimization step.

- **Get Q function (4):**
This function produces the outcome Q 4.3 a matrix with length size equal to the total number of data instances. Note that within a step of the optimization solver we compute the formula presented in the description of the kernel computation. During this step, index i remains constant and j receives all values from 0 up to the number of data instances. Thus, the kernel computation is performed as many times as the number of rows (data instances) of the input file. However, in the computation of the *get_Q*() function we also add the labels of the respective vectors based on formula 4.3.

- **Solve function (5):**
This function is called once during the execution of the program. As indicated by its name, it outputs the complete solution of the SVM optimization problem. Thus, it contains the entire process of the algorithm within which consists of the arbitrary number of steps of the dual formulation optimization problem. Due to the fact that in each step, certain variables of the algorithm are updated and these updated values affect further computations, we cannot avoid leaving the execution of the optimization problem to the host.

4.1.1.3 Important Data Structures

This section describes all the important data structures of the SVM Training phase, which are required by our hardware-based architecture. As mentioned in the beginning of the SVM modeling section 4.1.1.1 and the background of LIBSVM 2.2.4.1, the LIBSVM software receives as input a file that follows a specific format. The information contained in this file is copied to an appropriate data structure so as to be able to utilize the given information. Recall that it is not necessary for each data instance to have all its respective features equal to a non-zero value. In other words, this representation illustrates how sparse a dataset is. LIBSVM transforms the input information into a compact data structure that only contains the useful information $<index_i>$: $<value_i>$ where $value_i$ is non-zero. More specifically, each of the $<index_i>$: $<value_i>$ pairs is by itself an individual data structure that contains an integer number to denote the identification number of the feature and a number of type double that corresponds to the respective value of the same feature. In order to represent the complete set of data instances another data structure contains a list of $<index_i>$: $<value_i>$ pairs which belong to the inputs. An input instance is separated from the next by assigning the value -1 to an index. In addition, the same data structure contains a list of target labels, one for each training data instance and the number of data instances in the file. The data structure that contains the complete dataset is essential to the algorithm, as all the important functions of the implementation need it to produce outcomes.

Note that the aforementioned data structure is two-dimensional NxM. The first dimension N remains static during the training phase as it denotes the number of input data instances. However, the second dimension M is dynamic, since it depends on the number of non-zero features that correspond to as specific instance. Thus the value of M ranges from zero to the maximum feature index.

4.1.1.4 Performance Opportunities and Considerations

So far we have presented the most significant parallelization opportunities offered by the SVM Training phase, and subsequently the SVM classification phase as in both stages the time-consuming process is the matrix-vector computations. However, besides parallelization, we also need to take into account several other factors that can affect our design, introduced by the nature of the technology we are using and the software reference we are comparing our implementation to. Since we have described the Maxeler technology platform in 2.1, the reference software LIBSVM in 2.2.4 and the hardware analysis in 4.1.1 we are ready to present several critical points that were taken into account when designing the SVM Training phase hardware implementation.

4.1.2 Training SVM

The Support Vector Machines Training implementation of LIBSVM was thoroughly described in section 2.2.4. In brief, it is based on the sequential minimal optimization algorithm introduced by Platt in [92]. SMO is a gradient descent method which implies that when the difference in gradients from successive iterations is small enough, the algorithm converges. The basic principle of SMO is that a single pair of Lagrange multipliers a_i and a_j is being optimized at a time. Due to the fact that computing a_i and a_j is not the main focus of this thesis, for a detailed description of this method we refer the reader to woks [92] and [49]. Yet, recall that for each x_i and x_j selected, the gradient computation requires the calculation of kernel functions $K(x_i, x_t)$ and $K(x_j, x_t)$, for all t. Hence, it is clear that the dot-product of data instances x_i and x_j with the rest of the training samples should be computed in each step of the optimization process. Note that LIBSVM uses 64-bit precision arithmetic, and therefore the hardware processing was also performed on 64-bit precision data instances to avoid accuracy loss. Furthermore, once again we stress out that successive iterations of the LIBSVM training implementation cannot be parallelized due to the fact that the updated working set is based on the gradients calculated in the previous iteration. Thus, the focus of our SVM Training approach is to evaluate whether we can implement a faster hardware-based kernel function using Maxeler than the LIBSVM software-based kernel computation written in C++.

In the following sections, we present our dataflow FPGA-based co-processor for training Support Vector Machines. Our system is designed to efficiently exploit available FPGA logic resources, to approach the theoretical bandwidth of the tool, to optimize memory allocation and to yield the exact same accuracy as the LIBSVM software for dataflow SVM Training computing. These are achieved by optimizing memory allocation, by utilizing to the maximum possible bandwidth and by performing appropriate data management. Recall that we aim at training massive and dense dataflow datasets, to configure the optimal classifier; a problem that has not been yet addressed for FPGA-based designs due to the problem complexity.

4.1.3 First Hardware Architecture

This section describes our first attempt to accelerate the kernel computation of the LIBSVM SMO implementation with the use of the Maxeler platform. It allowed us to get acquainted with the tool, comprehend its benefits and drawbacks, as well as to create the basis for the further improved architectures.

4.1.3.1 CPU and FPGA Integrated System

This section presents how we partitioned the complete LIBSVM SMO algorithm between the CPU host and the FPGA platform. Restricting the whole execution of the algorithm exclusively to either of the two does not allow splitting the parallel from the sequential part of SMO. Figure 4.1 illustrates an overview of our system. The CPU (LIBSVM) host depicts the training SMO-based software which is written in C and C++. In brief, the host performs the sequential part of the optimization, and thereby executes step-by-step (iteration per iteration) the sequential minimization optimization method. For further details we refer the reader to sections 2.2.4 and 4.1.1.

At this point we will briefly describe the high-level view of our first architecture. More specifically, in each iteration CPU provides the hardware platform with the selected working set data instances i and j. In this first attempt we did not define these data instances (i and j) as streams, but we declared each feature of the data instance as an individual scalar factor (input number). Hence, if the maximum number of features in a dataset is n, then n scalar factors are sent from the host to the DFE during the whole hardware processing. The DFE consists of the Manager and the Kernel as described in 2.1.2. Recall that the Manager provides an interface for configuring connectivity between Kernels and I/O, while identifying possible violations. Furthermore, it is clear that the host and the FPGA should interact in order to exchange information. The first should provide all necessary data in order for the latter to carry out the appropriate computations, whereas the latter should return the final outcome. The FPGA co-processor performs the kernel function computations. In order to do so the training data instances are fed from the host to the Kernel. More specifically, a single data element (feature of an instance) is

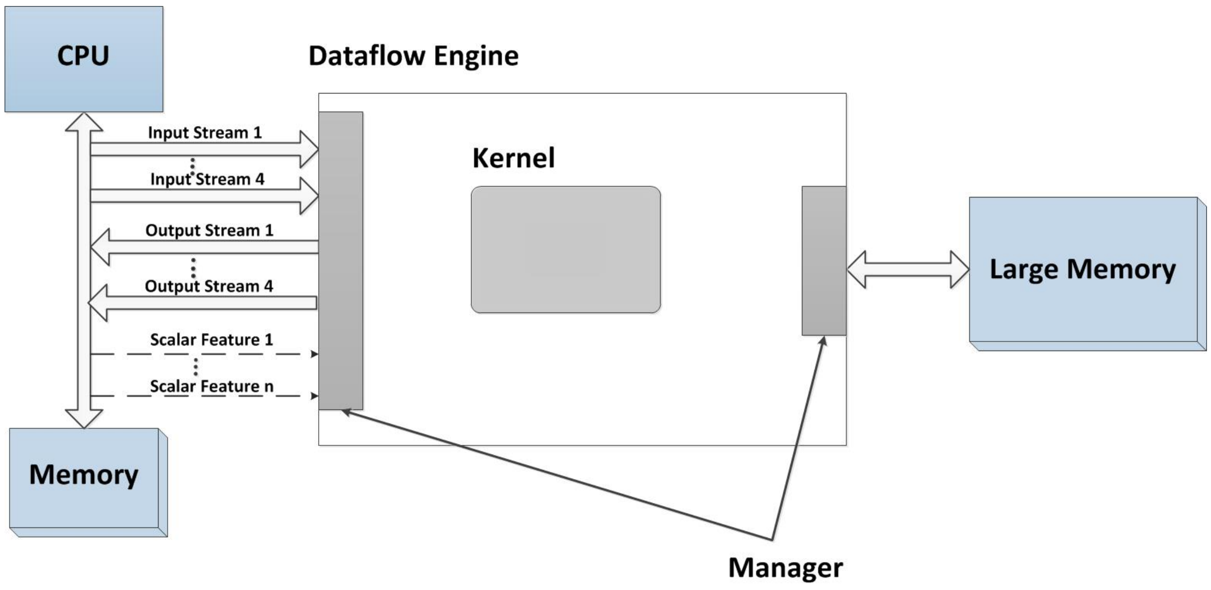


Figure 4.1: First SVM Training Top level Architecture

inserted in the FPGA per time instance. Thus, if the training dataset consists of NxM data instances, where N is the number of samples and M shows the maximum number of features, then the Kernel will require NxM time instances to process all data. Finally, the Kernel produces parts of the radial-basis function (RBF), which are suitable for hardware implementation that are returned to the host in each step. Note that we implemented the RBF function due to its popularity and efficiency.

4.1.3.2 Problem Partitioning

Given that our main goal was to accelerate the kernel function computation, we carried out the necessary problem partitioning. Recall that the input of the training algorithm is a dataset which comprises several data instances and whose size ranges from a few Megabytes to several Gigabytes. Furthermore, this dataset is transformed into a two-dimensional data structure with dimensions mxn, where m denotes the number of data instances and n shows the maximum number of features. In each step of the SMO problem 2*m dot-products are produced from the multiplication of the mxn array with two 1xn vectors, the first corresponding to the data instance i and the second to the data instance j (working set). This kernel computation architecture separately produces the total number of dot-products for the two aforementioned data samples.

Due to the fact that producing one dot-product is completely independent from producing another, we divided the mxn dataset into chunks as shown in Figure 4.2. Each of these chunks is processed in parallel with the others.

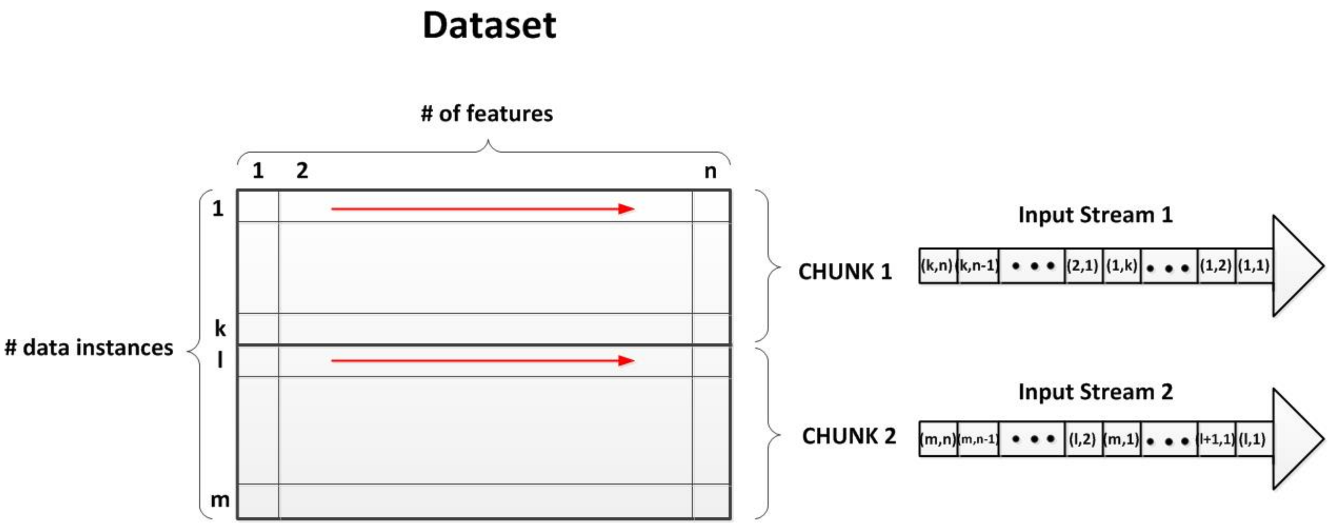

Dataset
# of features
1
2
n
1
k
l
m
# data instances
CHUNK 1
CHUNK 2
Input Stream 1
(k,n) (k,n-1) • • • (2,1) (1,k) • • • (1,2) (1,1)
Input Stream 2
(m,n) (m,n-1) • • • (l,2) (m,1) • • • (l+1,1) (l,1)

Figure 4.2: Initial Problem Partitioning

However prior to sending a chunk to the DFE, pre-processing occurs as the Maxeler platform only allows the exchange of streams of one-dimensional data or constant numbers. Therefore, the two-dimensional matrix is divided to f one-dimensional vectors all of equal size 1x$k * n$, where f denotes the number of chunks, k shows the number of data instances that correspond to a chunk and n is the maximum feature identification number. Note that Maxeler does not allow inserting one-dimensional arrays of different size, and therefore dataset mxn should be equally divided. In our first implementation the chunks equaled to 4, for reasons that we will discuss in the following sections.

