www.qualiMaster.eu

QualiMaster

A configurable real-time Data Processing Infrastructure mastering
autonomous Quality Adaptation

Grant Agreement No. 619525

Deliverable D3.1

Work-package	WP3: **Optimized Translation to Hardware**
Deliverable	D3.1: Translation of Data Processing Algorithms to Hardware
Deliverable Leader	Telecommunication System Institute
Quality Assessor	H. Eichelberger
Estimation of PM spent	6
Dissemination level	Public (PU)
Delivery date in Annex I	31/12/2014
Actual delivery date	31/12/2014
Revisions	4
Status	Final
Keywords:	QualiMaster, Adaptive Pipeline, Reconfigurable Computing, FPGA Computing, Hardware, Support Vector Machines (SVM), Latent Dirichlet Analysis (LDA), Count Min, Exponential Histogram, Hayashi-Yoshida Correlation Estimator

Disclaimer

List of Authors

Partner Acronym	Authors
TSI	E. Sotiriades, G. Chrysos, P. Malakonakis, N. Pavlakis, S. M. Nikolakaki, I. Papaefstathiou, A. Dollas.
MAX	J.B. Robertson, O. Pell
LUH	S. Zerr
SUH	C. Qin

Table of Contents

Executive summary

Field Programmable Gate Array (FPGA) computers (also known as reconfigurable computers) are a special category of computer hardware, in which algorithms are directly translated into hardware designs, rather than being implemented as software which runs on general purpose computers. Despite FPGA technology being almost 30 years old, the process of algorithm mapping to FPGA supercomputers is no trivial task because it entails one-of-a-kind hardware designs. In the QualiMaster project, we aim at using such FPGA-based computers as hardware accelerators offer various elements in the adaptive pipeline. This entails identification of algorithms which are suitable for hardware implementation, interfaces with existing software platforms and tools for seamless operation, hardware implementation of the chosen algorithms, and performance evaluation to quantify the performance benefits and the cost-performance tradeoffs from this approach. This deliverable reports on the methodology and the design phase of the specialized hardware of the QualiMaster adaptive pipeline for several Data Processing Algorithms and it follows Task 3.1 which comprises of three important subtasks, per the Description of Work. In the Introduction Section, below, the progress in these three subtasks is summarized, with the body of this deliverable elaborating on progress in each one of these subtasks.

1 Introduction

The QualiMaster project includes the use of reconfigurable hardware to boost its performance and to operate in real-time problems which conventional computers cannot address adequately. Reconfigurable computing (also known as FPGA-based computing) is the field in which algorithms are mapped directly to hardware resources for execution. As a rule-of-thumb, reconfigurable computers run at clock speeds which are ten times slower vs. conventional computers, but if the available parallelism is high and the granularity of computation vs. Input/Output requirements is high, this form of computing may offer substantial speedups vs. conventional computers. The use of this form of computing in the processing of Big Data seems promising. In the QualiMaster project the goal is to exploit reconfigurable computing in the QualiMaster pipeline, in order to have real-time processing of streaming data.

This Work Package and this deliverable D3.1 are directly connected to WP2. WP2 identifies the important classes of problems and algorithms for the QualiMaster project. In the present Deliverable D3.1 several of these algorithms, from all classes of problems, were selected for translation into hardware. Specifically, five different algorithms were selected and an in-depth study was made for each of them. This study shows that four of them are suitable for mapping at reconfigurable computers and can offer significant speedup. According to this study, the hardware designs of these algorithms were made, considering the restrictions of WP4 in which the pipeline configuration is designed.

The progress in WP3 which is reported in D3.1 can be summarized below:

1.1 Algorithmic classes for acceleration via Hardware

Following extensive collaboration with all partners and especially those involved in WP2, several classes of algorithms were identified and studied. These algorithms are Support Vector Machines (SVM), Latent Dirichlet Analysis (LDA), Count Min, Exponential Histogram, and Hayashi-Yoshida Correlation Estimator. All these classes of algorithms were methodically evaluated. All of these algorithms were chosen by the software partners and studied by the hardware partners of the QualiMaster project, so that the results would be relevant to the project goals.An important contribution of the QualiMaster project is that through interaction of the software and hardware design teams of the project, the algorithms that are chosen by the software team form a real-life area in which the hardware team needs to apply its expertise, and likewise, the capabilities of the specialized reconfigurable hardware allow for the software team to have an alternative computing paradigm, which would not be possible to consider otherwise.

1.2 Guidelines translating algorithms to FPGAs

All of the above algorithms were profiled with respect to computational characteristics, available parallelism, Input/Output (I/O) requirements, and suitability for hardware implementation. This is the standard practice within the hardware community, as it may lead to the optimization of the computationally intensive part, or computationally equivalent mathematical transformations to lead into more hardware parallelizable versions of the algorithm. One algorithm proved to be less suitable for translation to hardware (LDA) whereas all others proved to be highly suitable (e.g. Count Min, Exponential Histogram) or moderately suitable (e.g. SVM). Subsequently, the process of developing hardware modules to interact with the software within the QualiMaster pipeline progressed.

1.3 Interaction with WP2 and WP4

Whereas this subtask is complementary to subtasks 1 and 2 (see above), it also entails the significant issue of how to connect the specialized Field Programmable Gate Array- based supercomputing nodes (made by Maxeler in this case) to the environment used by the software community in order to form a seamless QualiMaster pipeline. Following the examination of several alternatives, and in cooperation of WP2 and WP4 partners, we jointly developed the communication interfaces and libraries to make the Maxeler system become a node of the STORM distributed environment. This is a significant step which has been fully completed and is currently operational, as it allows for the QualiMaster pipeline to run seamlessly either on software platforms alone (made of distributed computational nodes) or combined software/hardware platforms, comprising of the aforementioned environment plus the Maxeler specialized hardware node. This QualiMaster platform allowed for the detailed study of communication overhead, data rates, and software/hardware interaction which was crucial in the final determination of the algorithms which are suitable for further hardware development.

This Deliverable introduces the fundamentals of reconfigurable computing, the QualiMaster pipeline, and the algorithms selection in Section 2. Section 3 presents the reconfigurable infrastructure and its integration to the rest QualiMaster infrastructure. Section 4 makes a brief presentation of the selected algorithms as they are extensively presented in D2.1. Section 5 reports the study methodology, and the studies for each of the five algorithms in order to map it to reconfigurable computing. Finally Section 6 shows how algorithms are translated to hardware designs, both in terms of the methodology and how this methodology is applied specifically on each of the selected algorithms.

2 Identification of classes of algorithms or tasks from WP2

2.1 Reconfigurable Hardware Technology

Reconfigurable Computing was introduced at the late 1980s, when the first Field Programmable Gate Arrays (FPGA) chips were designed. These chips, where the technology evolvement of the CPLDs and other older programmable devices, using different implementation technology. The main computational paradigm in this form of computing is that the hardware resources (logic gates, memories, digital signal processing – DSP blocks, etc.) are connected after power up in an application-specific way, with a design which was created earlier for that purpose. The designer maps algorithms directly to hardware, exploiting parallelism and datapaths which are not available in conventional general-purpose computing. Using even early FPGAs several computationally intensive problems have been mapped and proved that FPGA computing, or reconfigurable computing, can be a solution for performance boosting of several algorithms. Eventually, FPGAs proved not only to be a cost effective rapid system prototyping platform vs. ASIC, but a versatile technology of choice for Image Processing, Data encryption (eg. The RSA and DES algorithms), Video Processing, String Pattern Matching, FFT Implementations, Data Compression – to name a few.

Later generation devices offered significant resources in addition to the reconfigurable fabric. Special I/O transceivers, dedicated logic blocks for memory, powerful general purpose processors on chip, special modules for digital signal processing, and fast floating point operations were added on the device. Even the reconfigurable fabric had changed, offering more logic, better routing resources and run time reconfiguration characteristics. In addition, a large collection of functional Intellectual Property cores (IPs) is freely available to the designer through IP generator tools such as the Xilinx Core Generator, or, distributed by designers through web sites such as OpenCores. All these available resources help designers to take up with new applications, with considerable results on network systems such as network switches, network intrusion detection systems, financial data analysis. Data streaming applications become much more significant due to these technological advances of FPGAs, mostly in the forms of I/O transceivers on a chip and large amount of available memory.

Nowadays multi FPGA platforms have been developed offering opportunities at system level design, using powerful General Purpose Processors with fast interconnection with FPGAs. FPGAs also have ultra fast access to external memory and direct fast connection to the internet. These systems have a "look and feel" of a conventional General Purpose Server with a Linux -based operating system, using special compilers. Designers usually keep the official software at its original form and change only the computational intensive procedures, with hardware procedures calls which are functional equivalent. These servers can become nodes of greater systems, as QualiMaster projects intents, and reconfigurable computing can be easily integrated with conventional computing. The reconfigurable node performs the compute intensive parts of the algorithm and the conventional nodes perform all the other procedures which are difficult to be translated in hardware and have not significant computational load. This is considered to be a new era for reconfigurable computing which can easily incorporate heterogeneous computing systems providing them with powerful coprocessors.

2.2 QualiMaster Pipeline

A core concept of QualiMaster is the notion of the adaptive data processing pipeline. Basically, a data processing pipeline defines the data flow from data sources through data processing

algorithms to data sinks. It is noteworthy that a processing pipeline defines the data flow among the elements of a pipeline rather than the control flow of the analysis algorithms.

Data sources produce data in a certain form, such as structured financial stock market data or unstructured Twitter data. While a data source can be characterized by technical information, such as the provided data fields or access credentials, an Infrastructure User (as introduced in D1.2) can also specify SLAs (Service Level Agreements) negotiated with the Data Provider to also consider deviations or violations of these agreements during adaptation. One extreme example on handling data source violations (if negotiated with the user) could be to notify the user that the SLAs on user side cannot be met as the input side does not fulfill its SLAs.

Starting at the data sources, a data processing pipeline links then data sources with data processing elements and, finally, data sinks. In QualiMaster, data processing elements exist in three types, namely elements representing processing families, generic stream processing operators and data management operations.

Processing families represent a set of algorithms performing the same task at different quality tradeoffs as described in D1.2 and detailed in D2.1 and D4.1. Changing the actual algorithm of a certain family or its functional parameter settings at runtime is the core idea of realizing adaptive data processing pipelines. Further, processing families enable the seamless integration of software-based and hardware-based execution on reconfigurable hardware. However, data processing algorithms can be more than just simple software components, e.g., representing a single processing algorithm possibly depending on a set of supporting libraries. In particular, processing algorithms can be realized in terms of reconfigurable hardware as described in this deliverable. Furthermore, a data processing algorithm can be implemented as a complex, already distributed stream processing algorithm such as the distributed correlation computation described in D2.1. Switching among these different kinds of algorithms requires specific strategies in order to consider the different functional and structural aspects as well as to use the overall resource pool in an optimal way.

Running a single algorithm on reconfigurable hardware may not be optimal for different reasons, including use of reconfigurable hardware resources, performance gain, or communication overhead. One specific optimization form is to lay out sub-pipelines of a data processing pipeline (if the translation to hardware is feasible) on the same hardware resource or within a cluster of reconfigurable hardware resources as provided by the Maxeler infrastructure (MaxelerOS). In turn, this may lead to dependencies during the design of a data processing pipeline (as the sequence must be met) as well as during its dynamic execution.

In contrast to user-defined data processing elements as handled by the processing families, generic stream processing operators, such as filter, project, fork or join are frequently considered in literature (e.g., [1, 2, 3, 4]). Generic stream processing operators can ease the adaptation (as the can be included in a domain-independent version of the QualiMaster infrastructure), simplify the definition of a data processing pipeline and facilitate reuse of generic functionality. In order to be applied, functional parameters need to be defined during the pipeline design, e.g., on what to project. These functional parameters can be specified during configuration of the pipelines. Although the QualiMaster consortium is aware of the need for such operators (e.g., D4.1 already provides an extension point for such operators), we currently focus on the more challenging user-defined processing families, handle forks and joins implicitly and will consider generic stream processing operators as part of future work.

Data Management operators as introduced in D5.1 support storing intermediary processing results for later (batch) analysis by the Data Management Layer at any point of a data processing pipeline.

As the Data Management operator is generic, it can be configured through functional parameters in order to perform the correct functionality in the actual pipeline. It is important that the data at the source and the sink may be stored transparently through the Data Management Layer in order to avoid storing the same data item multiple times by multiple pipelines.