Moreover, we exploited the parallelization opportunities offered by the dot-product computation. In particular, calculating the dot-product between two 1xn vectors requires $\frac{n}{2}$ multiplications and $\frac{n}{2}$ additions. As shown in Figure 4.4 all multiplications are simultaneously computed and the results are inserted into a balanced adder tree. Then, the adder tree produces the final dot-product outcome in $\log n/2$ stages.

Finally, a single RBF kernel function does not only require the result of the dot-product of a data instance with another, but also the dot-product of each data instance with itself. These separate dot-products are also produced in parallel to reduce the total kernel function computation time from t approximately to $\frac{t}{3}$. Yet, this is not exactly the case as we will see later on.

4.1.3.3 Data Movement on Host Side

Prior to any FPGA-based computations the involved data structures should be created and initialized. The Maxeler platform interface only allows one-dimensional vectors which are defined as streams. Furthermore, each hardware call may not necessary include input and output streams, but the number of ticks (similar to clock cycles) is mandatory to be defined in advance. However, in order for hardware to yield an output the number of ticks should be completely consistent with the input number of elements. Thus, if the declared number of ticks is N then each input and output vector should contain exactly N elements. Figure 4.1 shows how data move from software to hardware and vice versa.

More specifically, in our initial attempt the input and output number of streams was 4 and 4 respectively according to the number of chunks we created 4.1.3.2. We selected the specific number due to the fact that processing 4 streams concurrently was the maximum number we could reach given the available amount of hardware logic resources. As shown in 4.2 each stream comprises $\frac{M}{4}$ elements which follow one another. Note that the Maxeler platform only allows stream vectors whose size is a multiple of 16.

The 4 input streams contain the entire training data set, whereas the 4 output streams consist of the output of the kernel. The former does not require any further analysis as we have already described the inputs in 2.2.4.1. Regarding the latter however, the output streams do not contain the final outcome of the Q function 4.3 which was our initial goal. The reasons that led to this decision were two. The first was due to the fact that the dot-products of elements i and j with themselves remain constant during the hardware processing, and therefore their computation is only performed once on software. The second derives from the exponent involved in the RBF kernel function, since the exponent is a computationally expensive function and precise accuracy is the main goal of our implementation. Thus, the output streams include the outcomes of:

$$k(x_i, x_j) = dot\_product(x_j, x_j) - 2dot\_product(x_i, x_j) \tag{4.4}$$

and the rest of the Q function 4.2 computation is carried out by the host processor. This allows us to dedicate all hardware resources to the pipeline dot-product calculations.

Finally, the i and j instances of the working set are not inserted in the DFEs as vectors but as scalars, i.e. constant values. We decided our first architecture to follow this direction because at first we aimed at testing datasets with small feature spaces. On one hand, replacing streams with scalars values is more simple for a sketch implementation as it does not require any synchronization or alignment, and also is faster since no initialization and setup time overheads are in need. On the other hand however, the number of possible features, i.e. scalars, is limited by the number of available registers, and in addition each time this number changes (usually for each test dataset) several software and hardware modifications should occur. Note that all input values are doubles, i.e. 8-byte elements.

4.1.3.4 Dataflow Kernel Computation on Hardware Side

Figure 4.1 does not only show the movement of data from software to hardware and back, but also a top level architecture of our system. The system begins with initializing the streams, scalar factors and corresponding clock counters (number of ticks) as described in 4.2.2.3 on the host side. Next, the streaming values are passed to the reconfigurable hardware one-by-one per tick to allow the kernel computation. The scalar inputs are stored in registers on the FPGA, and thereby are always visible to the design. The computed corresponding results return back to the CPU for the final Q function calculation. We stress out the notion and impact of data flow movement. Data flow means that the input streams are pipelined into the hardware, which implies that each clock cycle a single element of a data instance is processed per stream. Hence, in our case the 4 input streams that are utilized in each clock cycle lead to 4 elements being processed concurrently.

Our Kernel architecture is shown in Figure 4.4 with control logic omitted. All inputs are inserted into the *Kernel* module which is responsible for sending each one of the inputs to the corresponding *Dot-Product* module. The *Dot Product* module denotes a dot-product function. The total number of *Dot-Product* modules is 8 since for each chunk, two dot-products need to be computed based on 4.4, and thereby a total of 8 dot-products can be computed concurrently. In addition, according to the aforementioned formula each stream is inserted in two *Dot-Product* modules out of the 8, whereas all scalars are inputted in 4 of these modules according to the same formula.

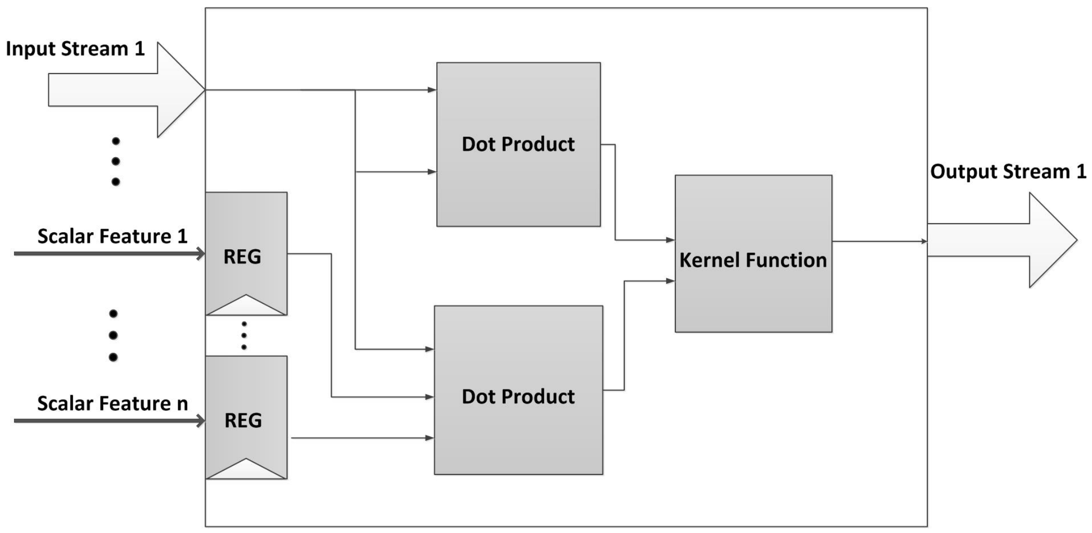


Figure 4.3: First DFE Kernel Computation Hardware Core

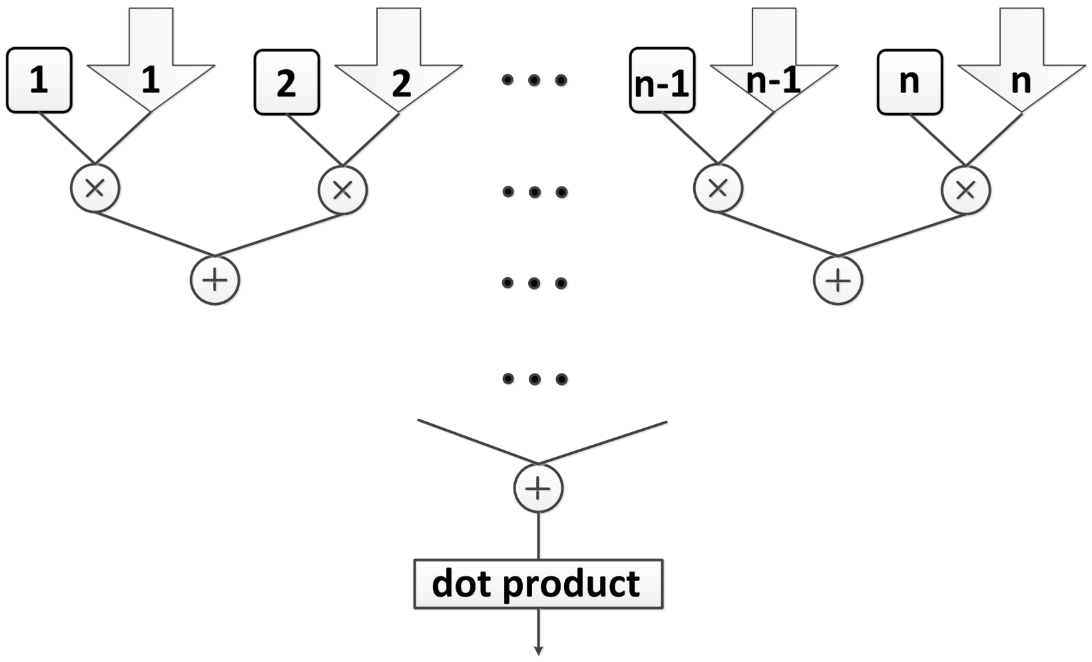


Figure 4.4: First Dot-Product Hardware Core

The core of the *Dot-Product* module is shown in Figure 4.4 module.

This figure illustrates the datapath of the kernel. In general, our first hardware system is quite similar to the respective parallel dot-product version described in ??.

The boxes in Figure $(X_1, X_2, ..., X_{N-1}, X_N)$ stand for different instances of the stream X, whereas the boxes $(I_1, I_2, ..., I_{N-1}, I_N)$ show scalars, where N is the feature space, i.e. the maximum number of features. It is clear that for N features $\frac{N}{2}$ multipliers are needed for a single *Dot-Product* module. Thus, for the total number of *Dot-Product* modules $8 * \frac{N}{2}$ multipliers are required. The same applies for accumulations, as each *Dot-Product* module uses $\frac{N}{2}$ summations, and thereby $8 * \frac{N}{2}$ accumulators should exist. The number of multiplications and accumulations is better presented in the following real dataset example. The *gisette_scale* dataset that was presented in the NIPS 2003 Feature Selection Challenge [93] is a 5M-dimensional problem with 6M training vectors. Thus, each optimization step requires the computation of 6M dot-products for the selected instance i and 6M dot-products instance j respectively. In computational terms this is equivalent to $30 * 10^{12}$ multiplications and accumulations for both samples i and j. Also, note that the SMO typically converges after thousands of iterations which leads to a massive number of multiplications and accumulations, thus indicating that the kernel computation is the most computationally expensive part of the algorithm.

On each cycle (kernel tick), $4+N$ elements are sent from CPU to FPGA, a number which is equivalent to the number of streams sent plus the number of scalars. These elements feed 8 pipelines. In order to navigate the streams of data we used the stream offset property provided by Maxeler. A core concept of dataflow computing is operating on windows of input streams. Maxeler holds a defined data window on an on-chip memory on the DFE, thus allowing to access data elements within a stream relative to the current location (input instance of current clock cycle). The distance from the largest to the smallest offset forms the window of data that is hold on the engine. More specifically, in the dot-product pipeline the window size equals the number of features to be able to correctly compute the result. Yet, imagine the data flowing in and out of the DFE. Let us consider in Figure **??** that data instance X_1 is the current instance, X_2 is the instance with offset -1, X_3 is the instance with offset -2 and so on, until the last element of the window X_N with offset $-(N-1)$. On each tick, the data in the stream moves through the window. Thus, in the immediate next tick instance X_2 will become the reference current instance, and all following elements will have the same offset as before added by 1. Note that element X_1 has exited the window and that a new element X_{N+1} will be the last element of the window. Recall that stream X contains subsequent data instances of the training dataset. Hence, if we assume that in clock cycle k all N elements of a data instance are in the window, then in N cycles the window will contain the next entire data instance. All results produced in between do not have any meaning, but in the first architecture we could not overlap this time with useful processing time. Consequently, a useful kernel computation is produced every N cycles where, N shows the feature space of the training set. Note that the pipeline still produces one result per cycle, but it takes a number of ticks for the result of a given input to propagate to the output, which is the latency of the pipeline.

4.1.3.5 Memory Allocation

Optimized memory allocation is always a target of hardware designers. In particular, the Maxeler platform used in our implementation offers a DRAM off-chip memory bank of 48 gigabytes per DFE and an on-chip Static RAM that holds some megabytes. An essential part of dataflow design implementations is to choreograph the data motion to allow the reuse of data while it is on the chip and limit unnecessary movement of data in and out of the chip.

The formula of the RBF kernel function is shown in 4.1, with the respective analytic expression being 4.2. In order to compute the dot-product of i with all other data samples a 2-level nested loop is required as MxN dot-products are produced, where M shows the number of data instances and N the maximum number of features. Due to the fact that the kernel functions computed in parallel presume that the necessary dot-products are produced we consider the inner loop to be the dot-product and the outer loop to be the kernel function.

Ideally, an optimization in memory allocation would be to reduce as much as possible the data streams sent real-time. This would be achieved by initializing on-chip SRAM with small-sized vectors that are constantly traversed, and thereby more bandwidth would be available during the hardware processing. In any case, in the specific version of our architecture we could only utilize a small portion of the PCI bandwidth due to the fact that a small amount of vectors could be processed at a time due to limited resources. Moreover, the only data vector that is continuously traversed during the 2-level nested loop is data instance i. Thus, only mapping i to the SRAM could be meaningful and possible.