A data processing pipeline can be considered as linear or sequential, i.e., the data flow links have data processing elements including stream forks and joins. However, if supported by the underlying Execution System, a data processing pipeline may also contain cycles in terms of feedback loops, i.e., processed data is fed back into a previous processing step, e.g., to improve the processing model. In this case, a data processing pipeline can be considered as a data processing network or graph.

The design of a data processing pipeline may specify quality constraints on the individual data processing elements, e.g., to guard extremely important processing steps as well as on data flows to provide guidelines for switching among explicit alternatives. In QualiMaster, These constraints fulfill two purposes, the first supporting the user, the second the QualiMaster consortium. On the user side, such constraints allow narrowing down the adaptation space and, if needed to specify explicit alternatives to be taken dynamically based on the constraint. However, we are aware of the fact that specifying constraints increases the specification effort and requires more knowledge about data processing. Thus, we focus currently more on the processing families rather than on the pipeline constraints. On the other side such constraints allow to change the scope of experiments, i.e., using a pipeline specification for multiple purposes by just modifying some experiment-specific settings.

Ultimately, the data produced by the entire data processing pipeline is directed to data sinks, i.e., the endpoints of a data processing pipeline. Multiple data sinks may provide different forms or qualities of output, e.g., data sinks may offer different levels of quality and, depending on the business model of the infrastructure/data analysis provider, possibly also at different levels of pricing. Akin to data sources, data sinks can be detailed by SLAs in order to reflect the negotiated client side quality. Further, data sources may be supplemented with technical access information, e.g., in order to protect the output data as well as different levels of result quality. Finally, a kind of web service realizes the data sink from a technical perspective in order to make the data available to the QualiMaster applications. Thereby, the (realization of the) data sink will not act as a one-way service, as it needs to provide an interface to communicate with the QualiMaster infrastructure, in particular to react on user triggers to be considered by the adaptation (see D1.2).

2.3 Classes of Problems and Algorithms Selection

Data processing with the QualiMaster pipelines includes several different steps, as shown in Figure 1 describes a possible execution scheme of the QualiMaster pipeline, as it was presented in D 1.1. Most of the computational elements shown in this Figure can be efficiently mapped to hardware. Data Reformatting, Data Filtering, Data Classification, Streaming Computation, Correlation, Transfer Entropy, Granger Causality, Data Clustering, Graph Analysis, Data Synopsis and Sentimental Analysis are classes of problems that have been mapped efficiently to reconfigurable hardware and have been reported in the relevant literature on several occasions. However, this does not make the integration of the reconfigurable hardware with the software an easy task. The implementation of single computational elements with reconfigurable computing resources, as reported in the literature, not only is based on specific assumptions regarding Input/Output, dataset size, and algorithm accuracy, to name a few, but even if such were not significant issues (and they are), the QualiMaster project needs to address effective use of the specialized reconfigurable hardware resources in order to optimize the entire pipeline. To illustrate,

two possible scenarios in which the end result would lead to a slowdown at the system level are presented.

Scenario A: if algorithm A runs exceptionally well on reconfigurable hardware (e.g. 100 times faster than software execution of the same algorithm) but it expects results from algorithm B which runs on software, and algorithm B does not produce data at a sufficiently high rate, then not only the potential of the reconfigurable hardware is not realized, but the Input/Output time overhead to move data in and out of the reconfigurable hardware may dominate over the computational time benefits, leading to an actual slowdown. Referring to Figure 1, if the Fast Codepedency Filter (see step 8) is run in software and Correlation (see step 9) is run in reconfigurable hardware, the overall performance of the QualiMaster pipeline will not only depend on the performance of the individual elements of the pipeline, but also their integration.

Scenario B: If separate elements of the pipeline run well on the hardware, it is possible that it is beneficial to run non-optimal elements as well, in order to avoid Input/Output overhead. Continuing from scenario A, if in order to address the problems which were describe above we run forth the Fast Codependency Filter and the Correlation in hardware, it is possible that we will also need to run in hardware the Transfer Entropy and the Element Causality as well (both are in Step 8 of the pipeline), in order to avoid excessive Input/Output of the (partial in this case) results of the Fast Codependency Filter and Correlation pipeline from having to be forwarded to software, potentially causing an Input/Output bottleneck. However, even in this case, it is not clear that the hardware has enough resources to run all of the above.

As a result from the considerations, above, different elements of the QualiMaster pipeline need to be implemented in hardware, benchmarked, and evaluated vis a vis the same elements running on software, so that there will be a system-level optimization. The only way to achieve this goal is to have many different potential implementations of the QualiMaster pipeline and many different forms of data (e.g. twitter, financial data), as the optimal solution in one case might be undesirable in another. Hence the QualiMaster pipeline will be truly adaptive to the workload, desired processing algorithms, and available hardware resources (in principle there can be more than one reconfigurable hardware nodes in the system).

In this context, WP3 interacted with WP2, in order to find out the classes of problems and the algorithms which are relevant for QualiMaster, in particular in the context of the priority pipeline reconfigurable platform elements. The algorithm families of interest are identified as:

- Data Classification
- Sentiment Analysis
- Data Synopsis
- Correlation Estimation

The above classes of problems seem to be closely connected to the goals of the QualiMaster project. The above problems are very computationally intensive, thus the acceleration of these workloads would be a benefit to the final QualiMaster infrastructure. There is substantial related work on similar classes of problems, where hardware-based accelerators have been proposed. Several algorithms were proposed in collaboration with WP2 for each class of desirable algorithms.

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”

classes based on its correlation with a general subject, i.e. finance, news, or social. Given that the scope of the project is to receive Twitter data with a view of performing efficient risk and Sentiment Analysis, categorizing streamed text information for its effective use is crucial for QualiMaster. In addition, Sentiment Analysis refers to the use of natural language processing and text analysis to identify and extract subjective information from source materials. In the QualiMaster project extracting sensible and related to financial knowledge is significant for precise and accurate risk analysis. Furthermore, Data Synopsis focuses on configuring data structures and algorithms for efficiently processing and storing massive datasets or swiftly arriving data. The use of synopses allows fast response times to queries on big datasets, a function necessary to QualiMaster as efficient handling of streaming financial data (commodities) is essential. Finally, Correlation Estimation is a method that reports the dependence between two random variables. Thus, in QualiMaster correlation is used to determine the dependence between commodities.

Implementing our own text classifiers by hand would be time-consuming and could be quite difficult. In general, text comprises several aspects that need to be taken into consideration, such as the high dimensional input space, linearly separability and few irrelevant features. Thus, in accordance with WP2 and by studying the related literature we concluded that the Support Vector Machines (SVM) method is appropriate for text classification.

Various machine learning algorithms, which are used for data regression and classification, have been introduced at D 2.1 and proposed for hardware acceleration. We studied thoroughly the algorithms about Linear Regression, Bayesian Linear Regression, Support Vector Machines and Linear Generative Classification. The Support Vector Machines (SVM) algorithm was selected as the most common one and with parallel processing characteristics to be implemented for the data classification problem.

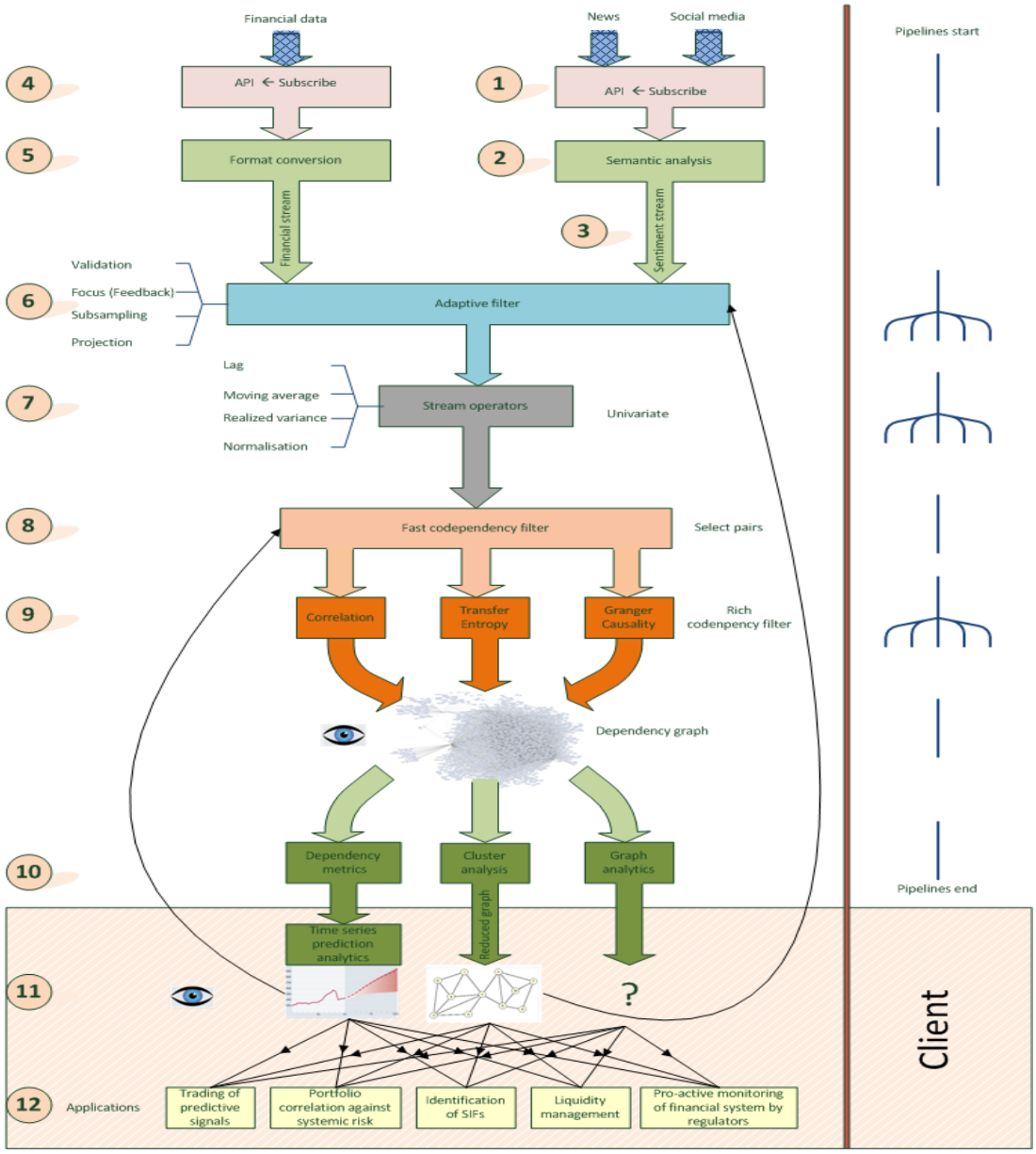


Figure 1: QualiMaster pipeline example execution steps

For the sentiment analysis the LDA algorithm was chosen to be implemented in hardware, and more specifically its training phase as it is the most time consuming part of the algorithm. LDA is an algorithm used in order to classify a set of documents into topics. The topics can contain opinion targets as well as the polarity of the opinion. This allows sentiment analysis to exact information about the main theme of the document, as well as polarity, by examining the topics it is associated with (taking into account the probabilities for each topic). Topic modeling is a basic step for the sentiment analysis of documents. Moreover, topic modeling is a common method, which is used in machine learning and natural language processing. Atopic model is a type of statistical model for discovering the abstract "topics" that occur in a collection of documents. The most widely used algorithm is LDA, which is also the one used by WP2. In LDA algorithm, each document may be viewed as a mixture of various topics. This is similar to probabilistic latent semantic analysis (pLSA). In addition, the topic distribution in LDA algorithm is assumed to have a Dirichlet prior probability distribution. In practice, this results to more reasonable mixtures of topics in a document.

The large volume of data streams poses unique space and time constraints on the computation process. There are problems that may be acceptable to generate approximate solutions for problems with streaming huge volumes of data. A number of synopsis structures have been developed, which can be used in conjunction with a variety of mining and query processing techniques in data stream processing. Sketches are one of the simplest data structures that are used for data synopses. Count-Min sketch is a probabilistic data structure that serves as a frequency table of events in a stream of data. Another basic data structure, which is used for data synopsis, is the histogram. There is a great variety of algorithms that create histogram data synopsis from streaming data. Exponential Histograms is an efficient data structure that enables answering frequency queries over streaming data. In addition, the ECM sketch is a data structure that combines the CM sketch with the Exponential Histogram data structure to an application for efficient querying over sliding window data streams. The ECM is considered to be a more sophisticated way of data synopsis. As the ECM method is a combination of the Count-Min algorithm and the Exponential Histogram data structure both of them were proposed to be developed. Lastly, these implementations will be combined in order to implement the efficient mapping of the ECM sketch on reconfigurable hardware. The ECM method takes as input a stream of elements and updates the corresponding CM sketch data structure. The CM sketch is updated by keeping an order-preserving aggregation of all streams. The update of the data structure takes place in different buckets of the data structure thus it needs a lot of CPU clock cycles in order to be updated. According to the theory the update of the ECM has O(1) amortized time complexity. A pipelined reconfigurable system that maps the update function will offer O(1) time complexity in any case. In addition, the small memory footprint of the CM data structure leads to a system that maps such data structure internally in an FPGA device offering high throughput in case of CM updating or CM querying.

One of the main goals of the QualiMaster project is the use of methods that will monitor data streams for event detection. A statistical technique that can show whether and how strongly pairs of variables are related is the correlation measure. In the world of finance, financial correlations measure the relationship between the changes of two or more financial variables, e.g. stock prices, in time. Financial correlation measurements are considered very important and they are used in advance portfolio management. The Hayashi-Yoshida cross-correlation estimator is an important estimator of the linear correlation coefficient between two asynchronous diffusive processes, e.g. stock market transactions. This method is really important as it can correlate high-frequency financial assets. Also, the Hayashi-Yoshida estimator correlates the variables, e.g. the stock prices, in real time, which is really important for forecasting the mid quote variation of the corresponding values. Lastly, the Hayashi-Yoshida estimator is considered as one of the most important and most basic estimators over streaming data, thus was proposed for implementation at the QualiMaster Use Cases as they have indicated at D1.1.

The correlation matrix algorithm can calculate the correlation between multiple commodities in parallel. Also, the calculation of a single correlation estimator can be processed in parallel using the corresponding data. Reconfigurable hardware can offer high parallelization levels by using different resources in parallel or/and in a pipelined architecture.

3 Interface of Reconfigurable Hardware with QualiMaster Platform

In this section there is a short description of the target platform (a Maxeler C-Series FPGA-based supercomputing node) for the algorithms mapped to hardware and the interface with the project platform (Storm) which is the framework for the project.

3.1 Maxeler Technology

Maximum Performance Computing (MPC) changes the classical computer science optimization from ease-of-programming to maximizing performance and minimizing total cost of computing. Performance is optimized by constructing compute engines to generate one result per clock cycle, wherever possible. Ease-of-programming is still important but takes second place to performance, computational density and power consumption. As such, MPC focuses on mission critical, long running computations with large datasets and complex numerical and intense statistical content. Maxeler drives MPC via 'Multiscale Dataflow Computing'. This section summarises the components of MPC, illustrates how MPC dataflow computers are programmed and how the resulting tools are presented to the users. An overview of MPC, along with detailed examples of applications, is available in [27, 28].

One Maxeler Dataflow Engine (DFE) combines 10^4 arithmetic units with 10^7 bytes of local fast SRAM and 10^11 bytes of 6-channel large DRAM. MaxelerOS allows the DFEs and CPU to run in parallel, so while the DFEs are processing the data, the CPU typically performs the non-time-critical parts of an application.

Maxeler's MPC programming environment comprises of several components:

- MaxCompiler, a meta-programming library used to produce DFE configurations by way of the MaxJ programming language, which is an extended form of Java with operator overloading. The compute kernels handling the data-intensive part of the application and the associated manager, which orchestrates data movement within the DFE, are written using this language. MaxJ is a Hardware Description language which produces the computational intensive part of the design configuration.
- the SLiC (Simple Live CPU) interface, which is Maxeler's application programming interface for CPU-DFE integration;
- MaxelerOS, a software layer between the SLiC interface, operating system and hardware, which manages DFE hardware and CPU-DFE communication in a way transparent to the user;
- MaxIDE, a specialised integrated development environment for MaxJ and DFE design, a fast DFE software simulator and a comprehensive debug environment used during development.

These components and their use are described in details in [29, 30].

In such an MPC system, the CPUs are in control and drive the computations on the DFEs. The data-intensive parts of the computations are typically confined to the DFE configuration and are made available through the SLiC interface. At its simplest level, DFE computation can be added to an application with a single function call, while for more fine-grained control SLiC provides an "action"-based interface. SLiC interface calls are automatically generated from the corresponding DFE program pieces and allow DFE programs to be used by a range of programming languages including C/C++ or R, Python, and MATLAB by way of a adaptation layer (or skin). Thanks to

MaxelerOS's seamless management of DFE resources, these programming languages may make use the DFE program as they would make use of any other external library or service.

3.2 Maxeler - Storm Interface

The integration of reconfigurable hardware (e.g. Maxeler machines) along with the QualiMaster infrastructure plays an essential role. A communication framework needed to be created so that the reconfigurable hardware could receive data, make any calculations needed, and then transmit the results back to rest of the infrastructure. At D5.2 the QualiMaster infrastructure and priority pipeline are describe. Figure 2 depicts the QualiMaster priority pipeline, where the Adaptation Layer may decide to execute 3a or 3b depending on the configuration and monitoring of the pipeline execution. Hence, 3a needs to implement the communication framework between the QualiMaster pipeline and the reconfigurable hardware. Basically it involves the implementation of a flexible interface between Storm and Maxeler, which could be also used with different tools or reconfigurable hardware platforms.

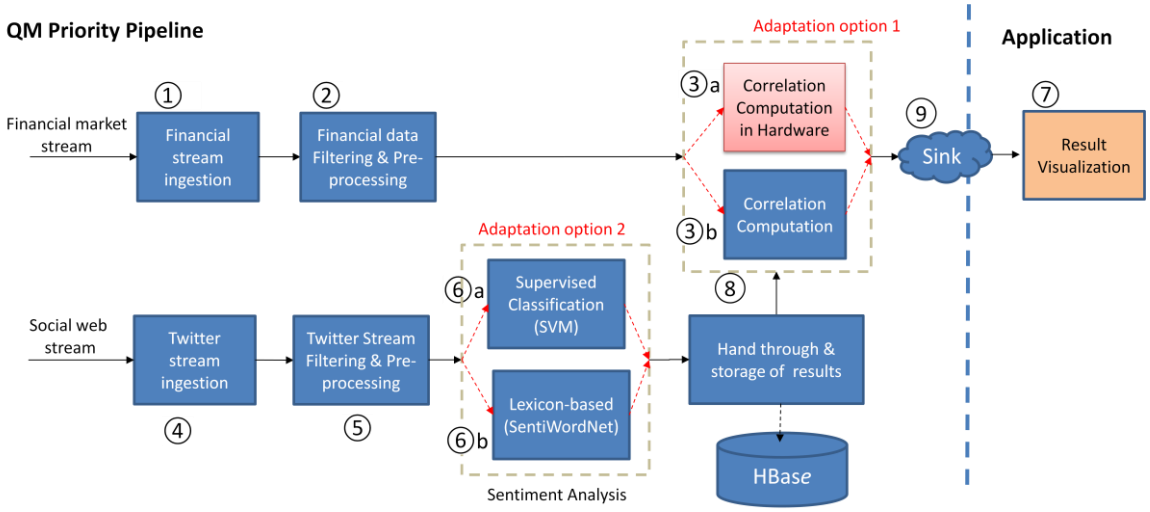


Figure 2: QualiMaster Infrastructure-Reconfigurable Hardware Integration (from D5.2)

In order to initiate Storm [31] the Adaptation Layer creates topologies. A Storm topology is a graph of computation. Each node in a topology contains processing logic, and links between nodes indicate how data should be passed around between nodes. The core abstraction in Storm is the "stream". A stream is an unbounded sequence of tuples. Storm provides the primitives for transforming a stream into a new stream in a distributed and reliable way.

The basic primitives Storm provides for doing stream transformations are "spouts" and "bolts". Spouts and bolts have interfaces that are implemented to run application-specific logic. A spout is a source of streams. For example, a spout may connect to the Twitter API and emit a stream of tweets. A bolt consumes any number of input streams, does some processing, and possibly emits new streams. Complex stream transformations, like computing a stream of trending topics from a stream of tweets, require multiple steps and thus multiple bolts. Bolts can do anything from run functions, filter tuples, do streaming aggregations, do streaming joins, talk to databases, and more.

Networks of spouts and bolts are packaged into a "topology" which is the top-level abstraction that is submitted to Storm clusters for execution. A topology is a graph of stream transformations where each node is a spout or bolt. Edges in the graph indicate which bolts are subscribing to which streams. When a spout or bolt emits a tuple to a stream, it sends the tuple to every bolt that subscribed to that stream.

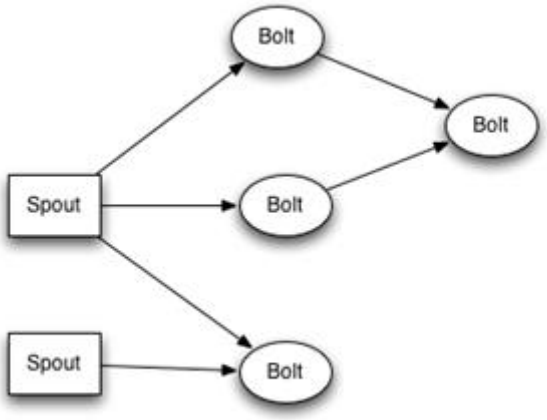


Figure 3: Storm Topology

Each node in a Storm topology, Figure 3, executes in parallel. In the topology, the amount of parallelisms specified, and then Storm will spawn that number of threads across the cluster to do the execution. A topology runs forever, or until the user kills it. Storm will automatically reassign any failed tasks. Additionally, Storm guarantees that there will be no data loss, even if machines go down and messages are dropped.

The Storm framework was installed on the Maxeler workstation and was tested as a simple Storm node. After that the connection with the Maxeler hardware had to be established. The Maxeler hardware is called by a C/C++ host code. The main problem that had to be addressed is basically the interface between Java (Storm) and C. Three methods that allow C/C++ and Java connection were considered, SWIG that allows the call of C function through Java, the exec function which calls the C executable and network sockets.

In order to use SWIG to connect the Storm Java code with C the Maxeler project was compiled as a shared library. SWIG (Simplified Wrapper and Interface Generator) [32]is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. SWIG is used with different types of target languages including common scripting languages such as Javascript, Perl, PHP, Python, Tcl and Ruby. The list of supported languages also includes non-scripting languages such as C#, Common Lisp (CLISP, Allegro CL, CFFI, UFFI), D, Go language, Java including Android, Lua, Modula-3, OCAML, Octave and R. SWIG is most commonly used to create high-level interpreted or compiled programming environments, user interfaces, and as a tool for testing and prototyping C/C++ software. SWIG is typically used to parse C/C++ interfaces and generate the 'glue code' required for the above target languages to call into the C/C++ code. After wrapping the C code with SWIG the C function can be called as a native code call from the Java. The problem was that in order to use SWIG, its' data types had to be used in order to be able to exchange arguments from C to Java.

The exec in Java is able to directly call the Maxeler executable. The exec function takes as arguments the executable file along with the arguments needed. Also a custom scheduler would have to be implemented in order to send the specific Java code (with the exec call) to the Maxeler workstation.

The first method proposed was the use of network sockets [33] in order to establish communication between the Java code and the C host code running on the Maxeler workstation. The Maxeler server creates sockets on start up that are in listening state. These sockets are

waiting for initiatives from client programs (e.g. Storm nodes). A TCP server may serve several clients concurrently, by creating a child process for each client and establishing a TCP connection between the child process and the client. Unique dedicated sockets are created for each connection. These are in established state, when a socket-to-socket virtual connection or virtual circuit (VC), also known as a TCP session, is established with the remote socket, providing a duplex byte stream. We can have up to 4 children on a Maxeler workstation as they can have 4 different hardware designs running simultaneously on the 4 FPGAs.

Two socket clients are used in order to make a call on the hardware server, a transmitter and a receiver. They are written in Java and run on a Storm subtopology (3a), which implements the appropriate algorithmic family interface. The transmitter creates a new socket in order to connect to the server and sends the configuration as well as the input data from previous components. The transmitter can also request for the results to be sent to the receiver. The receiver is connected via a different socket. It receives the results and forwards them to the rest of the pipeline. The socket server is written in C and serves as the Maxeler host code. It stays on listening state until a connection is established by both a transmitter client and a receiver client. The transmitter streams configuration and input data to the server. The server stores the data coming from the transmitter and performs the Maxeler hardware call to process them. After the hardware call has returned, it sends the results to the receiver client whenever a result request is received. The receiver and transmitter libraries can also be used outside Storm, which provides even more flexibility.