Nevertheless, the necessity to pipeline data transfer and computing is essential for a hardware design. In order to provide a better understanding of the dataset sizes we are considering let us present such an example. Recall that the *gisette_scale* dataset described in 4.2.2.4 is a 5M-dimensional problem with 6M training vectors. Given that LIBSVM involves 64-bit precision arithmetic the total size of training vectors equals 240 megabytes. Note that if we increase the training vectors of this dataset from 6M to 25M or the number of features from 5M to 20M then the size of training vectors reaches 1 gigabyte. Thus, in each iteration of the optimization step 2 gigabyte of data should be processed. We concluded that due to the fact that we could not approach the theoretical PCI bandwidth, as we had reached the maximum utilization of FPGA resources, it was sufficient to stream the chunks of data real-time into the DFE. We decided to not send the dataset to the DRAM to avoid the initialization cost induced by the data transfer time which could not be overlapped with computation.

Finally, data instance i and j are neither stored in the on-chip SRAM nor are retrieved from the off-chip DRAM. We decided to split the instance into individual features and pass each one of them to the DFE as a scalar. These are stored to the DFE in registers and with this practice we avoid any stream initialization cost and data transfer delay. Also, declaring the features as scalars in registers implies that the kernel will continuously have access to these registers.

4.1.3.6 Throughput Utilization

The MPC-C series of Maxeler provides up to 8 input and 8 output streams from CPU to the DFE and vice versa. In this first architecture we only used 4 input and output streams which implies that half of the PCI bandwidth was used. However, we could not utilize the rest of the offered bandwidth as the bottleneck of this architecture was the full utilization of available logic resources.

4.1.3.7 Observations on the First Architecture

The design and implementation of the first FPGA-based architecture on the dataflow platform allowed us to comprehend the notion of data flow computing, as well as the opportunities and difficulties introduced by the Maxeler platform. In order to assist future data flow designs we present the observations made during this process which also led to the second version of our implementation.

Observations

- The presented architecture exploited the available FPGA resources to the fullest, but the same did not apply for the available PCI bandwidth which reaches 2 GB/sec. In particular, we barely used the 50% of this bandwidth. This observation allowed us to understand that with better resource utilization we could significantly increase the number input pipelines and achieve balance between throughput and resources.

- In addition to the unexploited available bandwidth, in the first architecture we did not utilize the large off-chip memory located in the DFE. Simply streaming vectors from CPU to the FPGA and back was sufficient as we only considered small datasets in our first implementation. However, the off-chip DRAM is a tool that is essential for massive data processing.

- The Manager component of the DFE which provides the interface that connects the software and hardware is convenient to use. More specifically, defining inputs and outputs with the use of the API allows testing several cases and evaluating the overall performance. This is particularly important when we aim at reducing the learning curve of a new tool.

- We have stressed out that Maxeler is a data flow platform. In general, few hardware-oriented tools are specialized on data flow computing. If the target platform is an FPGA or a GPU card then the respective interface that will handle the continuous input should be implemented and tested, whereas Maxeler provides a simple API for tick-by-tick (cycle-per-cycle) input pipeline. In addition, it also offers the stream offset feature which provides easy access to data instances that precede or follow the current one. However, due to the nature of data flow computing we receive an actual result every N cycles which means that $N-1$ cycles produce indifferent outcomes. Thus, even if the stream offset property allows easy data access, it does not assist the programmer to overlap the aforementioned overhead with computationally useful time.

- It is clear that our first data flow SVM Training system did not perform efficient resource allocation. This limitation was primarily due to the big amount of multipliers and accumulations that are required even for relatively small datasets. In order to achieve maximum throughput utilization the number of arithmetic units had to radically decrease.

- Until now we have not mentioned the simulation mode offered by the data flow tool. It proved to be accurate, with results very close to the actual ones, both in terms of resource utilization and final arithmetic outcomes. Also, due to the fact that mapping an implementation to the DFE usually requires more than an hour, according to the size of the design, evaluating the design's performance using simulation is essential.

- Each hardware call introduces an overhead to the overall hardware processing time. We noticed that in the case of Maxeler this time equals 0.03 sec and in order to configure an efficient hardware design, the overall FPGA execution time should be significantly bigger than 0.03 sec, and more specifically should be 1% of the total hardware runtime thus being a negligible amount of time.

4.1.4 Second Hardware Architecture

In this section we describe our second attempt to create an accelerated hardware-based data flow architecture for the SVM Training phase. The novelties of the specific implementation were based on the observations made in the previous one 4.1.3.7. The organization of the second architecture's description follows the one of the previous architecture, thus allowing to indicate comparisons and differences more clearly.

4.1.4.1 CPU and FPGA Integrated System

This section is not a lot different than the respective one presented in 4.1.3.1. Once again, the host performs the sequential minimization optimization method and the hardware undertakes the kernel function computations. One of the main differences in a high-level perspective of the second architecture compared to the first is that now data instances i and j of the selected working set are not split into scalars, but are streams. Thus, the reconfigurable system does not depend on the feature space of the training dataset, but is fully parameterized; it changes dynamically the dimensions the data structures according to the needs of the application. Another high-level modification is that now the training dataset is not inserted into the DFE from the CPU in streams. More specifically, the training data samples are stored in the beginning of the program execution into the off-chip DRAM (LMem) which is also hosted in the DFE 2.1.5. Then, the LMem feeds the Kernel with a data element per time instance, whereas previously this task was carried out by the host. The modified top-level architecture is shown in Figure 4.5.

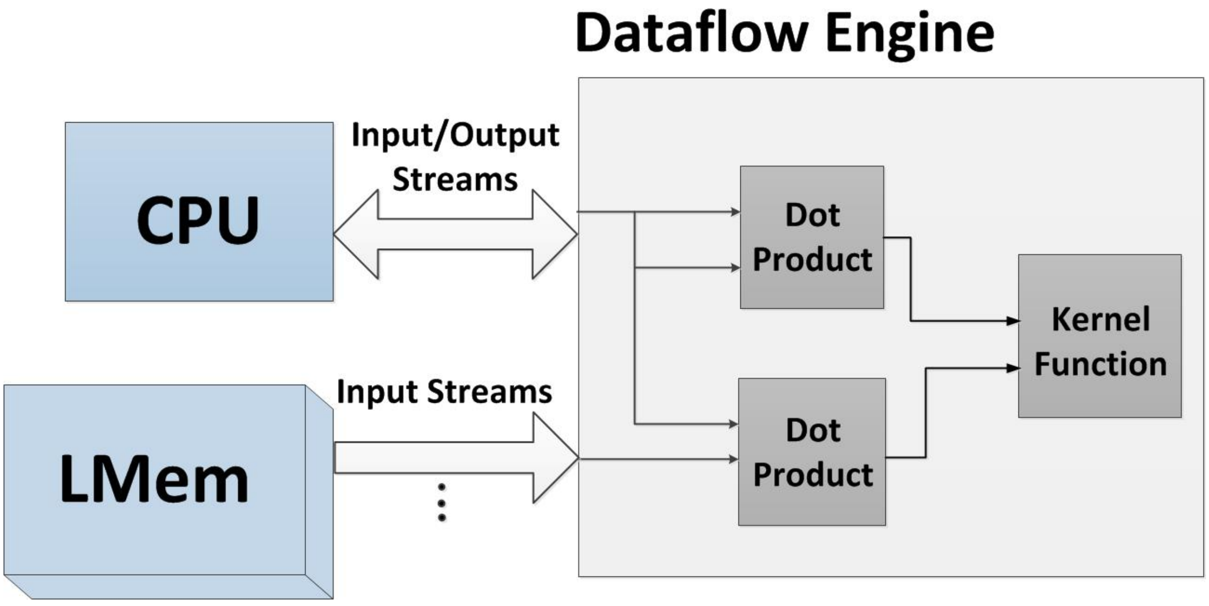


Figure 4.5: Second SVM Training Top level Architecture

4.1.4.2 Problem Partitioning

Recall that in 4.1.3.2 we could only divide the problem into 4 chunks that consumed all the available resources. Nevertheless in the second, improved architecture we achieve a more efficient resource utilization, and thereby the allowed number of chunks becomes 7. This shows that we process more elements in parallel and there is also better throughput utilization.

More specifically, the same data pre-processing as in 4.1.3.2 is performed , yet for 7 chunks. Hence, 7 one-dimensional vectors are created all of equal size. These streams are pipelined into the DFE and produce simultaneously 7 kernel computations, instead of 4.

4.1.4.3 Data Movement on Host Side

This section significantly differs from the respective previous one 4.2.2.3 due to the fact that we aimed at achieving dynamic data structure initialization and processing, as well as faster data transfer.

We first focused on how to replace the limited number of input scalars with a more efficient and convenient structure. Given that the only alternative choice was the use of streams it was our final selection. Recall that scalars are constant numbers stored in registers, as well as that all input and output streams in Maxeler should have the same length. However, neither can streams be stored in registers, nor can data instances i and j have the same length as the rest of the input streams. Based on these observations, the appropriate adjustments were carried out. More specifically, both problems were addressed by creating a stream of equal size as the other streams that contained consecutive replicas of the i or j instance. In other words, instead of storing each feature of these instances in a register, we continuously feed the DFE with the same values. Note that we need to stream i and j as many times as the size of the training set to produce the respective dot-products, and thereby the new vector will contain exactly the same number of elements as the rest of the streams. One one hand this approach is not limited to the feature space dimension and allows the kernel to "remember" the features of i and j. On the other hand however, an initialization overhead is introduced that cannot be hidden.

Furthermore, the input streams are no longer transferred from CPU to the FPGA device, but from LMem which is located on the DFE to the re-configurable platform. We compared the required time to transfer data from the two different sources for different sized datasets. In brief, results showed that data transfer which origins from the off-chip memory is always faster. Still, this does not necessarily imply that the use of LMem is always efficient as one should also take into account the overhead required to write and read from the off-chip memory. Based on our findings regarding data transfer and on the fact that our implementation only requires writing on the off-chip memory which occurs few times, we decided to stream the data samples from the LMem to the DFE. The described top-level architecture is illustrated in Figure 4.5. The different data origin is defined in the Manager interface. Moreover, the kernel still considers the input as a stream and receives a data element per cycle, regardless of the source. The main difference compared to streams coming from the CPU is that in order to write a vector into memory it must be a multiple of the number 384 for memory alignment purposes that we are not aware of. To conclude the exact interaction between the host, the off-chip memory and the FPGA is that the 7 streams containing the partitioned training data samples are written few times from the CPU into the LMem, and in each hardware call the entire streams are sequentially inserted tick-by-tick in the FPGA.

Note that even though we designated a data path from the LMem to the FPGA, it is a one-way path as the results produced by the hardware implementation are returned to the host and are not written into the memory. Writing the results back to the LMem would yield a prohibitive overhead as results should be written for each hardware call. On the contrary, the same training dataset is traversed multiple times, and thereby the respective writing overhead is negligible compared to the entire runtime of the program. Moreover, writing back to the LMem would also allocate additional memory ports which could be exploited more efficiently. Thus, it is efficient to read the training data samples from the LMem, but not to read the produced results from the memory.

4.1.4.4 Dataflow Kernel Computation on Hardware Side

The hardware design of the second architecture significantly differs from the first. As we have already mentioned our main goal was to achieve efficient resource allocation in order to allow further throughput utilization. The modified system is illustrated in Figure 4.5. Due to the fact that we pipeline 7 streams in this architecture, instead of 4 we have 7 parallel processing units to reach maximum performance. Each of these units produces a partial kernel computation that is sent back to the host in order for the final output to be carried out.

More specifically, recall that in the previous architecture we needed $\frac{n}{2}$ multiplications and totals for the computation of each product, where n denotes the feature space of the training set. The new core of the Kernel, which will be discussed in detail is illustrated in Figure 4.6.