The three methods were compared in terms of flexibility and functionality. The use of SWIG basically reduces the flexibility of the interface, as the Java code would have to be rewritten using SWIG's data types. The exec function allows more flexibility as only a function call would have to be included in the Java code, but even though it worked perfectly when called by Java on the workstation, it didn't work if the code came through Storm. The network sockets approach was chosen, because they are flexible as they can be used by any tool that can implement sockets. Also sockets are faster as there is no need of interface functions (SWIG) or system calls (exec).The only drawback is that socket connections have to be opened on the client side (e.g. Java sockets), while exec or a native function call would be simpler.

4 Study of the algorithms

In this section the algorithms that have been selected are presented. The descriptions of the algorithms is more detailed in WP2 and we recapitulate important aspects for our work in this deliverable. This section has been added for reason of completeness of the deliverable.

4.1 Count-Min (CM)

The QualiMaster project focuses on the development of novel approaches that deals with large-scale data streaming. The Count-Min sketch is a popular and simple algorithm for summarizing data streams by providing decent summary statistics. The Count-Min data structure can be used in terms of the QualiMaster project for handling multiple and high-frequency large datasets in the proposed data processing settings with surprisingly strong accuracy. Count-Min [41] sketches are a widely applied sketching technique for data streams. A Count-Min sketch is composed of a set of d hash functions, $h_1(.)$, $h_2(.)$, ..., $h_d(.)$, and a 2-dimensional array of counters of width w and depth d. Hash function h_j corresponds to row j of the array, mapping stream items to the range of [1... w]. Let CM[i,j] denote the counter at position (i,j) in the array. To add an item x of value v_x in the Count-Min sketch, we increase the counters located at CM[h_j(x), j] by v_x, for j ∈ [1 ... d]. A point query for an item q is answered by hashing the item in each of the d rows and getting the minimum value of the corresponding cells. Note that hash collisions may cause estimation inaccuracies only overestimations. By setting d=⌈ln(1/δ)⌉and w =⌈e/ε⌉, where e is the base of the natural logarithm, the structure enables point queries to be answered with an error of less than e$||a||_1$, with a probability of at least 1-δ, where $||a||_1$ denotes the number of items seen in the stream. Similar results hold for range and inner product queries.

The goal of parallelizing the Count-Min algorithm and mapping it on a reconfigurable platform is the improvement of both the result quality and the processing times. In any skech-based sequential algorithm, the most expensive operations are the update and querying of the sketch data structure as it is updated for every item in the stream. To achieve scalability, our FPGA-based solution tries to accelerate these operations.

4.2 Exponential Histogram (EH)

The QualiMaster project focuses on processing of streams that come from different and distributed data sources. In addition, the goal of the QualiMaster is the efficient processing of huge amounts of data over time-based sliding windows. Exponential histograms (EHs) [17] guarantee complex query answering over distributed data streams in the sliding-window model. The use of EHs in the QualiMaster project would offer fast answering queries over distributed streams and efficient storage of the statistics over sliding windows. Exponential histograms [17] are a deterministic structure, proposed to address the basic counting problem, i.e., for counting the number of true bits in the last N stream arrivals. They belong to the family of methods that break the sliding window range into smaller windows, called buckets or basic windows, to enable efficient maintenance of the statistics. Each bucket contains the aggregate statistics, i.e., number of arrivals and bucket bounds, for the corresponding sub-range. Buckets that no longer overlap with the sliding window are expired and discarded from the structure. To compute an aggregate over the whole (or a part of) sliding window, the statistics from all buckets overlapping with the query range are aggregated. For example, for basic counting, aggregation is a summation of the number of true bits in the buckets. A possible estimation error can be introduced due to the oldest bucket inside the query range, which usually has only a partial overlap with the query. Therefore, the maximum possible estimation error is bounded by the size of the last bucket.

To reduce the space requirements, exponential histograms maintain buckets of exponentially increasing sizes. Bucket boundaries are chosen such that the ratio of the size of each bucket b with the sum of the sizes of all buckets more recent than b is upper bounded. In particular, the following invariant (1) is maintained for all buckets j:

$$C_j/(2(1+\sum_{i=1}^{j-1} C_i)) \leq e \ (1)$$

where e denotes the maximum acceptable relative error and Cj denotes the size of bucket j (number of true bits arrived in the bucket range), with bucket 1 being the most recent bucket. Queries are answered by summing the sizes of all buckets that fully overlap the query range, and half of the size of the oldest bucket, if it partially overlaps the query. The estimation error is solely contained in the oldest bucket, and is therefore bounded by this invariant, resulting to a maximum relative error of e.

The EHs access each data element at its arriving time and needs to be processed in real time. This constraint can be really challenging to be satisfied especially when there are irregularities and bursts data arrival rates. This problem is mainly due to insufficient time for the underlying CPU to process all stream elements or due to the memory bottleneck to process the queries. The Qualimaster project focused on the mapping of the EH data structure on reconfigurable hardware in order to develop new hardware-accelerated solutions that can offer improved processing power and memory bandwidth to keep up with the update rate.

4.3 Hayashi-Yoshida Correlation Estimator

One of the objectives for the QualiMaster project is the implementation of a platform that processes in real time financial data. The financial data can arrive in a non-synchronous way, thus the processing of such data streams is a really critical issue. The covariance among the prices of the market stocks plays a crucial role in modern finance. For instance, the covariance matrix and its inverse are the key statistics in portfolio optimization and risk management. There are two crucial points pertaining to practical implementation of computing correlation over streaming data. First, the actual transaction data is recorded at non-synchronous times. The covariance estimator calculation is, usually, based on regularly spaced synchronous data but this can lead to unreliable estimation due to the problematic choice of regular interval and the data interpolation scheme. Second, a significant portion of the original data sets could be missing at pre-specified grid points due to such randomness of spacing. Thus, the correlation would lead to unreliable results. One of the most efficient correlation estimators is the Hayashi-Yoshida Correlation Estimator. The proposed method does not require any prior synchronization of the transaction-based data. The Hayashi-Yoshida covariance estimator is defined as follows:

$$HYcov = \sum_{i,j} \left(P^1_{t_i} - P^1_{t_{i-1}}\right) * \left(P^2_{t_j} - P^2_{t_{j-1}}\right) * 1_{\{[t_i,t_{i-1}] \cap [t_j,t_{j-1}] \neq \emptyset\}} \quad (1)$$

$$where\ P^1_{t_i}, P^1_{t_{i-1}} the\ values\ of\ variable\ 1\ at\ times\ t_i, t_{i-1}, respectively, and$$
$$P^2_{t_j}, P^2_{t_{j-1}} the\ values\ of\ variable\ 2\ at\ times\ t_j, t_{j-1}, respectively$$

Eq. 1 uses the product of any pair of increments that will contribute to the sum only when the respective observation intervals are overlapping with each other. The Hayashi-Yoshida covariance estimator is consistent and unbiased as the observation time intensity increases to infinity. Using

the Hayashi-Yoshida covariance estimation Eq. 1, the proposed non-synchronous correlation estimator is computed by the Eq. 2.

$$HYcor = \frac{\sum_{i,j}\left(P^1_{t_i} - P^1_{t_{i-1}}\right) * \left(P^2_{t_j} - P^2_{t_{j-1}}\right) * 1_{\{[t_i,t_{i-1}] \cap [t_j,t_{j-1}] \neq \emptyset\}}}{\sqrt{\sum_i \left(P^1_{t_i} - P^1_{t_{i-1}}\right)^2 * \sum_j \left(P^2_{t_j} - P^2_{t_{j-1}}\right)^2}} \quad (2)$$

$$where\ P^1_{t_i}, P^1_{t_{i-1}} the\ values\ of\ variable\ 1\ at\ times\ t_i, t_{i-1}, respectively, and$$
$$P^2_{t_j}, P^2_{t_{j-1}} the\ values\ of\ variable\ 2\ at\ times\ t_j, t_{j-1}, respectively.$$

The quantities at the denominator represent the realized volatilities calculated using raw data. As shown in Eq. 2, the calculation of the Hayashi-Yoshida Correlation Estimator can be easily parallelized. Also, the reconfigurable hardware offers high parallelization level in order to calculate in parallel the correlation estimator among different stock markets.

4.4 SVM

The QualiMaster project exploits data derived from Twitter, in order to achieve risk analysis of financial data. Therefore sentiment analysis is a significant procedure of the project which is achieved with the SVM classification method as the specific method has yielded remarkable results in this area. In particular, Support vector machines (SVMs) were introduced by Vapnik et al. [6, 7] and they are considered to be highly accurate methods for a various set of classification tasks [5, 8, 9, 10]. Manning et al. [11] presented a work that uses SVM method for text classification. The algorithm takes as input a set of n training documents with the corresponding class labels and trains the SVM model. The linear SVM method aims at finding a hyperplane that separates the set of positive training documents from the set of negative documents with a maximum margin. The separating hyperplane, i.e. decision hyperplane[11] or decision surface [12], takes the "decision" for separating the input documents. However, Bernhard E. Boseretal. suggested a way to create nonlinear classifiers by applying the kernel trick (originally proposed by Aizermanetal. [13]) to maximum-margin hyperplanes [14]. The final algorithm is similar to the initial; apart from that every dot product is replaced by a nonlinear kernel function. This allows the algorithm to fit the maximum-margin hyperplane in a transformed feature space, and thereby achieving linear separation. The transformation may be nonlinear and the transformed space high dimensional. Thus, despite the fact the classifier can be a hyperplane in the high-dimensional feature space; it may be nonlinear in the original input space.

Furthermore, given training vectors $x_i \in R^n$, i= 1, …, l, in two classes, and an indicator vector $y_i \in R^l$, such that $y_i \in \{1,-1\}$ that represents the respective labels, SVM solves the following dual optimization problem:

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\boldsymbol{\alpha}^T Q \boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \le \alpha_i \le 1/l, \qquad i = 1, \ldots, l, \\ & \boldsymbol{e}^T \boldsymbol{\alpha} \ge \nu, \quad \boldsymbol{y}^T \boldsymbol{\alpha} = 0, \end{aligned}$$

In this problem, Q = $y_i y_j K(x_i, x_j)$, where $K(x_i, x_j)$ is the kernel and e = $[1, \ldots, 1]^T$ is a the vector of all ones.

Traditionally, SVMs have been used for binary classification scenarios, but they can be used for multiclass cases, as well. In our work, we built binary classifiers using the SVM methods from the LIBSVM package [15][47]. We integrated several LIBSVM functions to our hardware implementation and used the LIBSVM tool as a point of reference for SVM translation to hardware. However, there are many other open source SVM implementations, such as the SMO variant and the L2-loss linear methods implemented in the Weka library [16].

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.

4.5 LDA

QualiMaster uses social networks, news articles and other sources in order to gather data that can assist in financial risk analysis. A basic information retrieval method for documents is Latent Dirichlet Analysis (LDA). LDA is used in order to associate each document with a number of topics, generated at the training phase, with a certain probability for each topic. Techniques like Latent Dirichlet Analysis (LDA)[19] can be employed for uncovering the latent semantics of corpora, basically extracting the meaning of words in the corpora. Latent Dirichlet allocation identifies a given number of |Z| topics within a corpus. Being the most important parameter for LDA, this number determines the granularity of the resulting topics. In order to find the latent topics, LDA relies on probabilistic modeling. This process can be described as determining a mixture of topics z for each document d in the corpus, i.e., P(z|d), where each topic is described by terms w following another probability distribution, i.e., P(w|z).

By applying LDA latent topics as a list of terms with a probability for each term indicating the membership degree for the topic can be represented. Furthermore, for each document in the corpus LDA can determine through topic probabilities P(zj|di) regarding which topics the document belongs to and to which degree. The model needs to be trained on an example dataset and can be applied to any document later assigning the probabilities of topics to occur in that document. The training phase of LDA was considered to be implemented in hardware as it involves multiple iterations through the corpus in order to extract the topics of a dataset. The training phase's computations scale with the size of the dataset. FPGAs can be used in order to process the data in parallel, and thus reduce the execution time.

5 Algorithm Analysis for Hardware Implementation

The first step to map an algorithm to hardware is to analyze it accordingly. This section describes the analysis methodology and how it was applied to the selected algorithms. Using this analysis designer will create the first hardware model with the I/O study and proper data structures. This model is the intermediate level between algorithm and actual hardware design.

5.1 Analysis Methodology

An algorithm, that the designer want to translate in hardware, has to be analyzed for several characteristics from the designer point of view. Inputs and outputs of the algorithm, performance issues and the basic data structures and operations are the most important. 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 (e.g. too many forks/joints for little work done in hardware).Performance issues are very important in the scope of QualiMaster, as performance quality tradeoff is of main importance as it has been analyzed in D 2.1.

5.1.1 Study of Inputs and Outputs

Every algorithm has a set of inputs that are needed to be calculated to produce a set of outputs. Inputs and Outputs define the interface between the computational system and the user.

Inputs study consists of the data volume, their width, their nature, their source, and if there are used more than one time in the calculations. Volume could be the most important parameter for the study of the algorithm as it can lead to very important decisions, such as use of memory, number of processing units etc. If input data is very little then the system does not need any special handling for them as internal resources can handle them. If input data becomes bigger, and internal memory is not enough for handling their volume, then internal structures should be designed in order to compress or encode data, and, finally if volume of data is big, that reconfigurable device cannot store them, then an system e.g. external memory based system for buffering or swapping, should be designed to manipulate them. Data size is little or big depending on the target reconfigurable technology and device which is used from the designer.

The size and the characteristics of the input are important as for example if it is very large, and comes through conventional protocols (Ethernet, PCI etc) then special data structures should be designed to reassemble data. Such structures can be shift registers, very long word registers etc., which handle input data and reform them to the appropriate format for reconfigurable hardware processing.

The input data rate is also an important aspect to a system design. If the algorithm handles input data in burst mode data structures as FIFOs should be designed. If there is input data that an algorithm uses more than one time then the corresponding data structures, as cashing schemes, cyclic buffers, local memories, should also be used.

It is also important, for the designer, to study the if there is a need to design the controller for the protocol that the inputs may follow. If this system, for example, is directly connected to network with TCP/IP for streaming inputs, or an external DDR memory for large databases etc. The proper controller in hardware should be designed and integrated to the system.

The designer handles outputs similarly to the inputs. For QualiMaster scope Input and Output study is critical as it handles streaming data. Inputs and outputs have been often proved to be a

critical issue to designers for reconfigurable hardware, constraining the system performance as I/O was the bottleneck.

5.1.2 Data Sets

Application or algorithm mapping to hardware demands the appropriate data set on the design and testing procedure. Dataset should be representative accordingly to the algorithm and the way the final user will use this algorithm, in order to help designer to make the right decisions. An inappropriate data set can mislead the designer to decisions that will make the final system without the desired functionality or with low performance. Data sets are used at three design phases: at profiling, simulation and verification phase. These Data Sets are the same for the hardware and software designs but analyzed in different manner.

At profiling phase the designer must analyze the algorithm or the application in order to find the most computationally demanding part. If the data set is not proper, then the designer can focus accidentally on a different part of the code than he or she should, and as a result he/she will map to hardware an inappropriate part. The resulting system will not achieve high performance as the hardware part will not be accelerating the most demanding aspect of the algorithm.

At simulation phase, the data set has to be representative and to cover every state of the algorithm. If all states are not covered, then the system cannot be tested correctly and it will probably fail at run time.

At verification phase, proper data sets lead to proper functional verification. If the data set does not cover all cases then the system will not have been properly verified and at run time it may produce wrong results. Many times these results are very difficult to be found. The well-known 1994Intel Pentium FDIV division bug is the most the famous such case.

One solution could be to have a data set that will exhaustively test the algorithm. Such a solution is completely inapplicable in practically all cases, as due to state explosion the profiling, simulation and verification phases will take too long for testing. Such Datasets are the same used in software to show proper functionality.

5.1.3 Algorithm Profiling

Designers are trying to boost software performance by mapping applications to hardware. Inmost applications, usually 10% of the code consumes the 90% of the execution time, known as the 90/10 law in this context [38]. Following this rule of thumb, designers try to focus on the most demanding computational parts of an application. This approach helps the designer to save design area as he avoids to map large and complex parts of hardware that are lightly used. This area is used to map the computational demanding parts of the algorithm more than once to have a parallel execution, and a faster run time for the application. It should be noted, however, that optimizing 90% of the execution time may still be too little, as from Amdahl's law [39], even if an infinite speedup applies to the optimizable part and there is no communication overhead, a speedup of factor 10 will be achieved at most. This is a ceiling, and even the 10X speedup can be easily evaporated when communication overhead or limited performance improvement come to place. There exist applications which have substantial speedup on some critical section but in the total execution time (including I/O and communications overhead) there was a slowdown.

Several tools are used for the algorithm performance profiling. Tools as Intel VTune or GProf can profile an algorithm a procedure or instruction level. The designer runs the algorithm with a specific data set and the tool produces a report that shows the allocation of run time to every part of the

code. It also shows the number of times that a function has been called which may also very important depending on the designers target platform.

It is important for the designer to use representative data sets. If, for example, the dataset is really small as compared to the data sets that will be used on the final system then the system initialization for example, can demand a significant percentage of run time. If the system initialization is independent from input data size, then for a much larger data set the same initialization function will demand a much smaller percentage of the execution time. Also the data nature can affect the run time distribution to functions. Usually large data sets give the proper distribution for most application. A study of the algorithm options have to be done in order to use the data with the determined nature at profiling state.

5.1.4 Important Data Structures and Operations

The next step after profiling for the designer is to locate the important data structures and operations at the computationally intensive part of the algorithm. Data structures are important as using the proper ones the computations can be in parallel. As most appropriate Data Structures for mapping at reconfigurable hardware are considered static structures, with 1 and 2 dimensions tables as the most suitable. Dynamic structures using pointers, as trees, are considered as inappropriate structures for mapping in reconfigurable hardware If for example there is a comparison of an input against many comparators working in parallel will boost design performance, but even a DFS in a binary tree is challenging to map it in hardware efficiently.

Operations are also important in order to identify how system arithmetic will be implemented. If the application uses integers, and in which range, vs. single- or double-precision floating point numbers the required hardware resources may change substantially. Identifying the arithmetic, the designer will assign the available resources in order to achieve the maximum performance.

5.2 Count Min (CM) Modeling

The Count-Min algorithm is based on probabilistic techniques to serve various types of queries on streaming data. The Count-Min algorithm is able to handle massive data using data structures that occupy sub-linear space vs. the size of the input dataset. The CM sketch data structure can accurately summarize arbitrary data distributions with a compact, fixed memory footprint that is often small enough to fit within cache, ensuring fast processing of updates.

There have been several efforts to implement sketch data structures in hardware to accelerate their performance. The simplicity and the parallelism of the sketching algorithms makes such implementations convenient. Lai et al. [51] presented an implementation of sketching techniques using an FPGA-based platform, for the purpose of anomaly detection. Their implementation scales easily to network data stream rates of 4Gbps. Lai and Byrd [52] implemented a Count-Min sketch on a low-power stream processor, which processes a throughput rate up to 13 Gbps according to their results. In [53], Thomas et al. describe their implementation on a IBM cell processor with 8 processing units. Their results show an almost 8-fold speedup vs. the single-thread sequential code. Wellem et al. in [54, 55] proposed to use Graphics Processing Units (GPUs) for offloading heavy sketch computations for network traffic change detection. Their experiment results showed that GPU can conduct fast change detection with query operation up to 9 million distinct keys per second and one order of magnitude faster than sequential software version.

This section presents the analysis of the input and the output of the proposed sketch data structure. Also, we describe the datasets that were used for the analysis, the validation and the

testing of the implemented hardware-based system. Next, the algorithmic analysis of the CM method is presented. Last, we present the basic data structures and their operations that the proposed algorithm implements.

5.2.1 Study of Inputs and Outputs

The Count-Min sketch provides a different kind of solution to count tracking. It uses a fixed amount of memory to store count information, which does not vary over time. Nevertheless, it is able to provide useful estimated counts, as the accuracy scales with the total sum of all the counts stored. Streaming data from different applications, like IP networking, machine learning, distributed computing and signal processing, can be used as input to the proposed algorithm.

The Count Min sketch is a data structure that summarizes efficiently a stream of input data and answers queries with high accuracy over the input stream. Thus, the CM algorithm takes two different types of inputs: a stream of input data that is summed up on the CM sketch data structure or a query over the input dataset.

Input

The CM sketch takes as input raw streaming data that is used for building the sketch data structure. The input stream of data can be typically modeled as vector a[1 .. n]. The input vector consists of tuples with two values each: (the element id, the element's value), i.e. a = [(element_idx, value_x), (element_idy, value_y), (element_idx, value_z), etc]. The id is used for indexing the sketch data structure. The elements' values are used for changing accordingly the values of the sketch data structure.

In addition, the CM sketch is used for real time querying and answering over the input stream. In this case, the algorithm takes as input the just the querying element id.

Output

The algorithm does not output anything in case of updating the sketch data structure. On the other hand, in case of the query mode the algorithm outputs the information that is stored in the sketch data structure, which refers to the element id that is queried about.

5.2.2 Data Sets

The implementation of the Count-Min algorithm focused on the efficient mapping of the method on a hardware-based platform. We used a frequently used real-life data set, i.e. the worldcup'98 [37] (wc'98), for the algorithmic analysis, the performance evaluation and the validation of the output results. The wc'98 data set consists of all HTTP requests that were directed within a period of 92 days to the web-servers hosting the official world-cup 1998 website. It contains a total of 1.089 billion valid requests. Each request was indexed using the web-page URLas a key. We created point queries over the Count-Min sketch data structure estimating the popularity of each web-page by counting the frequency of its appearances.

5.2.3 Algorithm Profiling

This section describes the algorithmic analysis of the Count-Min algorithm. The Count-Min Algorithm consists of three main functions: the sketch initialization, the update and the estimation. The Count-Min sketch data structure consists of a fixed 2-D array of counters, of width w and depth d, as shown in Figure 4. Each row of counters is associated with a different hash function. The hash function maps items uniformly onto the range {1, 2, . . . w}. For items represented as

integers i, the hash functions can be of the form (a*i+b mod p mod w), where p is a prime number larger than the maximum i value (say, $p = 2^{31} - 1$ or $p = 2^{61} - 1$), and a, b are values chosen randomly in the range 1 to p − 1. It is important that each hash function is different; otherwise there is no benefit from the repetition.

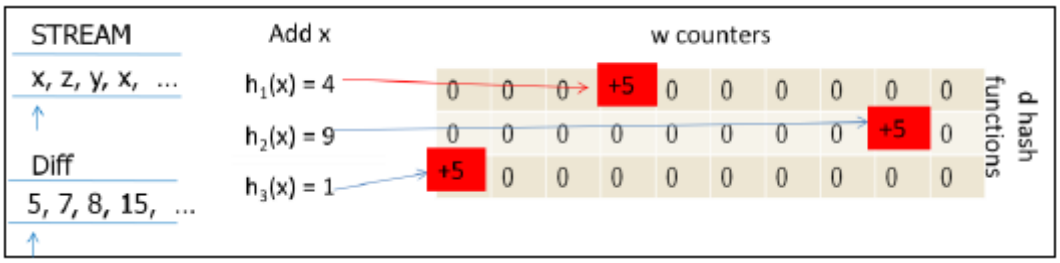


Figure 4: Count-Min sketch data structure

During the initialization stage the counters are initialized to zero, as shown in Figure 5. In addition, the parameters for hashing scheme are selected.

The update function takes as input a new tuple of data, i.e. (i, c), at each clock tick and updates the data structure in a straightforward way. The hash functions of all the rows are applied to the id of the incoming tuple. The result from the hash function is used to determine a corresponding counter. Next, the update function adds the c value to all the corresponding counters. Figure 4 shows an example of an update operation on a sketch with w = 10 and d = 5. The update of item i is mapped by the hash functions to a single entry in each row. It is important to mention that the incoming item is mapped to different locations in each of the rows. The pseudo code for the update function is presented in Figure 5.

Function 1: Count_Min_Initialize(w, d, num)

C[1, 1] . . . C[d, w] ← 0;
for j ← 1 to d do
Pick a_j , b_j uniformly from [1. . . num];

Function 2: Count_Min_Update(i, c)

for j ← 1 to d do
$h_j(i) = (a_j \times i + b_j \bmod p) \bmod w$;
$C[j, h_j(i)] \leftarrow C[j, h_j(i)] + c$;

Funtion 3: Count_Min_Estimate(i)

e ← ∞;
for j ← 1 to d do
$h_j(i) = (a_j \times i + b_j \bmod p) \bmod w$;
$e \leftarrow \min(e, C[j, h_j(i)])$;
return e

Figure 5: Basic functions for the Count-Min algorithm

The Count Min estimation function is used for queries over the streaming input data. The process is quite similar to the update function. It applies all the hash functions of the sketch data structure on the queried input element id. This process returns the values of the counters from the tables positions, where the querying id corresponds. Then the smallest value of the counter is returned as

the estimation result of the query, as shown in Figure 5. Figure 6 presents the flowchart of the Count-Min algorithm.