Unlike 4.2.2.4, in the new architecture we replace the $\frac{n}{2}$ number per dot-product with 1. Given that we pipeline 7 streams into the kernel, the hardware produces 7 partial kernel functions similarly to 4.2.2.4 where we inputted 4 streams and produced 4 sub-kernels. In order to do so the FPGA carries out 14 dot-products in parallel, and therefore

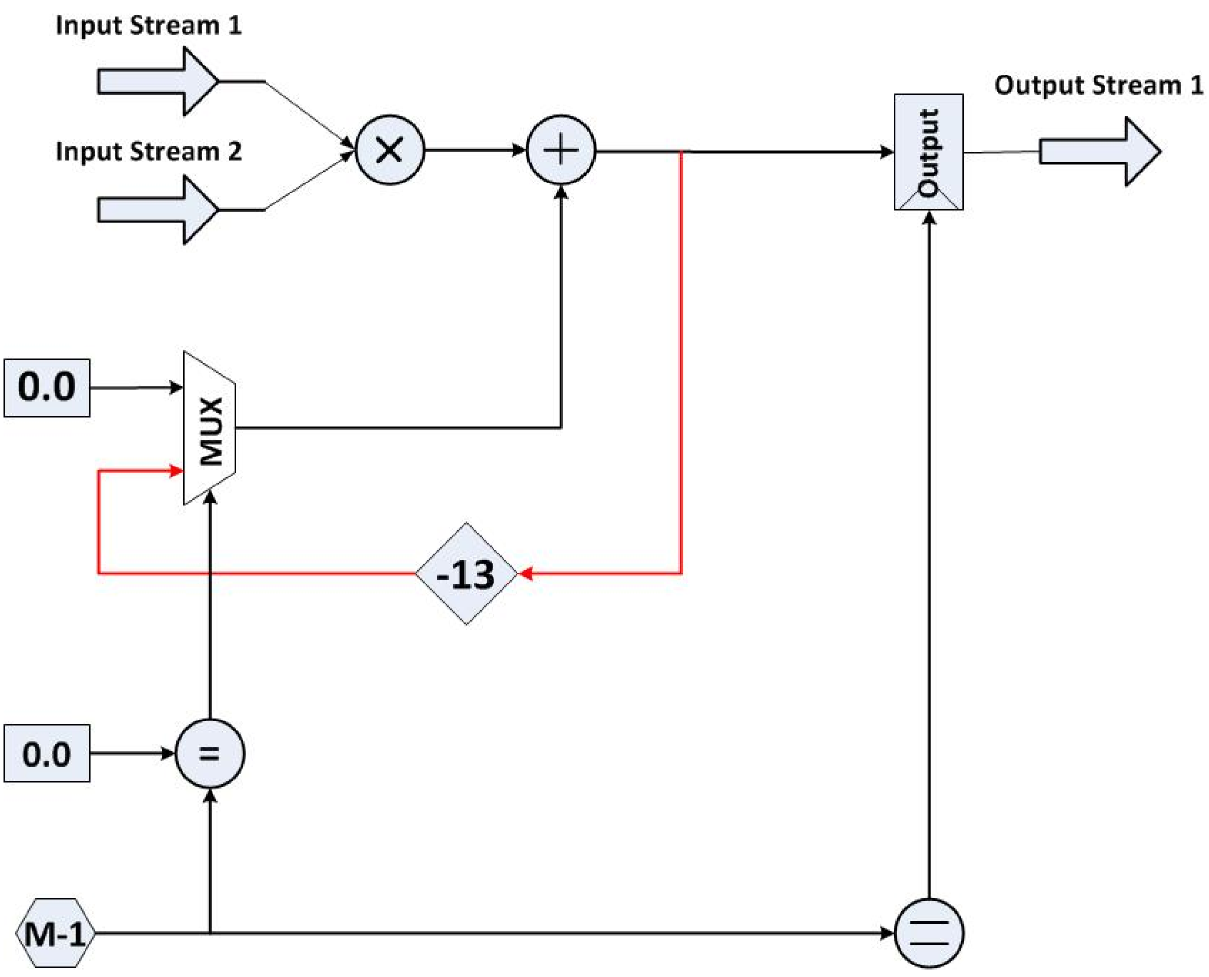


Figure 4.6: Second Dot-Product Computation Hardware Core

a total of 14 multiplications and 14 sums will be required for the whole design. Note that in the previous architecture for the same purpose we would need $14 * \frac{n}{2}$ such numerical functions.

In order to achieve practical resource utilization we completely modified the core of the Kernel. In the trivial loop used for the dot-product computation we have looked at so far, each iteration of the loop relies on a previous value. In particular, the appearance of a new feature of a data instance leads to a multiplication of this feature with the respective feature of instance i, as well as to an aggregation of the computed value with the rest of the subtotal, i.e. $sum = sum + i_n * x_n$, where sum is the remainder of the dot-product computation, and i_n and x_n show the features of data instances i and x respectively.

Recall, that the kernel computation also requires computing value $sum = sum + x_n * x_n$, which denotes the dot-product of a data instance with itself. Hence, since we have 2 partial aggregations for each parallel instance, and given that there are 7 parallel units running on the DFE , we have to preserve 14 such partial aggregations. However, each of these aggregations creates a pipeline where each stage of the pipeline calculates a value of *sum* based on the value of *sum* from the previous stage of the pipeline. Due to the fact that there is dependence from one iteration to the next, and thus a cycle is created in the data flow graph.

In our first architecture we removed the cycle of the data flow graph by fully-unrolling the innermost loop, but a big number of features led to a prohibitively large amount of logic. This approach would have been efficient if the innermost loop was as small as at most 10 loops. In the respective improved version, we pass the subtotal back into the adder, creating a cycle. The backward edge is shown as a dotted red line in Figure 4.6. In order to keep track of the current position of the input data instance we utilize an advanced counter which resets all values when a new data instance appears in the kernel to begin the computation of a new dot-product. The cycle in the kernel data path causes an overhead in the beginning of the hardware execution, and therefore the first useful output will be produced some ticks after the other useful results.

Furthermore, we need a variable that will carry the value from the previous iteration. This variable can be considered as a wire that connects the result of the previous iteration, which is stored in a register, with the adder, thus creating the cycle in the data flow. The hardware-based loop comprises three parts. The first is a multiplexer that selects the initial value of an iteration. In our case this value is initialized with 0 every time that a dot-product computation is completed and a new data instance flows into the kernel. During the dot-product computation the multiplexer selects the dependent value produced step-by-step in the previous loops to be used in the following ones. Hence, the second part of the loop computes the updated new value. Finally, the third part of the hardware-based loop is responsible for connecting the new value back.

In section 4.2.2.4 we used the stream offset property of the data flow programming tool to access future elements. In our new approach we also need to access an element other than the current one, as we need to ”remember” the previously generated value. However, there are two main differences compared to the first architecture, which are the number of stream offsets used and the range of the window. The first modification is easy to comprehend as instead of $N-1$ stream offsets for the subsequent $N-1$ features of a data instance , we only require one that will access the previous updated subtotal. On the other hand, understanding the window length is not as trivial since it is not easy to comprehend when will the old subtotal be updated with a new value as by the system. In order to achieve a parameterized architecture and to avoid mistakes introduced by human observations that may lead to unnecessary delay we used the autoloop offset feature of Maxeler that automatically calculates the lowest valid offset for a cycle in the graph. Specifically, in our example the adder of double-precision is a multi-stage pipeline with the results being available after 13 ticks from the beginning of the function. Given, that prior to the sum we introduced a multiplexer which has a pipeline depth of 1. Thus, the pipeline created has depth equal to 14 and setting the stream window respectively yields valid outcomes.

Finally, the 14 clock cycles delay introduces the need to control the stream output. Let us assume that we do not monitor the outputs of the kernel and send all produced elements to CPU. Then, given that we need N features of a data instance to flow in and out of the kernel, as well as 14 additional ticks per feature to update the previous value with the new, it is clear that $N*14$ results would be outputted in total for a single data sample. However, the number of kernel clock cycles should equal the number of elements in the training data set since we need each feature of every data instance to be pipelined to the kernel. Hence, instead of producing $N*14$ results per data sample, we control the output stream to yield an outcome every 14 cycles.

4.1.4.5 Memory Allocation

In section4.2.2.4 we provided an example of a 5M-dimensional problem with 6M training vectors. We also mentioned that with the use of 64-bit precision arithmetic the total size o this dataset equals 240 megabytes. Hence, mapping the entire dataset to the BRAM to achieve the fastest data transfer is not possible. We considered updating the on-chip memory with blocks of the training set at runtime. However, transferring data from the CPU to the BRAM for each hardware call requires essentially longer time than writing the data from the CPU to the off-chip memory few times during the execution of the program. Based on the same observation we concluded that writing the outputs of the Kernel back to the LMem would also introduce a significant overhead compared to simply streaming them to the CPU via the PCI. However, in case an application reuses data during a single hardware call it is definitely more efficient to update either the on-chip or the off-chip memory respectively depending on the size of the data.

Moreover, ideally we would retrieve data instances i and j by addressing the LMem appropriately and repeatedly traversing and sending the same instance to the FPGA. Addressing the elements of the working set in such way would only insert a negligible stream initialization cost since the stream would have length exactly N, where N denotes the feature space of the data set. However, the Maxeler tool does not allow random accessing the memories of the DFE. It only permits sequential traverse that is declared prior to the program execution in the Manager API. Yet indexes i and j change in each iteration of the optimization problem, and thereby it is impossible to retrieve them from the off-chip memory. Thus, we create a vector in the host program that contains the instance i or j replicated several times, as described in 4.1.4.3 and stream this structure at runtime. The overhead introduced equals the time required to initialize the vector with the appropriate values plus the time to initialize the stream that will be sent to the FPGA.

4.1.4.6 Throughput Utilization

Our main concern in the previous architecture that led to a completely different hardware approach was the limited throughput utilization offered by the PCI. By reducing the arithmetic operators we were able to pipeline 3 additional input streams to the FPGA which overall yielded the production of 7 kernel functions concurrently. Each element streamed either from the LMem or from the host, is inserted once in the DFE and then it is discarded. It is this challenging I/O ratio that makes it difficult to create parallel instances of independent units. Hence, focusing on utilizing adequate memory access bandwidth is crucial for any data flow computing architecture.

4.1.4.7 Observations on the Second Architecture

Even though the majority of the second architecture was designed based on the observations made during the implementation of the first approach, still the new system resulted in a new set of remarks. Some of these could be addressed, thus leading to an enhanced second version implementation, while others can only be used as considerations for similar future hardware designs.

Observations

- The first architecture requires $M * N$ kernel clock cycles to produce the total kernel functions, where M the training set size and N the respective feature space. However, although the second architecture was improved in terms of efficient logic resources and throughput utilization, it also introduced an additional delay of 14 ticks per data instance. Thus, in order to output all results the second architecture needs $14 * M * N$ clock cycles. Furthermore, note that the number 14 corresponds to the pipeline of a 64-bit adder. It is an arbitrary number whose number increases for more complex functional units. We cannot avoid waiting $M * N$ clock cycles to produce all outcomes, since it is necessary that the entire dataset floats through the DFE. In addition, waiting as much time multiplied by 14, eliminates our efforts to enhance parallelism and the efficiency of our design. Therefore, it is clear that the overlap of the 14 cycle delay with beneficial computational time is critical for the second architecture.

- The maximum number of possible input and output streams that can be transferred from and to the CPU via the PCI is 8 and 8 respectively. Recall that for each input stream we also produce a respective output stream, and thereby given that we pipeline 7 streams from the LMem into the DFE we also need to stream back to the CPU 7 streams through the PCI. Hence, 7 out of 8 output streams are being used which is very close to the optimal. We do not need to use the 8 input streams since our findings showed that transferring data from the LMem was faster, thus allocating 7 streams from the LMem to the DFE instead. However, the LMem allows a maximum of 15 streams to be connected which implies even better throughput utilization can be achieved, if supported by the implemented algorithm.

- Note that we could partition the problem into 8 chunks instead of 7, since the maximum number of streams is 8. Yet, even though we did so the debugger did not permit as to continue by claiming that we had reached the maximum number of possible output streams. This was an inconsistency that could not be easily interpreted as no other message appeared. We assume that it is due to the fact that we use 64-bit precision and more than 7 streams of the PCI were eventually required internally to send all of the data.

- In general, inconsistency was a matter that we encountered several times while creating the second architecture. The simplicity of the design itself was not reflected during the implementation. For example, we had completed a design that used the Advanced Dynamic SLiC interface, instead of the default Manager interface. The advanced interface allowed us to control the data movement and hardware calls more efficiently as the programmer specifies the properties of the interface himself. Yet, in the actual DFE execution the advanced interface presented several bugs when the dataset size increased which were not caused from our part as the same bugs also appeared in their own examples. Another example was encountered when the second design was performed on relatively small datasets where each hardware call required at most 10 sec. In these cases we noticed that the hardware execution time for a constant and pre-defined number of cycles initially was very small (0.03 sec) and increased until it reached a peak (0.15 sec) to decrease to the initial time again (0.03 sec). This process continues repeatedly throughout the entire program runtime and shows that there is probably some buffer that fills gradually and then empties. In order to only keep the small execution times we loaded and unloaded the max file, which is similar to a bit stream, on and off the DFE before and after the hardware call respectively. Other problems regarding the pre-defined number of cycles, the accuracy of the simulator and others were also met during the implementation, thus showing that several properties provided by the Maxeler platform are still under development.

4.1.5 Improvements on Second Architecture

Recall that even though the second design achieved efficient memory and resource allocation, as well as adequate bandwidth utilization, there was a prohibitive overhead induced by the the cycle in the data flow graph. More specifically, due to this cycle the required time to produce an output was not equal to $M * N$ clock cycles as expected, where M the number of data samples in the training dataset and N the corresponding feature space.

In fact, it became $14 * M * N$, where the number 14 denotes the depth of the pipeline required to update the previous dependent value with the new. Our main goal was to improve the second architecture by eliminating this multiplier.