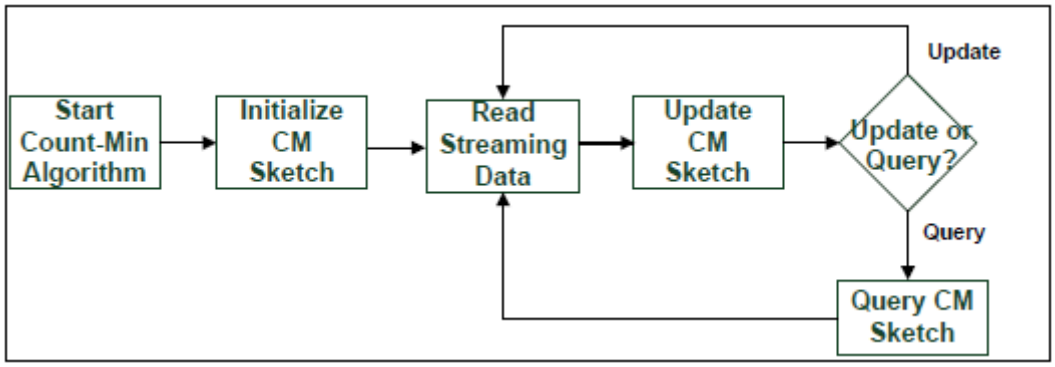


Figure 6: Count-Min Algorithm Flowchart

Next, we analyzed the algorithm as far as its possible parallelization level, which we can take advantage for a hardware-based implementation. The sketch update process has two different parallelization levels. First, each row of the sketch is updated independently of others, so the sketch can be partitioned row-wise for parallel processing. Second, another more coarse grained parallel formulation is the building of sketches on different subsets of data. The produced sketches can be combined in a straightforward way in order to give the sketch of the union of the data. This approach can be dramatically more efficient in terms of network communication.

5.2.4 Important Data Structures and Operations

This section describes all the important data structures and operations, which are implemented by the Count-Min algorithm, as described above, and they will be presented in our hardware-based proposed architecture.

As described in Section 4.1, the Count–Min algorithm implements a data structure that summarizes the information from streaming data and it can serve various types of queries on streaming data. This core data structure is a 2-dimensional array, i.e. count[w, d], which stores the synopses of the input streaming data. The size of the 2-dimensional array is defined by the w and d parameters, which are defined by two factors: ε and δ, where the error in answering the query is within a factor of ε with probability δ.

Thus, the factors ε and δ, which as described in Section 4.1 are selected with the formulas $d=\lceil \ln(1/\delta) \rceil$ and $w = \lceil e/\varepsilon \rceil$, can tune the size of the implemented 2-dimensional array based on the space that is available and the accuracy of the results that the data structure can offer.

The Count-Min algorithm uses hashing techniques to process the updates and report queries using sub linear space. Thus, the hash functions are pair wise independent to ensure lower number of collisions in the hash implementation. These hash functions can be precomputed and placed in local lookup tables, i.e. internal BRAMs of the FPGA device.

Another important issue of the Count-Min algorithm is the data input during the update process. The data streams are modeled as vectors, where each element consists of two values, i.e. the id and the incrementing value. When an new update transaction, i.e. (id, value), arrives, the algorithm hashes through each of the hash functions $h_1 \ldots h_d$ and increment the corresponding w entries.

At any time, the approximate value of an element can get be computed from the minimum value in each of the d cells of count table, where the element hashes to. This is typically called the Point

Query that returns an approximation of an input element. Similarly a Count-Min sketch can get the approximation query for ranges, which is typically a summation over multiple point queries.

5.3 Exponential Histogram (EH) Modeling

Another important data structure, which is an effective method for estimating statistics on sliding windows, is the exponential histograms (EHs). The EH is used for the counting problem, i.e. the problem of determining, with a bound on the relative error, the number of 1s in the last N units of time. The exponential histogram data structure is a histogram, where the buckets that record older data are exponentially wider than the buckets that record more recent data. When, a query takes place, i.e. to find the number of 1s that are seen in the last n units of time, we simply iterate over the buckets starting with the bucket containing the most recently recorded 1 till we find the bucket that covers the time we are interested in. Then we return the probabilistic distance of that bucket from the current timestamp.

Streaming processing and sliding-window domain is an important application domain and there are various hardware-based works that accelerate such workloads. Fowers et al. [56] analyzed the sliding-window applications domain when executing on FPGAs, GPUs, and multicores. For each device, they presented optimization strategies and analyzed the cases, where each device was most effective. The results showed that FPGAs can achieve speedup of up to 11x and 57x compared to GPUs and multicores, respectively, while also using orders of magnitude less energy. Qian et al. in [57] presented an novel algorithm named M3Join, which was implemented on an FPGA platform. The system needs only one scan over the data streams since different join queries share the intermediate results. The experimental results show that the hardware can accelerate join processing vastly.

This section presents the analysis of the input and the output of the proposed EH data structure. Also, we describe the data sets that were used the validation of our implemented system. Next, the algorithmic analysis of the EH method is presented. Lastly, we present the EH data structures and their basic operations.

5.3.1 Study of Inputs and Outputs

The EH model offers a probabilistic solution to the counting problem. The EH algorithm either updates the Exponential Histogram data structure with new streaming data or it estimates the number of 1s that have arrived from a specific timestamp up to the current timestamp.

Input

The EH algorithm takes as input a stream of data that can be typically modeled as vector a[1 .. n]. Each element of the input vector is a tuple of two values: the value and the corresponding timestamp, i.e. input stream = [(value_1, timestamp_0), (value_0, timestamp_1), (value_1,time_stamp_2), etc]. The first element of each tuple is either the value 1 or 0, while the second one is the clock timestamp which increments by one at each arrival. Only the timestamps of the elements, which have value 1, are stored in the Exponential Histogram. Also, the EH data structure takes as input the size of the window that we are going to process the input data.

Moreover, the EH data structure is used for real time querying and answering over the input stream. In that case, the algorithm takes as input a timestamp and estimates the number of the elements with value 1, which arrived from that timestamp up to the current timestamp.

Output

The algorithm does not output anything in case of updating the EH data structure. On the other hand, in case of processing a query, the algorithm outputs the estimation of the number of 1s that have arrived from a specific timestamp till to the current time.

5.3.2 Data Sets

As described above, the Exponential Histogram is a method that can efficiently offer a probabilistic solution to the counting problem. For the testing and the evaluation of our implemented system we used, again, the real-life data set from the Worldcup '98 [37] (wc'98). We streamed the data into the EH data structure. During the streaming process, we created and made queries over the EH data structure about the number of appearances of specific valid requests. The results that we took as answers were cross validated vs. the answers from Java implementation that we used as basis for our EH implementation.

5.3.3 Algorithm Profiling

This section makes the algorithmic analysis of the Exponential Histogram algorithm. The EH data structure is used for solving the Basic Counting problem. As described above, the Exponential histogram is a data structure that maintains the count of the elements with value 1 in the last N elements seen from the stream. The Exponential Histogram algorithm is based on two processes: 1) insert a new element and 2) estimate the number of elements with 1 value. Both of these algorithms were implemented on the hardware platform.

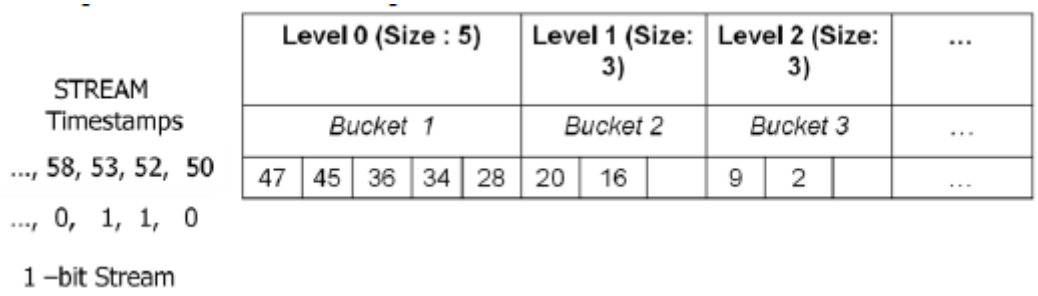


Figure 7: The main structure of the EH data structure

First, we analyzed the main function of the algorithm, which is the insert process. This function implements two basic steps of the algorithms: i) store the new element if its value is 1 in the first bucket and ii) update of the Exponential Histogram. The EH consists of a number of buckets that are placed in a row, as shown in Figure 7. The buckets keep the timestamps of the most recent elements and the total number of elements with 1-value, called bucket size. When the timestamp of the most recent element of a bucket expires (reaches window_size + 1), we are no longer interested in such data elements, thus we drop that bucket. If a bucket is still active, we are guaranteed that it contains at least a single 1 that has not expired. The first bucket maximum consists of C number of elements whereas the rests buckets consist of C/2 elements. When a new element with 1 value arrives, it is placed in the first bucket. The next step of the algorithm undertakes the update of the EH data structure. The process moves to the inner buckets and checks, if the buckets reach to its size. If this is true, then the algorithm merges the two last timestamps of the full bucket and passes the new element to the next bucket. The process continues till the last active bucket of the EH. The steps for the insert function are presented in Figure 8.

Function 1: Insert

1. When a new data element arrives, calculate the new expiry time. If the timestamp of the last bucket indicates expiry, delete that bucket, and update the counter Last containing the size of the last bucket and the counter Total containing the total size of the buckets.
2. If the new data element is 0, ignore it; otherwise, create a new bucket with size 1 and the current timestamp, and increment the counter Total.
3. Traverse the list of buckets in order of increasing sizes. If there are k/2 + 2 buckets of the same size, merge the oldest two of these buckets into a single bucket of double the size. (A merger of buckets of size 2^r may cause the number of buckets of size 2^r+1 to exceed k/2 + 1, leading to a cascade of such mergers.)
4. Update the counter Last if the last bucket is the result of a new merger.

Figure 8: Pseudo code for the Insert function

The second main function of the EH data structure is the estimation of the active elements with 1 value. The EH algorithm offers two choices: the estimation of the total number of 1's that exist in the EH data structure or the estimation of the 1's that exist up to a specific timestamp. For the first type of estimation, the EH data structure maintains two counters: one for the size of the last bucket (Last) and one for the sum of the sizes of all buckets (Total). The estimate itself is Total minus half of Last. For the second type of estimation, we first find the bucket that the timestamp belongs to. Second, for all but the last bucket, we add the number of the elements that are in them. For the last bucket, let C be the count of the number of 1's in that bucket. The actual number of active elements with value 1 in this bucket could be anywhere between 1 and C, and so we estimate it to be C/2. The pseudo code for the estimation function is presented in Figure 9. The flowchart of the complete EH algorithm is presented in Figure 10.

Function 1: Estimation

1. If the total number of 1's estimation is required move to step 2 otherwise move to step 3.
2. The total number of 1's estimation is calculated by adding the Total and Last counters, which the EH maintains.
3. Traverse the list of buckets in order of increasing sizes unitl you find a timestamp greater than the querying one. The total number of 1's is computed by adding the size of the traversed buckets + half the size of the last bucket.

Figure 9: Pseudo code for the Estimate function

It is important to mention that the arrival of each new element can be processed in O(1) amortized time and O(logN) worst-case time, due to the possible need for cascading merges. On the other hand, the estimation of the total number of elements with 1 value can be provided in O(1) time, as the EH maintains two counters: the Last and the Total counter, which can be updated in O(1) time for every data element. Lastly, the estimation number of elements for a specified timestamp can be processed in O(n) worst case time and in O(logN) best-case time due to the search bucket process.

It is clear from the algorithmic analysis that the problem of EH algorithm is not a parallelizable problem due to its sequential nature. On the other hand, the hardware can take advantage of the fine grained parallelization due to the cascading processes of the algorithm (either during the update process or the estimation process).

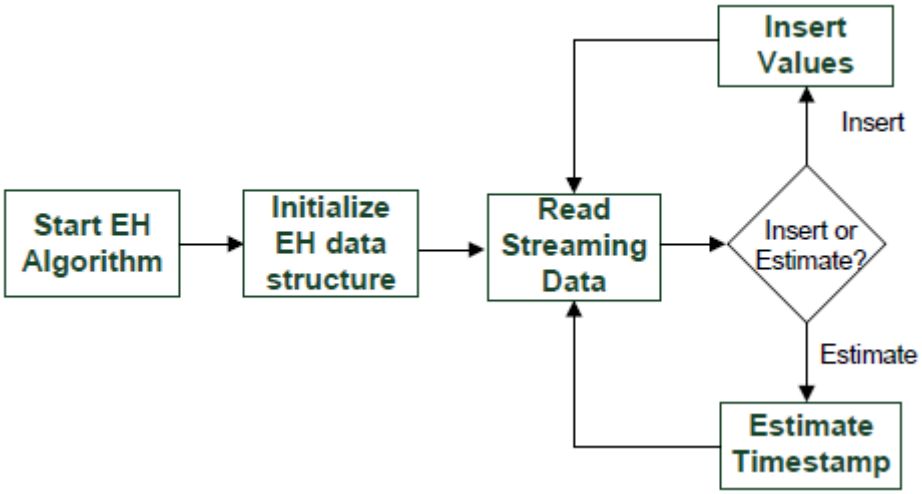


Figure 10: Exponential Histogram algorithm flowchart

5.3.4 Important Data Structures and Operations

This section describes all the important data structures and operations of the Exponential Histogram algorithm, which are implemented by our hardware-based architecture. As described above, the EH algorithm maintains the number of the elements with 1 values over a stream. The EH data structure is a list of buckets in a row, which are connected with each other. The number of buckets depends on the processing window size while the number of the size of each bucket is defined by the acceptable error rate ε, as described in 2.4.

During the insert function there are three different processes that need to take place. First, the EH data structure is examined, if it contains expired data, i.e. data that do not belong anymore to the processing window. Second, the new timestamp of the element with 1 value inserts to the first bucket. Third, all the buckets of the data structure are examined in order to merge buckets that reach to their maximum size.

Moreover, the EH data structure can estimate the number of the 1's that have appeared either in a complete window of time or from a specific timestamp up to the most recent current timestamp. The estimation of the 1s values over a window size is an easy procedure as the EH keeps at each time the total number of the 1s that have appeared and the number of the 1s at the last bucket level. Thus, the calculation of the total number of the 1's values takes places using these two counters. On the other hand, the calculation of the 1's values from a specific timestamp till recent timestamp needs the traversing of the EH data structure till to the specific timestamp by adding the estimation values from the previous buckets.

5.4 Correlation Modeling

The QualiMaster project's main goals are the use of methods that will improve the risk analysis on financial data and the implementation of systems that will monitor fine granular data streams for event detection. An important method that focuses on the correlation among the stocks' values is the correlation estimator.

There are many works that implement various correlation estimators on reconfigurable hardware. Ureña et al. in [58] described the design and development of a correlation detector a low-cost reconfigurable device. Fort et al. presented [59] an FPGA implementation of a synchronization system using both autocorrelation and cross-correlation. Their results showed that FPGA devices can efficiently map cross-correlation synchronizers. Lindoso et al. in [60] presented an FPGA-

based implementation of an image correlation algorithm, i.e. Zero-Mean Normalized Cross-Correlation. The experimental results demonstrated that FPGAs improved performance by at least two orders of magnitude with respect to software implementations on a high-end computer. Liu et al. in [61] presented a multi-channel real-time correlation system on a FPGA-based platform. Their system offered sliding correlation processing. Their proposed system achieved higher flexibility and accurate data-flow control when compared to previous traditional parallel processing architectures.

This section describes an initial hardware-based modelingof the well-known Hayashi-Yoshida Correlation Estimator [36]. This estimator can be applied directly on series of stock prices without any preprocessing.

In this section we analyze the input and the output of the described algorithm, followed by the algorithmic analysis of the Hayashi-Yoshida method. Lastly, we analyze and present the basic data structures and the operations that were used for mapping the Hayashi-Yoshida Correlation Estimator on a reconfigurable platform.

5.4.1 Study of Inputs and Outputs

As described above, the Hayashi-Yoshida (HY) Correlation Estimator measures the pair wise correlation of the input market stocks. The HY Correlation Estimator uses the transaction prices of two stocks in order to calculate their correlation. The correlation is calculated over time intervals that the stocks transactions take place. On the other hand, the QualiMaster project focuses on the correlation among a group of market stocks. The HY estimator calculates the correlation of all the different pairs of the processing market stocks. Figure 11 presents the equation for calculating the HY estimator for two different market stocks.

$$HYcor = \frac{\sum_{i,j}\left(P^1_{t_i} - P^1_{t_{i-1}}\right) * \left(P^2_{t_j} - P^2_{t_{j-1}}\right) * 1_{\{[t_i,t_{i-1}] \cap [t_j,t_{j-1}] \neq \emptyset\}}}{\sqrt{\sum_i \left(P^1_{t_i} - P^1_{t_{i-1}}\right)^2 * \sum_j \left(P^2_{t_j} - P^2_{t_{j-1}}\right)^2}}$$

$$where\ P^1_{t_i} = value\ of\ stock\ 1\ at\ time\ t_i\ and\ P^2_{t_j} = value\ of\ stock\ 2\ at\ time\ t_j$$

Figure 11: Hayashi-Yoshida Correlation estimator

Input:

The proposed system takes as input the names and the number, N, of the stocks that are going to be processed. Next, it takes as input a stream of the transaction prices of the N stocks. The actual transaction data are recorded at random times, i.e. different timestamps, which means that we have to calculate the correlation estimator of all the pairs of the stocks during the different time intervals. The input stream can be modeled as vector a[1 .. n], where each element consists of tuples that describe the id of the stock, the new value of the stock and the transaction time, e.g. a = [(stock_1, value_0, timestamp_0), (stock_10, value_0, timestamp_0), (stock_3, value_1, timestamp_1)].

Output:

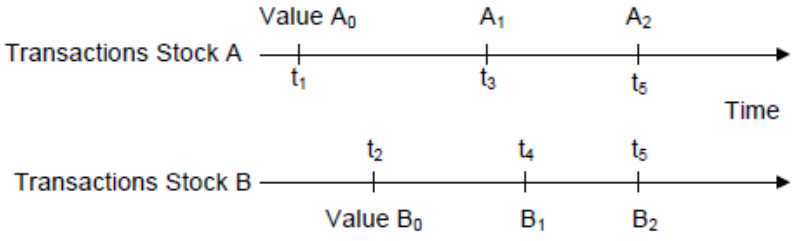
The output of the algorithm is an upper triagonal matrix with each cell showing the computed correlation between two market stocks.

5.4.2 Data Sets

The QualiMaster project will use high volume financial data streams. We tested and evaluated the correlation software-based system by using real data from the stock market. The stock prices are provided by an API from the SPRING that provides access to real time quotes and market depth data to the consortium.

5.4.3 Algorithm Profiling

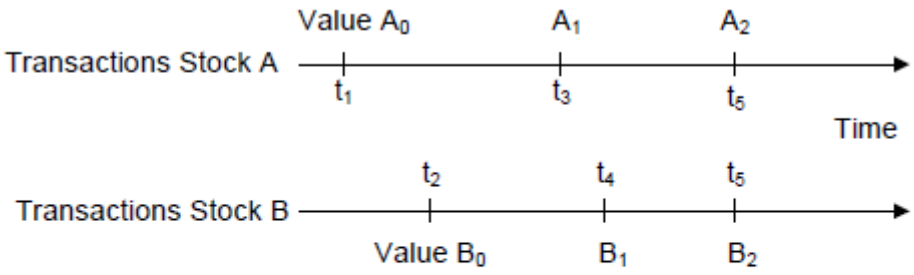
This section presents an algorithmic analysis of the HY correlation estimator. The HY estimator calculates the correlation between two stock markets using their transaction prices. As the stock transactions are non-synchronous, the HY correlation estimator is calculated over all the overlapping transaction time intervals. Figure 12 shows an example for the calculation of the HY estimator over the transactions of two market stocks The HY estimator calculation is based on the computation of the covariance for all the pairs of the market stocks and the calculation of the transactions of each market stock separately. The direct use of equation of Figure 11 has O(mn) complexity(where m and n are the number of transactions for the first and the second stock respectively).



$$HYcor = \frac{(A_1-A_0)(B_1-B_0)*1+(A_1-A_0)(B_2-B_1)*0+(A_2-A_1)(B_1-B_0)*1+(A_2-A_1)(B_2-B_1)*1}{\sqrt{(A_1-A_0)^2+(A_2-A_1)^2}*\sqrt{(B_1-B_0)^2+(B_2-B_1)^2}}$$

Figure 12: Example of calculation HY correlation estimator

The high complexity of the algorithm and its streaming nature led us to propose a new method for calculating the HY estimator. Our proposed method omits the non-overlapping time intervals from the calculation of the HY estimator by finding the overlapping time intervals during the streaming data arrival. The overlapping time intervals of all the pairs of the market stocks are found on the "fly". Thus, each time a new transaction arrives, we keep only its value and its timestamp. Then, the coefficients from the past overlapping intervals are added to the coefficients of the HY estimator, which are kept in a table. This procedure is repeated at each timestamp, thus we know exactly the overlapping intervals of the market stock values and their values at the start and the end of the overlapping time interval. The correlation estimator can be calculated for each pair of the stocks at any timestamp using the coefficients that were computed above. If a stock did not change value during the previous time interval, the HY estimator does not change due to the term of ΔP(I) = Pi-Pi-1. Our proposed method, which can be applied to software, too, has O(m+n) time complexity. Figure 13 shows the calculation of the HY correlation using our new proposed algorithm for the same example as the one presented in Figure 12. The flowchart of the HY estimator method is presented in Figure 14.



$$HYcor = \frac{(A_1-A_0)(B_1-B_0)+(A_2-A_1)(B_2-B_0)}{\sqrt{(A_1-A_0)^2+(A_2-A_1)^2}*\sqrt{(B_1-B_0)^2+(B_2-B_1)^2}}$$

Figure 13: Example of Calculation HY with our proposed method

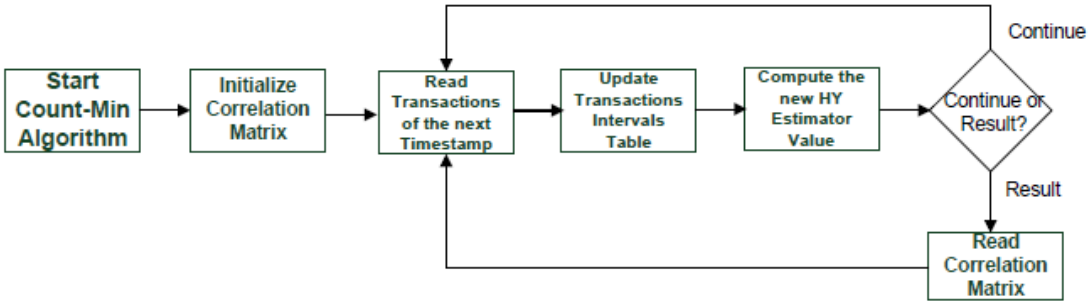


Figure 14: HY estimator algorithm flowchart

It is clear from the algorithmic analysis that the computation of the HY correlation estimator can be easily parallelized. In our future plans, we aim to calculate the HY coefficients of different pairs of the stock markets in parallel.

5.4.4 Important Data Structures and Operations

This section describes the basic data structures that were used for the computation of the HY estimator. As described above, we implemented a variation of the official algorithm that calculates the Hayashi-Yoshida estimator. Our proposed solution offers lower time complexity and takes advantage of the algorithm's streaming nature.

First, we implemented a data structure that keeps the stocks' transaction values at the beginning and at the end of the overlapping time intervals for each pair of the input stocks. This data structure is a 2-dimensionalarray that stores at each timestamp the new transactions of the stocks (if they exist). Next, these values are used for the computation of the HY covariance, as presented from equation of Figure 11.

Next, we used another 2-dimensional array that keeps just the transactions of the stocks. This table is, also, updated by the transaction values that arrive at each timestamp. The values of this array are used for the calculation of the denominator values of the HY estimator.