In order to do so we basically modified the order of the data sequence in the software, without affecting anything on the hardware side. More specifically, in 4.1.3.2 and in 4.1.4.2 each chunk was transformed into a one dimensional vector. This vector comprised concatenated rows, where each row depicted a data instance of the training set. To overlap the delay of the cycle with useful computational time we basically changed the sequence according to which the 2-dimensional matrix is transformed into a vector, thereby changing the sequence of the data flow. The 14-depth pipeline yields a 14 cycle delay per data element in the DFE. We continuously feed this pipeline with new data elements so that it is never empty, as it was in the first version of the second architecture. Let us assume that $feature_1$ of $instance_1$ is inserted in the pipeline. We know in advance that this pipeline has 14 stages, and therefore the updated value will be produced after 14 clock cycles. Thus, $feature_2$ of $instance_1$ should be inserted to the FPGA 14 clock cycles later to produce the next partial result. In between, immediately after $feature_1$ and its updated value is in the second stage of the pipeline, we feed the first stage with a new partial value which belongs to $feature_1$ of $instance_2$. Next, $feature_1$ of $instance_3$ follows until the pipeline is full with the last element inserted being $feature_1$ of $instance_{14}$. Then, the exact next clock cycle the updated dot-product of $instance_1$ will be ready and $feature_2$ of $instance_1$ is inserted to the pipeline and the same process continuous. Once all the dot-products of the first 14 data samples have been produced, $feature_1$ of $instance_{15}$ flows into the DFE and so long. Thus, starting from the first element of the 2-dimensional matrix, the sequence of the data flow is vertical until the element which fills the pipeline of the cycle in the graph is met (in this case 14). Then, the next data element is the second feature of the initial data instance and the elements are again traversed in the same vertical manner. The final data traversal is similar to a zigzag. This process is completed once the entire chunk has been transformed to a vector.

Note that the pipeline is continuously fed. Thus, the first dot-product result is produced after $N * 13$ clock cycles, the second after $N * (13 + 1)$ and so on. Yet, the total amount of clock cycles required to produce all the partial kernels of a chunk in the improved second architecture is approximately $\frac{N*M}{7}$ since all the features of each data instance should be processed by the FPGA.

4.2 Mutual Information

We perform the software code analysis of the Mutual Information (MI) computation from the hardware designer's perspective, following the same steps as those followed for the SVM Training phase analysis 4.1.1. This allows us to estimate the hardware opportunities and bottlenecks induced by the MI calculation.

4.2.1 Modeling for Hardware

The structure of the MI hardware modeling section is exactly the same as the one in the respective SVM Training one.

4.2.1.1 Inputs and Outputs

Mutual Information receives as input two random variables in the form of time-series and produces a single arithmetic value which is the result of the MI computation. The input time-series can have various sizes provided that they fit into memory. The arithmetic precision of MI depends on the type of the input time-series that are being considered, and therefore by the application's arithmetic requirements.

4.2.1.2 Algorithm Profiling

In order to calculate the final Mutual Information statistic value between two random variables, the mutual and joint Probability Density Functions (PDF) need to be estimated based on formula 2.3. Yet, the time required to produce the PDF estimations directly depends on the length of the time-series, i.e. the size of the input data. Table **??** depicts this dependence by showing how the execution time for the PDF estimation scales with time-series of different sizes. More specifically, the results in Table 4.2 show that the execution time for PDF estimation increases linearly with the input size.

Furthermore, due to the fact that all MI computations sought to estimate densities given a finite number of data points an appropriate estimator should be used. We used the equi-distant histogram estimator which is the simplest non-parametric density estimator and is easy to produce and comprehend. Note that regarding the resolution R of a histogram, when this becomes higher for a specific application then granularity becomes smaller and MI accuracy is improved. This resolution shows the number of bins used

Time-Series Size(elements)	**Execution Time (ms)**
10000	0.23 **(1)**
100000	1.53 **(2)**
1000000	14.3 **(3)**
10000000	143 **(4)**

Table 4.2: Dependency between the execution time required for PDF estimation and the size of the inputs.

Number of Bins	**Execution Time (ms)**
100	0.11 **(1)**
500	24.5 **(2)**
1000	98.2 **(3)**
5000	2460 **(4)**
10000	9870 **(5)**

Table 4.3: Dependency between the execution time required to compute MI and the number of the bins in the histogram.

to quantify the time-series into sets. Yet, the increase of resolution also causes the squared increase of the execution time required to compute the MI value, as the time complexity for the MI computation is $O(R^2)$. This is reflected in the MI formula presented in 2.38 which requires iterating over the two PDF estimations P(X) and P(Y). The aforementioned dependency between resolution R and PDF estimation execution time is illustrated in Table 4.3.

The results of the software profiling analysis in Tables 4.2 and 4.3 indicated that the most time-consuming part of the process followed to calculate the MI value between two random variables with the use of histograms, was the final computation of this statistic value. The only exception is observed for a small histogram resolution and a great number of input data. In such cases density estimation consumes more time than MI calculation. One such example is when the number of bins equals 500 (24.5 ms) and the number of input data is equal to 10^7 elements (143 ms). Still, even though in a few cases the PDF estimation execution time is the majority of the entire program runtime, these cases are met in a limited amount of applications. Thus, we address the general case which suggests that in order to accelerate the MI calculation between two random variables, formula 2.38 needs to be parallelized and mapped to hardware. For example, for 1000 bins and 10^5 input elements, the MI computation requires 98.2 ms, whereas the creation of the histogram only needs 1.53 ms. Hence, the former constitutes 98% of the overall execution time of the program, while the latter requires only 2%.

4.2.1.3 Important Data Structures

In this work equi-distance histograms were used to estimate density, due to their popularity, fast performance and convenient use. Note that configuring optimal algorithmic solutions to yield maximum accuracy is not the scope of this thesis as we aim in MI acceleration. 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. Moreover, the input time-series are stored in one-dimensional structures of type float whose sizes depend on the number of the window we have selected. This window basically represents what percentage of the time-series we are studying at a time.

4.2.2 Hardware Architecture

The hardware implementation of the MI computation was significantly based on the observations made during the SVM Training hardware-based implementation. Naturally certain points were adapted to the requirements of the new problem, but the main body of the new special-purpose approach was built based on the previously derived observations.

4.2.2.1 CPU and FPGA Integrated System

The MI system architecture is illustrated in Figure 4.7.

In general, the MI processing is divided into two basic stages, the probability density function (PDF) estimation and the final calculation of Mutual Information based on formula 2.38. Our findings during profiling showed that a histogram-based PDF estimation is not sufficiently computationally intensive to be mapped to hardware, and thereby this

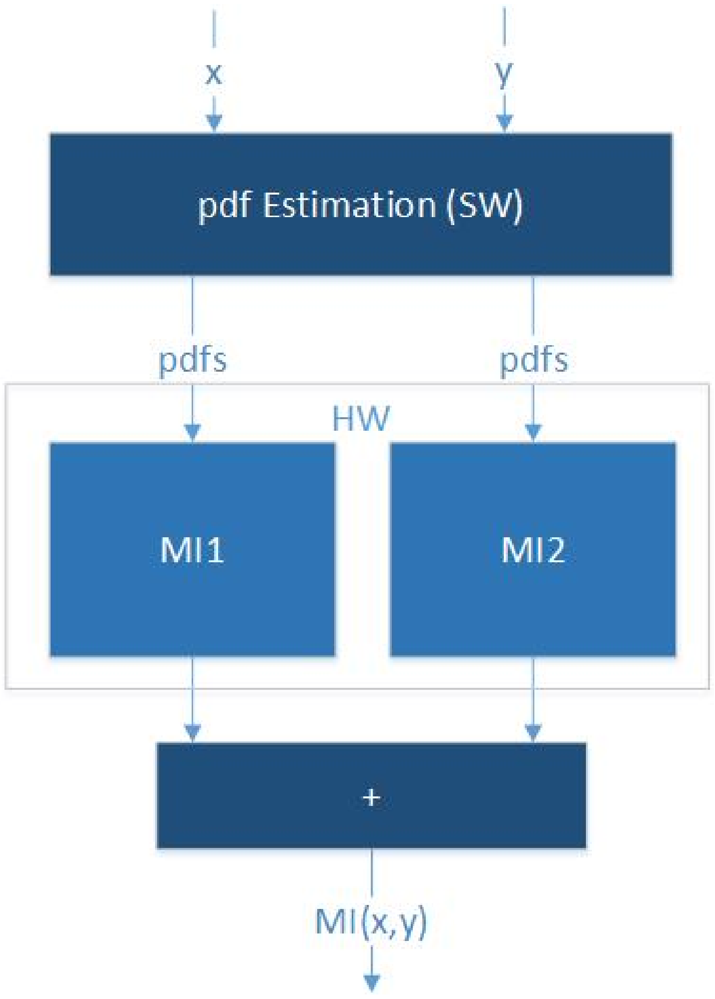


Figure 4.7: Mutual Information Basic System Architecture

task was carried out on the host side. Unlike PDF estimation, the profiling results presented that MI calculation execution time can reach 98% of the total program runtime. Thus, the core MI computation was assigned to the hardware side. Calculating the final MI value on hardware would result in reduced performance as it would disrupt the pipeline flow by introducing controlled inputs and outputs. Hence, the results produced by the DFEs are streamed back to CPU which accumulates the individual parts to yield the MI value. We resorted to this approach because it was the most efficient one, given the Maxeler system.

4.2.2.2 Problem Partitioning

Given that our main goal was to accelerate the kernel function computation, we carried out the necessary problem partitioning. More specifically, given the produced density estimations P(X), P(Y) and P(X,Y) the final outcome is ready after R^2 clock cycles, since all streams have length size equal to R^2. In order to reduce this time to $\frac{R^2}{2}$ we simply considered splitting P(Y) and P(X,Y) into two streams. Then, each half of P(Y) and P(X,Y) is processed simultaneously as there is not dependence between the different levels (bins) of the distribution function. Note that we do not have to split P(X) 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 concurrently two partial MI, the first produced from the first half $\frac{R}{2}$ levels and the second produced from the second half $\frac{R}{2}$ levels that are summed on hardware and streamed back to the host. Hence instead of the original R^2 clock cycles required to produce a MI result , with the aforementioned partitioning we divide this time by the factor of 2, i.e.$\frac{R^2}{2}$ clock cycles.

4.2.2.3 Data Movement on Host Side

Based on 4.2.2.1 we need to stream the PDF functions P(X,Y), P(X) and P(Y) from the host side to the hardware side in order for the MI calculation to take place. The length of each of these streams is defined by the maximum length that needs to be sent. Given that P(X,Y) is a two-dimensional structure 4.2.1.3 of size R^2, where R denotes resolution, we know in advance that the length of the streamed vectors will also be R^2. However, the fact that distribution estimations P(X) and P(Y) are one-dimensional vectors of size R led to the decision to stream each one of them R times into the DFE so as to match the size of P(X,Y). This introduced an additional consideration, to stream P(X) and P(Y) in the correct order. The distribution estimation of random variable X is simply streamed R times with the use of a R single loop. However, re-ordering P(Y) appropriately induced an initialization cost as it could not be replicated R times. Furthermore, 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) estimations should arrive simultaneously at the DFE component.

4.2.2.4 Dataflow Kernel Computation on Hardware Side

The basic hardware architecture that corresponds to the substance of the MI calculation 2.38 is shown in 4.8. Note that our system uses two parallel special-purpose units to compute an MI value twice as fast compared to a single hardware core. Thus, the aforementioned architecture represents each one of these cores shown in Figure 4.7.

We stress out that the MI hardware-based architecture is fully pipelined, and thereby an iteration of the MI accumulation is performed every clock cycle. The three PDF estimation vectors are initially inserted in the pipeline, one value of each vector per clock cycle. Then, these values are processed in the pipeline and the respective results are accumulated in the *Sum* module. The basic arithmetic functions, illustrated in 4.8 with circles use single-precision floating-point, whereas more complex arithmetic operations denoted with squares utilize ready floating point hardware cores. In particular, the two square arithmetic components perform log approximation and accumulation, and are described in detail in the following paragraphs.

The logarithm base 2 approximation component was designed based on the fastlog2 work. This approach was selected due to the fact that it was proved very efficient with respect to resource utilization and computation overhead. In addition, the relative accuracy of this log approximation is approximately equal to $2.09352\exp{-05}$ which does not yield substantial differences. However, the existence of such accuracy propagates a bigger error to the final MI computation as this error is accumulated during the sum calculation of formula 2.38. In particular, for a high histogram resolution value R the accuracy reaches $1.0564\exp{-02}$. In order to implement a hardware-friendly log approximation module, we also considered two other methods, the first being the construction of Look Up Tables, and the second being Taylor Series. Although the former technique produced relatively good results in terms of accuracy , its utilization was limited by the BRAM resources required. For massive datasets and high resolution, the PDF estimations contained very small values which could not be addressed by the LUT method. Similar observations were also noticed in the Taylor Series technique for log approximation, since it only performed well and required limited resources allocation for specific inputs. More specifically, if a bigger time-series was inserted into the Taylor series method, then the computation required a lot of iterations to converge, and subsequently significant hardware resource

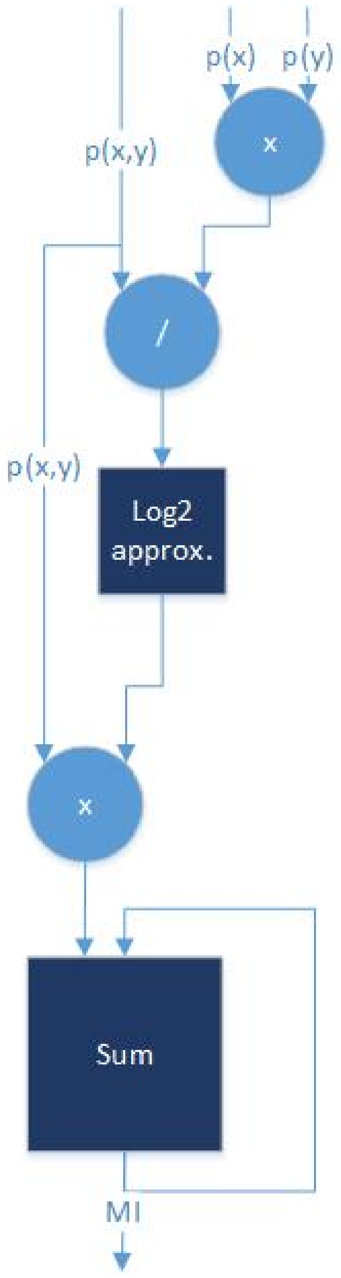


Figure 4.8: Mutual Information Hardware Core Architecture

allocation due to the fact that the Maxeler systems do not provide an efficient way to address operations with feedback loops.

Another optimization was made regarding the aggregating operator in 2.38. Recall that the *Sum* module is responsible for accumulating the partial results. Similarly to the kernel computation in SVM we need to store the previous partial value, and thereby a floating point adder is used. Thus, the same delay pipeline is introduced, only now it requires 13 clock cycles instead of 14 to be ready for the next accumulation. Note that in order to produce the correct hardware-based result one value per each of P(X,Y), P(X) and P(Y) density functions had to be streamed every 13 clock cycles, as we did in 5.1.2. Yet, this would create a 13 times slower system architecture. This overhead was prevented by rearranging the dataset in such way that an instance per stream is inputted every clock cycle, accumulated and stored in the 13-depth pipeline buffer, but with the first slot of the buffer containing the results of instances 1 + 14 + 26 + ... and so on. Similarly the second slot of the buffer contains the partial results of inputs 2 + 15 + 28 and so on. This applies for all the 13 slots of the feedback buffer. Thus, by the end of the hardware execution the 13 slots will contain the final 13 partial sums that are streamed out to the host in order to be accumulated. Finally, the CPU sums the 13 last elements of each received stream and produces the final MI outcome. When two parallel MI cores are implemented on hardware we receive two output streams, leading to the final accumulation of 26 partial results, 13 introduced by each core.

4.2.2.5 Memory Allocation

In the Mutual Information computation hardware-based approach we did not utilize the properties of the off-chip memory as we did in SVM. The reason is because there was no dataset that was continuously traversed. Recall that in the case of SVM a specific dataset which was loaded in the LMem from the beginning of the program, was read twice in each iteration. However, for each MI calculation each time-series is read once with a single call to hardware, and thereby there was no need to use the on-chip and off-chip memories.

4.2.2.6 Throughput Utilization

The single core MI computation utilizes 3 out of the 8 available streams of the dataflow architecture. In order to further accelerate the application the PDF estimations were divided into half, and in this case 5 streams were used instead of 3. Note that the new number of streams is not 6, because once again the same vector P(X) is streamed continuously. Thus, we only had to divide P(Y) and P(X,Y). The use of two parallel cores instead of one resulted in better bandwidth utilization, and consequently better performance. However, mapping on hardware two cores instead of one only allocated 10% of the available resources. Further exploitation of the FPGA platform is bounded by the number of available PCI streams.

Chapter 5

Experimental Results

5.1 Support Vector Machines

5.1.1 Setup

We implemented parallel kernel computations on a MAX3A Vectis PCI Express card version of a Maxeler Dataflow Supercomputing System. The MAX3A Vectis card is inserted into an Intel-based workstation with an Intel XEON 2 6-core processor running at a clock rate of 3.2 GHz with 50GB RAM. MAX3A Vectis card is equipped with a Xilinx Virtex-6 FPGA device and 24 GB of DDR RAM. Connection to the host was via PCI express and all designs were compiled to run at 200 MHz. We used datasets of various dimensions as one of the goals of this thesis was to study the behavior of the parallel kernel computation for large datasets.

For our experimental results we utilized the *ijcnn1* dataset [94] with $49,900$ data instances and 22 features, the *gisette* dataset [93] with $6,000$ data instances and $5,000$ features and the *colon-cancer* dataset [95] with 62 data instances and $2,000$ features. We also created a small dataset with $4,320$ data instances and 4 features based on tick-by-tick financial stock values called *financial_stock*. Furthermore, in order to increase the computational demands of the specific datasets we enlarged them by various factors. Thus, if dataset *colon-cancer* corresponds to a dataset of size 62x$2,000$, where 62 shows the number of data samples and $2,000$ denotes the feature space, then we refer to the corresponding augmented by a factor of 70 dataset 70*62x$2,000$ as *colon-cancer_70*. Moreover, the software reference is the LIBSVM implementation which is widely accepted as one of the fastest and more accurate open source packages for SVM Training.

5.1.2 Results

5.1.2.1 First Architecture

In this section we present and interpret the results produced by our first SVM Training architecture. More specifically, Table 5.1 presents the resource summary.

	Used	**Maximum**	**Percentage**
LUTs	231414	297600	77.76%
FFs	282627	297600	94.97%
BRAMs	59	2128	25.19%
DSPs (Multipliers)	1760	2016	87.30%

Table 5.1: MAX3A Vectis Resource Utilization of First Architecture

Note that as described in 4.1.3.7 this architecture significantly utilizes the resources offered by the FPGA. For example, our first architecture occupies 87% of the available DSP units and 94.9% of the offered flip flops. Thus, it can be derived that additional parallel units could not be inserted in the specific FPGA, even though approximately only half of the available bandwidth was used.

Moreover, Table 5.2 presents the experimental results of our first approach for datasets with relatively small feature space. Recall that the specific system could not be tested on bigger datsets due to lack of available registers. It is clear that the software implemen-

Dataset Size	**SW (sec)**	**HW (sec)**	**Speedup**
financial_stock	1.12	12.65	0.08
financial_stock_2	4.47	34.08	0.13
ijcnn1	27.19	276.26	0.09
ijcnn1_2	130	982.74	0.13
ijcnn1_6	1804	9555.84	0.18

Table 5.2: Performance of First SVM Training Dataflow Architecture

tation is significantly faster than the first hardware-based approach. In particular for smaller datasets *financial_stock* the special-purpose implementation begins from being approximately 11 times slower, whereas for bigger datasets *ijcnn1_6* it becomes 5 times slower. These results can be interpreted based on the following observations. To begin with, we have computed that a call to the hardware platform requires at least 5 to 30 ms. We also calculated the actual hardware processing time in order to evaluate the percentage of the total hardware execution time it constitutes. Specifically for the biggest dataset *ijcnn1_6* the hardware processing time was approximately equal to 23 ms, while the total hardware runtime was around 50 ms. This shows that half the time was consumed to tasks irrelevant to the kernel computation. Therefore, we concluded that by evaluating bigger datasets where the hardware initialization overhead will be insignificant compared to the total hardware run-time, the overall performance could significantly improve.

In addition, the software time required to produce a single kernel computation was equal to 17 ms for dataset *ijcnn1_6*. Given that the hardware initialization time is approximately 27 ms, one concludes that it is impossible for hardware to overcome software for datasets of similar sizes. We had to evaluate datasets whose software kernel computation time significantly exceeded the maximum possible hardware initialization cost which is 30 ms. The reason why the call requires so much time can be attributed to the stream initialization cost, the operation system workload, data transfer from the CPU memory to hardware and other reasons that are attributed to the Maxeler system. In particular, the reason why the 4-core parallelization did not work is because the runtime was already completely dominated by set-up overhead. We noticed that even though increasing the number of streams allowed more parallel processing units, it also led to increasing the set-up overhead. This could only be overcome by inserting meaningful chunks of computation that require dozens of milliseconds.

Furthermore, in order to exploit the maximum possible throughput utilization more parallel kernel computation units had to be added to the architecture. Thus, processing more memory intensive streams in parallel could respectively accelerate the overall computation time. Thus, it was useful to explore different architectures as with the specific one we could not evaluate the system's potential for big datasets which are usually addressed by dataflow systems.

5.1.2.2 Second Architecture

The second architecture aimed at improving the available bandwidth utilization to achieve better throughput. The new resource summary is presented in 5.3.

	Used	Maximum	Percentage
LUTs	68025	297600	22.86%
FFs	98118	297600	32.97%
BRAMs	536	2128	25.19%
DSPs (Multipliers)	140	2016	6.94%

Table 5.3: MAX3A Vectis Resource Utilization of Second Architecture

Note that resource utilization is carried out less efficiently compared to our first approach since approximately the 39% of the FPGA is occupied. Of course we would like to allocate more resources thus allowing more hardware-based processing, but in our second approach resource utilization is bounded by the available PCI express bandwidth due to the fact that we use these lanes to return the results of the 7 parallel processing units.

Even though our goal was to improve our first approach by utilizing more throughput, Table 5.4 shows that the second architecture yielded worse results. Note that although

Dataset Size	SW (sec)	HW (sec)	Speedup
financial_stock	1.12	32.76	0.034
financial_stock_2	4.47	95.42	0.046
ijcnn1	27,19	686.58	0.04
ijcnn1_2	130	2500.86	0.052
ijcnn1_6	1804	21021	0.04

Table 5.4: Performance of Second SVM Training Dataflow Architecture

our second approach could evaluate datasets with feature space bigger than 22, we did not conduct such experiments. We reached this conclusion because it was clear that compared to the first architecture the second architecture's performance was substantially worse, and therefore considering datasets with higher dimensions would not yield any substantial observations or results.

The main reason that led to this deterioration was the fact that the double-precision adder required 14 clock cycles to update the previous partial sum with the updated one. Recall that this introduced a 14 clock cycle delay to the overall hardware execution time, due to the fact that the 14-depth pipeline produces a useful result every 14 clock cycles. Again, we computed the total hardware execution time, the hardware processing time and the hardware initialization time, with values 58 ms, 46 ms and 12 ms respectively. Yet, note that the 14 clock cycle delay is not reflected in the hardware execution time, as it increased from 9555 sec (first architecture) to 21021 sec (second architecture). In fact the overall execution time of the first approach hardly increased by a factor of 3. This can be attributed to two main reasons, the first being that we introduced three additional parallel units and the second being that Maxeler only needs to initialize one stream, as the rest are inputted from the off-chip memory. In addition, another suggestion is that the off -chip memory and the CPU use different buffers that handle data more efficiently when these are less. In other words, it is different to send all streams from the CPU than to partition this task among the DRAM and the host.

Nevertheless, again we noticed that as the dimensions of the dataset increased, the difference between the software and hardware execution times became smaller. The smaller dataset *financial_stock* requires 1.12 sec overall execution time on software and 32.76 sec on hardware which implies that the hardware is around 31 times slower. On the other hand the biggest dataset *ijcnn1_6* requires 1804 sec software execution time and 21021 on hardware, i.e. it is 11 times slower. However, neither this design allowed us to draw safe conclusions due to the 14 clock cycle delay overhead.

5.1.2.3 Improved Second Architecture

Our third and final approach was an improved version of the second architecture. More specifically, we aimed at hiding the 14 clock cycle delay by rearranging the data sequence, thus achieving a fully pipelined implementation. Recall that resource utilization was reduced, but at the same time we reached maximum possible throughput utilization. We do not present a table with resource summary for the improved architecture as it is exactly the same as in 5.3.

Even in our last attempt to create an accelerated SVM Training system by utilizing the maximum bandwidth and by performing in parallel as many operations as possible, we did not manage to accelerate the respective LIBSVM software execution time. Our final experimental results are presented in Table 5.5.

Dataset Size	SW (sec)	HW (sec)	Speedup
ijcnn1	27.19	145.56	0.18
ijcnn1_6	1804	5733	0.31
ijcnn1_12	6795	19536	0.34
gisette	$16*10^5$	$44*10^5$	0.36
gisette_3	$5*10^6$	$134*10^5$	0.37
gisette_8	$102*10^5$	$352*10^5$	0.28
colon-cancer_70	2.71	8.8	0.3

Table 5.5: Performance of Improved Second SVM Training Dataflow Architecture

In particular, let us compare the common biggest dataset among the three architectures, which is *gisette_6*. The first implementation required 2.6 hours to produce the SVM Training model, whereas the second and improved second systems required 5.83 and 1.6 hours respectively. It is clear that the improved architecture is more efficient than all previous systems. The comparison between the first architecture and the final one shows that the modifications we studied and carried out actually improved our system. In particular, our final system was approximately 2 times faster than the initial one, due to the additional parallel units, the further utilization of the available bandwidth, the exploitation of LMem and the rearrangement of the order of the data. In addition, compared to the second architecture, the improved one is 4 times faster as we have overlapped the 14 clock cycle delay with useful pipeline computations.

Regarding the comparison between LIBSVM and our more efficient hardware implementation we observed that our overall system's performance was approximately 3 times slower. This number corresponds to the time required by both systems to complete the SVM Training phase. Thus, it does not reflect the exact time required to perform the kernel computation which is the task that was mapped on hardware. More specifically, it also includes the overhead of writing to the off-chip memory and of initializing the streams that are inputted into the FPGA. In fact, we calculated that the core that performs the kernel computation for small datasets *ijcnn1* was 10 times slower on hardware than on software, but for bigger datasets *gisette* this number approached 1.

At this point we will analyze why the hardware-based kernel computation did not surpass the respective software computation. Firstly, we stress out that in a software implementation efficient data structures can be used, whereas the same is not possible in a Maxeler system. More specifically, recall that LIBSVM transforms the input dataset into a list that contains all non-zero values, therefore efficiently depicting a sparse dataset. However, on Maxeler we need to declare a single constant size for our inputs, which is inevitably the maximum possible number of features in a data sample. This also determines the number of clock cycles required by the hardware side to yield the final outcomes. Thus, while on software operations are only performed between non-zero values, on hardware zero values are also considered which leads to a significant unavoidable overhead, especially for sparse datasets. Moreover, recall that 61% of the FPGA's space remains unexploited. Yet, we could not map additional parallel units (kernel computation cores) to the FPGA because the PCI express only allows 8 input and 8 output lanes to and from the host to hardware respectively. In case we could utilize the whole FPGA we would expect better performance due to more parallelism. In addition to the limited number of inputs and outputs, our system significantly approached the maximum possible bandwidth provided by a stream. More specifically, the PCI express used by MAX 3A Vectis provides 250 MB/s per lane. For dataset *gisette* we computed that 285 MB/s per lane were transferred. Note that this number exceeds the theoretical maximum one. This can be attributed to the fact that when certain streams remain unused, then the Maxeler controller utilizes them to increase bandwidth. In order for the hardware kernel computation time to reach the respective software one, instead of sending 258 MB/s we had to transfer around 400 MB/s and even more to achieve acceleration.

Finally, again for dataset *gisette* we compared the hardware processing time with the overall hardware execution time. These numbers were equal to 20 and 120 ms respectively. We know in advance that 5 to 30 ms are used for the DFE initialization process. Even if we assume that this overhead is 30 ms, there are another 70 ms which are uncorrelated with the processing. We assume that these additional ms reflect time required by the Maxeler memory controller and for other processes not known to the programmer. However, also notice that the actual hardware processing is 4 times faster than the respective software processing time, 20 and 80 ms respectively. This led to the conclusion that the hardware core processing time should dominate the overall hardware runtime to yield efficiency.

5.2 Mutual Information

5.2.1 Setup

We implemented parallel kernel computations 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. Connection to the host was via PCI express and all designs were compiled to run at 200 MHz. In the Mutual Information computation 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 result was correct. Furthermore, the reference software was of our own creation, but its proper functioning was evaluated by Maxeler Technologies with their own test cases.

5.2.2 Results

To begin with, this section presents the performance of the hardware-based MI calculation. Table 5.6 presents the results derived from the comparison between the Mutual Information (MI) software and the respective one-core hardware approach. In this comparison the hardware architecture contained a single MI calculation core, whereas in Table 5.7 another comparison takes place between the reference software and a two-core hardware implementation.

Resolution	**SW (sec)**	**HW (sec)**	**Speedup**
100	0.002	0.031	0.06
500	0.025	0.036	0.69
1000	0.095	0.046	2
2000	0.4	0.1	4
5000	2.5	0.45	5.6
10000	10.3	1.8	5.7
20000	41.5	7.1	5.8
40000	159	30.5	5.3

Table 5.6: Performance of One-Core MI Calculation Dataflow Architecture

Resolution	SW (sec)	HW (sec)	Speedup
100	0.002	0.036	0.05
500	0.025	0.037	0.68
1000	0.095	0.047	2
2000	0,4	0.077	5.2
5000	2.5	0.31	8
10000	10.3	1.1	9.4
20000	41.5	4.9	8.5
40000	159	19.7	8.1

Table 5.7: Performance of Two-Core MI Calculation Dataflow Architecture

Regarding the results of 5.6, one draws the conclusion that for a small histogram resolution, i.e. few bins, the software implementation is much faster than the hardware one. This is attributed to the 5 to 30 ms hardware call delay, depending on the sizes of the input and output streams. Similar observations were also presented in the experimental analysis of the hardware-based SVM system. Yet, the main difference compared to SVM is that now as the histogram resolution increases, i.e. the streams become bigger, the hardware performance also improves and surpasses the software implementation. This is due to the fact that the percentage of the hardware call gradually becomes a smaller and smaller fraction of the overall hardware runtime. Thus, unlike the results of the kernel computation 5.1.2.3 in this case the calculations performed on the FPGA are sufficiently computationally intensive. In this first comparison we only used one MI calculation hardware core in order to evaluate the potential of reaching better performance. Note that for the second biggest allowed resolution (20000) we achieved a hardware system 5.8 times faster compared to the software-based one, while for resolution equal to 1000 and further the hardware implementation yielded speedup. In our first hardware approach 3 out of 8 lanes of the PCI express were used and only 5% of the available FPGA resources.

As we have already mentioned several times, there are 8 lanes available for data transferring between the host and the DFE. In order to utilize more of the available bandwidth, the PDF estimations of random variables Y and (X,Y) were divided into two, which immediately led to the use of 5 streams 4.2.2.2. Increasing the number of data vectors that can be processed in parallel, led to the increase of the number of meaningful parallel units. More specifically, we doubled the number of processing units by using two MI cores instead of one, and therefore the hardware processing is also doubled.

The resource summary of our special-purpose architecture with two cores is shown in Table 5.8.Note that the 2-core implementation utilizes 10% of the available FPGA resources and transfers data by using 5 of the 8 available PCI lanes. Assigning more

	Used	**Maximum**	**Percentage**
LUTs	18887	297600	6.35%
FFs	22808	297600	7.66%
BRAMs	739	2128	2.77%
DSPs (Multipliers)	4	2016	0.20%

Table 5.8: MAX3A Vectis Resource Utilization for Mutual Information Computation with Two Cores

parallel processors to the hardware side affected the overall performance. Table 5.9 shows the comparison between the single and double core hardware-based approaches.

Resolution	**HW1 (sec)**	**HW2 (sec)**	**Speedup**
100	0.031	0.036	0.86
500	0.036	0.037	0.97
1000	0.046	0.047	0.98
2000	0.1	0.077	1.3
5000	0.45	0.31	1.45
10000	1.8	1.1	1.6
20000	7.1	4.9	1.45
40000	30.5	19.7	1.55

Table 5.9: Performance Comparison between One-Core and Two-Core MI Calculation Dataflow Architectures

We observe that for resolution smaller than 2000 bins the two-core based approach is slower than the one-core system. Recall that the former architecture requires inputting two additional streams compared to the latter one. Yet, the hardware initialization cost of these two additional streams leads to a noticeable overhead. However as resolution increases, the two-core approach performance reaches being approximately 1,5 times faster than the one-core approach, since for higher resolution, and thus more computational demands, the initialization streams' overhead becomes a small fraction of the total hardware runtime.

Moreover, in Table 5.10 we depict the transfer rates from the host to the DFEs. Note that the throughput presented in Table 5.10 is not the actual PCI throughput but the application's general throughput because the initialization of the DFE is also taken into account. For a small number of bins we observe that the two-core implementation reaches lower throughput compared to the one-core approach, as the hardware initialization overhead remains a significant percentage of the overall hardware execution time. Specifically in the two-core architecture, throughput reaches 900MB/sec, a number close to the theoretical PCI bandwidth that corresponds to 5 PCI lanes which is 1250MB/sec (250MB/sec per lane). The one-core implementation also approaches the theoretical PCI bandwidth for 3 streams, with the former being 670MB/sec and the latter being 750MB/sec.

Resolution	**HW1 (MB/sec)**	**HW2 (MB/sec)**
100	3.8	2.8
500	83	68
1000	261	213
2000	480	519
5000	667	806
10000	667	909
20000	676	816
40000	629	812

Table 5.10: Throughput comparison between architectures utilizing 3 and 5 streams.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis we created two basic dataflow architectures, the first being the kernel computation of the Support Vector Machines method, and the second being the calculation of the Mutual Information statistical value. During the process of designing and implementing these two architectures we explored and reached several conclusions about the Maxeler platform as well as about similar dataflow tools. To begin with, our experimental results showed that dataflow architectures are more likely to be efficient when they address a massive amount of data, that is over a gigabyte. Applications that do not deal with such sizes are more likely to achieve increased performance with the use of other platforms, such as GPUs and FPGAs. Moreover, dataflow computing is convenient to use only in algorithms that require little control logic. This is attributed to the fact that the system itself is designed to process streams of data using multiple parallel units, but not to control their flow. Furthermore, even though the Maxeler system offers large off-chip memory, this memory cannot be randomly accessed. Thus, such systems should be primarily addressed by applications that traverse the same massive amount of data continuously. Among our other findings, we also reached the conclusion that when creating a dataflow architecture the following factors should be definitely taken into account in the early stages of the development. In particular, the hardware initialization time should be negligible compared to the overall hardware execution runtime, the number of hardware calls should be reduced to the smallest possible number, the utilization of both the on-chip and off-chip memories allows faster data transfer rates, the system's pipeline should always be full, even if data rearrangement is necessary to avoid clock cycle delays, and the special-purpose platform should contain a big amount of simple processing elements in order for it to be occupied to the fullest. Finally, it is clear that the difficulty in using Maxeler 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 Maxeler supercomputer presents unpredictable behavior in certain simple functionalities that cannot be known in advance. In the case where all the aforementioned conclusions are taken into account, dataflow computing can easily yield increased performance and resolve problems that previously seemed non-resolvable.

6.2 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.

The Maxeler supercomputer platform provides four Xilinx Virtex 6 FPGAs. Due to the fact that the scope of this thesis was to focus on yielding efficiency by exploiting the properties of dataflow architectures we only used one of the four available FPGAs. We assume that by mapping our already implemented hardware cores to all the FPGAs we will achieve higher performances.

Furthermore, we described our two core hardware-based Mutual Information computation. It was not in the scope of this thesis to improve the accuracy of the MI calculation itself, but rather to utilize the findings derived from the hardware-based kernel computation efficiently. As future work we could focus on achieving further acceleration by assigning more parallel units to the FPGA and on exploring other methods for PDF estimation.

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] Feist, T.: Vivado design suite. White Paper (2012) 5

[2] Xu, J., Subramanian, N., Alessio, A., Hauck, S.: Impulse c vs. vhdl for accelerating tomographic reconstruction. In: Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on, IEEE (2010) 171–174 5

[3] Najjar, W.A., Lee, E.A., Gao, G.R.: Advances in the dataflow computational model. Parallel Computing **25**(13) (1999) 1907–1929 7

[4] Johnston, W.M., Hanna, J., Millar, R.J.: Advances in dataflow programming languages. ACM Computing Surveys (CSUR) **36**(1) (2004) 1–34 7

[5] Sousa, T.B.: Dataflow programming concept, languages and applications. In: Doctoral Symposium on Informatics Engineering. (2012) 7, 13

[6] Hurson, A.R., Kavi, K.M.: Dataflow computers: Their history and future. Wiley Encyclopedia of Computer Science and Engineering (2008) 7

[7] Dennis, J.B.: First version of a data flow procedure language. In: Programming Symposium, Springer (1974) 362–376 7

[8] Gostelow, K., et al.: The u-interpreter. Computer **15**(2) (1982) 42–49 7

[9] Davis, A.L., Keller, R.M.: Data flow program graphs. (1982) 7

[10] Whiting, P.G., et al.: A history of data-flow languages. Annals of the History of Computing, IEEE **16**(4) (1994) 38–59 9

[11] Ashcroft, E.A., Wadge, W.W.: Lucid, a nonprocedural language with iteration. Communications of the ACM **20**(7) (1977) 519–526 9

[12] Ashcroft, E.A., Wadge, W.W.: Lucid, the dataflow programming language. APIC Studies in Data Processing, Academic Press (1985) 9

[13] Ackerman, W.B.: Data flow languages. In: Managing Requirements Knowledge, International Workshop on, IEEE Computer Society (1899) 1087–1087 9

[14] Travis, J., Kring, J.: LabVIEW for Everyone: Graphical Programming Made Easy and Fun (National Instruments Virtual Instrumentation Series). Prentice Hall PTR (2006) 10

[15] Sjoholm, S., Lindh, L.: VHDL for Designers. Prentice Hall PTR (1997) 10

[16] Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language lustre. Proceedings of the IEEE **79**(9) (1991) 1305–1320 10

[17] Sharp, J.A.: Data flow computing: theory and practice. Intellect Books (1992) 13

[18] Hurson, A., Hurson, A., Lee, B., Lee, B.: Issues in dataflow computing. Adv. in Comput **37**(285-333) (1993) 38–39 13

[19] Yip, A., Wang, X., Zeldovich, N., Kaashoek, M.F.: Improving application security with data flow assertions. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ACM (2009) 291–304 13

[20] Pell, O., Averbukh, V.: Maximum performance computing with dataflow engines. Computing in Science & Engineering **14**(4) (2012) 98–103 16

[21] Grigoras, P., Luk, W., Weston, S.: Aspect oriented design for dataflow engines. (2013) 16

[22] Vapnik, V.: The nature of statistical learning theory. Springer Science & Business Media (2000) 16, 32

[23] Cortes, C., Vapnik, V.: Support-vector networks. Machine learning **20**(3) (1995) 273–297 16

[24] Furey, T.S., Cristianini, N., Duffy, N., Bednarski, D.W., Schummer, M., Haussler, D.: Support vector machine classification and validation of cancer tissue samples using microarray expression data. Bioinformatics **16**(10) (2000) 906–914 17

[25] Tong, S., Koller, D.: Support vector machine active learning with applications to text classification. The Journal of Machine Learning Research **2** (2002) 45–66 17

[26] Tong, S., Chang, E.: Support vector machine active learning for image retrieval. In: Proceedings of the ninth ACM international conference on Multimedia, ACM (2001) 107–118 17

[27] Hua, S., Sun, Z.: Support vector machine approach for protein subcellular localization prediction. Bioinformatics **17**(8) (2001) 721–728 17

[28] Cao, L.J., Tay, F.E.H.: Support vector machine with adaptive parameters in financial time series forecasting. Neural Networks, IEEE Transactions on **14**(6) (2003) 1506–1518 17

[29] Zien, A., Rätsch, G., Mika, S., Schölkopf, B., Lengauer, T., Müller, K.R.: Engineering support vector machine kernels that recognize translation initiation sites. Bioinformatics **16**(9) (2000) 799–807 17

[30] Schmidt, M.S.: Identifying speakers with support vector networks. Computing Science and Statistics (1997) 305–316 17

[31] Joachims, T.: Learning to classify text using support vector machines: Methods, theory and algorithms. Kluwer Academic Publishers (2002) 17, 30

[32] Osuna, E., Freund, R., Girosi, F.: Training support vector machines: an application to face detection. In: Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on, IEEE (1997) 130–136 17

[33] Boyd, S., Vandenberghe, L.: Convex optimization. Cambridge university press (2004) 23

[34] Campbell, C.: An introduction to kernel methods. Studies in Fuzziness and Soft Computing **66** (2001) 155–192 28

[35] Amari, S.i., Wu, S.: Improving support vector machine classifiers by modifying kernel functions. Neural Networks **12**(6) (1999) 783–789 28

[36] Gomes, T.A., Prudêncio, R.B., Soares, C., Rossi, A.L., Carvalho, A.: Combining meta-learning and search techniques to select parameters for support vector machines. Neurocomputing **75**(1) (2012) 3–13 28

[37] Staelin, C.: Parameter selection for support vector machines. Hewlett-Packard Company, Tech. Rep. HPL-2002-354R1 (2003) 28

[38] Soares, C., Brazdil, P.B., Kuba, P.: A meta-learning method to select the kernel width in support vector regression. Machine learning **54**(3) (2004) 195–209 28

[39] Jebara, T.: Multi-task feature and kernel selection for svms. In: Proceedings of the twenty-first international conference on Machine learning, ACM (2004) 55 28

[40] Ali, S., Smith-Miles, K.A.: A meta-learning approach to automatic kernel selection for support vector machines. Neurocomputing **70**(1) (2006) 173–186 28

[41] Li, C.H., Lin, C.T., Kuo, B.C., Ho, H.H.: An automatic method for selecting the parameter of the normalized kernel function to support vector machines. In: Technologies and Applications of Artificial Intelligence (TAAI), 2010 International Conference on, IEEE (2010) 226–232 28

[42] Keerthi, S.S., Lin, C.J.: Asymptotic behaviors of support vector machines with gaussian kernel. Neural computation **15**(7) (2003) 1667–1689 30

[43] Jaakkola, T., Haussler, D., et al.: Exploiting generative models in discriminative classifiers. Advances in neural information processing systems (1999) 487–493 30

[44] Kondor, R.I., Lafferty, J.: Diffusion kernels on graphs and other discrete input spaces. In: ICML. Volume 2. (2002) 315–322 30

[45] Shawe-Taylor, J., Cristianini, N.: Kernel methods for pattern analysis. Cambridge university press (2004) 30

[46] Smola, A.J., Schölkopf, B.: Learning with kernels. Citeseer (1998) 30

[47] Hofmann, M.: Support vector machinesb•"kernels and the kernel trick. Hauptseminar report (2006) 32

[48] Burges, C.J.: A tutorial on support vector machines for pattern recognition. Data mining and knowledge discovery **2**(2) (1998) 121–167 32

[49] Chang, C.C., Lin, C.J.: Libsvm: a library for support vector machines. ACM Transactions on Intelligent Systems and Technology (TIST) **2**(3) (2011) 27 32, 35, 59, 66

[50] Joachims, T.: Making large scale svm learning practical. Technical report, Universität Dortmund (1999) 32

[51] Grauman, K., Darrell, T.: The pyramid match kernel: Discriminative classification with sets of image features. In: Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on. Volume 2., IEEE (2005) 1458–1465 32

[52] Nivre, J., Hall, J., Nilsson, J., Chanev, A., Eryigit, G., Kübler, S., Marinov, S., Marsi, E.: Maltparser: A language-independent system for data-driven dependency parsing. Natural Language Engineering **13**(02) (2007) 95–135 32

[53] Hanke, M., Halchenko, Y.O., Sederberg, P.B., Hanson, S.J., Haxby, J.V., Pollmann, S.: Pymvpa: A python toolbox for multivariate pattern analysis of fmri data. Neuroinformatics **7**(1) (2009) 37–53 32

[54] Dorff, K.C., Chambwe, N., Srdanovic, M., Campagne, F.: Bdval: reproducible large-scale predictive model development and validation in high-throughput datasets. Bioinformatics **26**(19) (2010) 2472–2473 32

[55] Allen, D.M.: Mean square error of prediction as a criterion for selecting variables. Technometrics **13**(3) (1971) 469–475 36

[56] Lee Rodgers, J., Nicewander, W.A.: Thirteen ways to look at the correlation coefficient. The American Statistician **42**(1) (1988) 59–66 36

[57] Cover, T.M., Thomas, J.A.: Entropy, relative entropy and mutual information. Elements of Information Theory (1991) 12–49 36

[58] Cover, T.M., Thomas, J.A.: Elements of information theory. John Wiley & Sons (2012) 36

[59] Shannon, C.E.: Prediction and entropy of printed english. Bell system technical journal **30**(1) (1951) 50–64 36

[60] Kraskov, A., Stögbauer, H., Andrzejak, R.G., Grassberger, P.: Hierarchical clustering using mutual information. EPL (Europhysics Letters) **70**(2) (2005) 278 36

[61] Shannon, C.E.: A mathematical theory of communication. ACM SIGMOBILE Mobile Computing and Communications Review **5**(1) (2001) 3–55 36

[62] Anguita, D., Boni, A., Ridella, S.: A digital architecture for support vector machines: theory, algorithm, and fpga implementation. Neural Networks, IEEE Transactions on **14**(5) (2003) 993–1009 45, 49

[63] Cadambi, S., Durdanovic, I., Jakkula, V., Sankaradass, M., Cosatto, E., Chakradhar, S., Graf, H.P.: A massively parallel fpga-based coprocessor for support vector machines. In: Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on, IEEE (2009) 115–122 46, 48

[64] Pedersen, R., Schoeberl, M.: An embedded support vector machine. In: Intelligent Solutions in Embedded Systems, 2006 International Workshop on, IEEE (2006) 1–11 48

[65] Kyrkou, C., Theocharides, T., Bouganis, C.S.: An embedded hardware-efficient architecture for real-time cascade support vector machine classification. In: Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on, IEEE (2013) 129–136 48

[66] Papadonikolakis, M., Bouganis, C.: Novel cascade fpga accelerator for support vector machines classification. Neural Networks and Learning Systems, IEEE Transactions on **23**(7) (2012) 1040–1052 48, 49

[67] Papadonikolakis, M., Bouganis, C.: A heterogeneous fpga architecture for support vector machine training. In: Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on, IEEE (2010) 211–214 48, 49

[68] Khan, F.M., Arnold, M.G., Pottenger, W.M.: Hardware-based support vector machine classification in logarithmic number systems. In: IEEE International Symposium on circuits and systems. Volume 5., Citeseer (2005) 5154 49

[69] Irick, K.M., DeBole, M., Narayanan, V., Gayasen, A.: A hardware efficient support vector machine architecture for fpga. In: Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on, IEEE (2008) 304–305 49

[70] Ramos-Lara, R., López-García, M., Cantó-Navarro, E., Puente-Rodriguez, L.: Svm speaker verification system based on a low-cost fpga. In: Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on, IEEE (2009) 582–586 50

[71] Ruiz-Llata, M., Guarnizo, G., Yébenes-Calvino, M.: Fpga implementation of a support vector machine for classification and regression. In: Neural Networks (IJCNN), The 2010 International Joint Conference on, IEEE (2010) 1–5 50

[72] Catanzaro, B., Sundaram, N., Keutzer, K.: Fast support vector machine training and classification on graphics processors. In: Proceedings of the 25th international conference on Machine learning, ACM (2008) 104–111 50

[73] Carpenter, A.: cusvm: A cuda implementation of support vector classification and regression. patternsonscreen. net/cuSVMDesc. pdf (2009) 51

[74] Herrero-Lopez, S., Williams, J.R., Sanchez, A.: Parallel multiclass classification using svms on gpus. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, ACM (2010) 2–11 51

[75] Cotter, A., Srebro, N., Keshet, J.: A gpu-tailored approach for training kernelized svms. In: Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM (2011) 805–813 52

[76] Do, T.N., Nguyen, V.H., Poulet, F.: Speed up svm algorithm for massive classification tasks. In: Advanced Data Mining and Applications. Springer (2008) 147–157 52

[77] Collobert, R., Bengio, S., Bengio, Y.: A parallel mixture of svms for very large scale problems. Neural computation **14**(5) (2002) 1105–1114 53

[78] Athanasopoulos, A., Dimou, A., Mezaris, V., Kompatsiaris, I.: Gpu acceleration for support vector machines. In: WIAMIS 2011: 12th International Workshop on Image Analysis for Multimedia Interactive Services, Delft, The Netherlands, April 13-15, 2011, TU Delft; EWI; MM; PRB (2011) 53

[79] Dey, S., Kedia, M., Agarwal, N., Basu, A.: Embedded support vector machine: Architectural enhancements and evaluation. In: VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., 20th International Conference on, IEEE (2007) 685–690 53

[80] Graf, H.P., Cosatto, E., Bottou, L., Dourdanovic, I., Vapnik, V.: Parallel support vector machines: The cascade svm. In: Advances in neural information processing systems. (2004) 521–528 53

[81] Cao, L.J., Keerthi, S.S., Ong, C.J., Zhang, J.Q., Lee, H.P.: Parallel sequential minimal optimization for the training of support vector machines. Neural Networks, IEEE Transactions on **17**(4) (2006) 1039–1049 54

[82] Zhu, K., Wang, H., Bai, H., Li, J., Qiu, Z., Cui, H., Chang, E.Y.: Parallelizing support vector machines on distributed computers. In: Advances in Neural Information Processing Systems. (2008) 257–264 54

[83] Zhao, H., Magoules, F.: Parallel support vector machines on multi-core and multiprocessor systems. In: 11th International Conference on Artificial Intelligence and Applications (AIA 2011), IASTED (2011) 55

[84] Chu, C., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G., Ng, A.Y., Olukotun, K.: Mapreduce for machine learning on multicore. Advances in neural information processing systems **19** (2007) 281 55

[85] Brugger, D.: Parallel support vector machines. (2006) 55

[86] Castro-Pareja, C.R., Shekhar, R.: Hardware acceleration of mutual information-based 3d image registration. Journal of Imaging Science and Technology **49**(2) (2005) 105–113 55

[87] Shao, S., Guo, C., Luk, W., Weston, S.: Accelerating transfer entropy computation. In: Field-Programmable Technology (FPT), 2014 International Conference on, IEEE (2014) 60–67 56, 57

[88] Shams, R., 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) 555–560 56

[89] Lin, Y., 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) 1–6 57

[90] Teßmann, M., Eisenacher, C., Enders, F., Stamminger, M., Hastreiter, P.: Gpu accelerated normalized mutual information and b-spline transformation. In: VCBM. (2008) 117–124 57

[91] Shams, R., Sadeghi, P., Kennedy, R., Hartley, R.: Parallel computation of mutual information on the gpu with application to real-time registration of 3d medical images. Computer methods and programs in biomedicine **99**(2) (2010) 133–146 57

[92] Platt, J., et al.: Fast training of support vector machines using sequential minimal optimization. Advances in kernel methodsb•"support vector learning **3** (1999) 64, 66

[93] Guyon, I., Gunn, S., Ben-Hur, A., Dror, G.: Result analysis of the nips 2003 feature selection challenge. In: Advances in Neural Information Processing Systems. (2004) 545–552 73, 97

[94] Prokhorov, D.: Ijcnn 2001 neural network competition. Slide presentation in IJCNN **1** (2001) 97

[95] Alon, U., Barkai, N., Notterman, D.A., Gish, K., Ybarra, S., Mack, D., Levine, A.J.: Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide arrays. Proceedings of the National Academy of Sciences **96**(12) (1999) 6745–6750 97