5.5 SVM Modeling

Related work has shown that SVM is a problem suitable for hardware. In particular, in [48] Cadambi, Srihari, et al. achieve 20x speedup with the use of a Virtex 5 FPGA, compared to a 2.2 GHz CPU processor. Furthermore, Papadonikolakis and Bouganis in [49] utilize an Altera Stratix III FPGA to reach 7x speedup compared to other hardware-based implementations. Additionally, in

[49] Pina-Ramfrez et al. use a Virtex II to implement the SVM method, but do not achieve speed up compared to a 550 MHz CPU processor. The aforementioned results indicate that SVM can be accelerated with the use of hardware for the purposes of the QualiMaster project. In order to perform SVM Modeling we used Version 3.20 of the C/C++ open source LIBSVM project. Modeling a system entails a functional (commonly referred to as behavioral) prototype with such considerations as data operation precision, functional units, sequence of operations, etc. In this context the model is a reference design done in software (typically with MATLAB or C/C++, but it can also be in Java or Python) which gives the designer a feeling for the cost vs. performance tradeoffs. To illustrate, Papadonikolakis and Bouganis' work on SVM [49] uses fixed point precision rather than floating point precision because fixed point takes fewer resources vs. floating point. The evaluation, however, of the quality of the results was performed prior to the development of an architecture and a detailed hardware design – if the quality was poor there would be no need to proceed with the time-consuming design. Similarly, if one needs a hardware system with the same accuracy as the software (and we assume that this is a realistic scenario) the system operation has to be modeled from the beginning, in the case of our example (and the QualiMaster work on the same algorithm) with floating point operations. Based on the bibliography, the majority of works on SVM compare their performance to the results of LIBSVM, both in terms of accuracy and speed([42], [43], [44], [45]). During the modeling process first we analyzed the data inputs and outputs of this implementation. Then we used specific data sets to perform profiling of the software code, and finally we identified important data structures and operations. A very useful additional result of modeling is not only that it allows for comparisons against pure software implementations of an algorithm, but it also provides detailed datasets and expected results for the actual hardware design.

5.5.1 Study of Inputs and Outputs

The SVM Training algorithm receives as input a two dimensional structure and outputs the SVM model.

Inputs

The input of the LIBSVM library is a file containing training data, with a specific format. Although the SVM Classification method will be performed on social streaming data, the Training phase which produces the appropriate classifier will be applied on historical data. 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 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. More specifically, the two following sequences represent two rows (data instances) of the file.

Data instance #1: -1.0 6:1 11:-0.73 12:0.17 13:0.0 14:0.25 15:0.01

Data instance #2: 1.0 6:1 5:-0.36 12:0.25 14:0.25 15:0.17 16:0.26

Values -1.0 and 1.0 indicate the labels of the data instances. Moreover, in all *a:b* expressions, *a* denotes the number of the feature and *b* represents the value of the respective feature. Note that it is not necessary that all data instances in a file share the same features. For example the first data instance has a value for feature 11, whereas the second provides no information about this feature. In this case we consider that feature 11 of data instance 2 has value 0. In order to utilize the information of the input file, the software code copies this information into a data structure.

Output

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. In addition it contains the same data instances that were included in the input data file, only now they are grouped based on their category. In particular, for binary classification all data instances with label 1 are together and the same applies for all instances with label -1.

5.5.2 Data Sets

So far we have used artificial data sets similar with the real that will be used in the QualiMaster project. Therefore we utilized the ijcnn1 data set which was created for the needs of the IJCNN 2001 neural network competition. The respective training data set contains 30000 data instances and the maximum number of features a data instance can have is 22.

We decided to use the ijcnn1 data set for two reasons. First because it required little execution time to produce an output, thereby allowing performing several tests in short time. Other data sets with bigger dimensions required several hours to produce a result and this is impractical during the implementation phase of an algorithm. The second reason is that our first implementation could not support data sets with a lot of features. We designed our first implementation based on a simple idea that would lead to the creation of a first hardware implementation of the SVM Training algorithm, without optimizations.

However, we are currently implementing a design that can support a data set of arbitrary dimensions. Once it is finished, we will be able to data sets of different sizes in order to observe the execution time required by the hardware implementation as the size of the dataset increases or decreases. One of these data sets will be provided by WP2, since we need to configure our hardware implementation based on the data sets they need to classify.

5.5.3 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 code using the Linux GNU GCC profiling tool (gprof) so as to detect potential parallelism.

Description	Time percentage	SVM Function
Computes $x_{i*}x_j$	70.23%	*dot_product()*
Computes the kernel function $K(x_i,x_j)$	8.27%	*kernel_computation()*
Finds sub problem to be minimized in each iteration	8.27%	*select_working_set()*
Computes $y_{i*}y_{j*}$ $K(x_i,x_j)$	6.18%	*get_Q()*
Solves the optimization problem	3.49%	*solve()*

Table 1: SVM profiling analysis

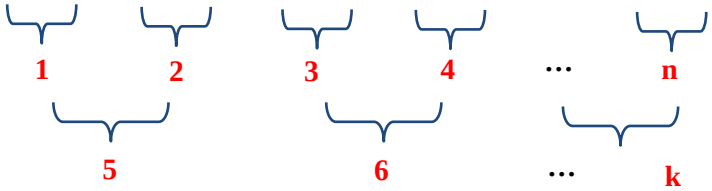
Quadratic programming optimization problems, such as the SVM classification algorithm are expensive. In cases where the data sets are high-dimensional and large, the kernel and inner product computations require a massive number of matrix-vector operations. This can be observed

in the above table since all functions that appear perform matrix-vector operations. A brief description of each function is given in the Table 1. However, a more detailed description is also presented below.

1) dot_product() : 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 input file. In software, computing the dot product of two vectors of size 22 implies a 22x22 double loop i.e. 484 iterations. However computing 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 22 dot products produced simultaneously. Once we have produced the products we can add them in pairs. In particular, dot products 1 and 2 could be added together, dot products 3 and 4 could be added together and so on. Note that in our 22 length vectors for example 11 addition pairs are formed, so that 11 additions are produced. Again, computing one addition is independent of computing another addition and this allows parallelization. Therefore, the 11 first additions can be performed in parallel. Then, the outcomes that are produced can also be added in pairs and simultaneously and this procedure continues until we are left with a single number, which is the dot product result.

For example, given A = [A_1, A_2, … , A_n] and B = [B_1, B_2, … ,B_n] the dot product is defined as:

$$\boldsymbol{A \cdot B = \sum_{i=1}^{n} A_i B_i = A_1 B_1 + A_2 B_2 + A_3 B_3 + A_4 B_4 + \cdots + A_n B_n}$$



More specifically, a parallel version of the dot product computes in parallel all products from **1** to **n.** Once these are produced, calculating the sum takes place. In particular, 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 implementation every loop of the optimization solver requires the computation of the dot product between a selected data instance with the rest of the data instances of the input file. It is worth mentioning that producing in parallel the maximum number of possible dot products is much more efficient than only computing a single dot product. However, we need to take into account the maximum possible number of vectors that can be drawn from memory in parallel. Also, we should guarantee that there is a sufficient amount of resources to perform all these computations in parallel.

2) kernel_computation(): The dual formulation of the SVM optimization problem introduces the notion of the kernel. In the case of linearly separable data, the kernel of two vectors is equivalent to the dot product of these two vectors. However, if we are dealing with non-separable data the kernel can be defined by a variety of functions. For instance, there is the polynomial kernel, the Gaussian kernel, the Laplacian kernel and others. 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. Hence, we selected a kernel function that was suitable for our training data set. In particular, we chose the exponential kernel function which according to the equivalent software implementation of this function is computed based on the following formula:

$$e^{-(gamma*dot(x_i,x_i)+dot(x_j,x_j)-2*dot(x_i,x_j))}$$

In the above formula gamma denotes a constant selected by the user and dot produces the dot product of the given vectors. We have already described how the dot product can be performed with parallel computations. What is indicated by this formula is that all dot products can be computed in parallel too. For instance, if the dot product function receives 22-length vectors and all products are performed in parallel, then in total 22*3 products are produced in an instance. However, the result of dot(x_i,x_i) is a constant number as for every loop of the optimization solver *i* remains the same, and thereby we do not need to compute this dot product every time. It is the indicator j that changes during a loop - this indicator which receives all values from 0 up to the number of data instances of the input file. At this point we illustrate the specific loop, in order to provide better understanding.

```
for(j=0;j<number_of_instances;j++)
{
    result[j]=exp(-gamma*(x_i+dot(x[j],x[j])-2*dot(x[i],x[j],));
}
```

, where *x_i* equals dot(x_i,x_i) and remains constant during the whole loop.

Nevertheless, although the expression within the brackets can be implemented effectively in hardware, the same does not apply for the exponent. In particular, the computation of the exponent is expensive for hardware due to the fact that it occupies a significant number of resources and requires a noticeable amount of time to be executed. Thus, the entire formula will be calculated on the software side and only the expression within brackets will be implemented on hardware.

3) select_working_set(): The Q matrix of the dual optimization problem 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 [46].

4) get_Q(): This function produces a vector with length size equal to the number of data instances. Note that within a loop of the optimization solver we compute the formula presented in the description of the kernel computation. During this loop 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 get_Q function we also add the labels of the respective vectors so as to compute the following formula:

$$y_i y_j K(x_i, x_j)$$

In the above formula K(x_i,x_j) is equal to the expression in the description of the kernel computation and y_i, y_j are the labels of the respective vectors. Note that *y* may either equal to 1 or -1, when we are performing binomial classification, or it may receive other values, when we are performing multi-class classification. We studied binomial classification, in accordance with WP2. Furthermore, due to the existence of the exponent in K(x_i,x_j) the expression presented in get Q cannot be entirely implemented on hardware.

5) solve(): 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 is the arbitrary sized loop of the dual formulation optimization problem. Due to the fact that in each loop, certain variables of the algorithm are updated and these updated values affect further computations, avoid dependencies in the optimization loop, and thus leave its execution to software.

5.5.4 Important Data Structures and Operations

Data Structures

As mentioned in the beginning of the SVM Modeling section, the LIBSVM software receives as input a file that follows a specific format. The information contained in this file is copied to a data structure so as to be able to utilize the given information. Prior to describing this important structure we need to provide further information about the input file. More specifically, although the number of features a data instance can have equals the maximum number of features of the dataset, this does not imply that all data instances will feature equal to the maximum number of features. For example, in a dataset where the maximum number of instances is equal to 6, data instance 1 could have features 1, 2 and 3 and data instance 2 could have features 4 and 5. Not having a specific feature implies that the value of this feature is 0. Thus data instance 1 has value 0 in columns 4 and 5, since these columns correspond to features 4 and 5. Similarly data instance 2 has value 0 in columns 1, 2 and 3.

The data structure that contains the above information is essential to the algorithm, as all the important functions of the implementation need it to produce outcomes. In order to understand its elements we describe the following two structures.

```
structsvm_node
{
        int index;
        double value;
};
structsvm_problem
{
        int l;
        double *y;
        structsvm_node **x;
};
```

Data structure *svm_node* represents the expression *a:b* that was described in 3.2.1. In particular, it contains an integer number that denotes a feature and a double number that corresponds to the respective value of the same feature. Note that in order to preserve information about the input file we need to store file_rows*(file_columns-1) *svm_node* components.

The above information is included in the *svm_problem* structure. More specifically, *structsvm_node* ***x* represents precisely a two dimensional structure that contains elements of type *svm_node*. Thus, the rows and columns of the file correspond to the respective dimensions of the structure. We know that a row ends and a next row follows when the index variable has value -1. In addition, variable *l* denotes the number of rows in the file and list *y* contains the labels of the data instances in *structsvm_node* ***x*. Therefore, the length of the list of labels will equal *l*. For example, if *l* equals 5, the length of the list of label also equals file, and the first dimension of the two dimensional data

structure is also 5. However, the second dimension of the structure varies based on the number of features a data instance has.

According to the above, we can conclude that structure *svm_problem* is the most important structure in the SVM Training implementation since it contains all essential information and is used by the most important operations.

Operations

The most important operations were revealed during the profiling phase of the software source code. These are hidden in the expression $y_i y_j K(x_i, x_j)$

and are the following:

- $x_i . x_j$ and $x_j . x_j$
- $gamma * (x_i . x_i) + (x_j . x_j) - 2 * (x_i . x_j)$
- $K(x_i, x_j) = e^{-(gamma * dot(x_i x_i) + dot(x_j x_j) - 2 * dot(x_i x_j))}$
- $y_i y_j K(x_i, x_j)$

A detailed description about all the aforementioned functions, as well as their importance for hardware is provided in Section 5.5.3.

5.6 LDA Modeling

Latent Dirichlet allocation (LDA) is a generative probabilistic model of a corpus. The LDA algorithm uses inference to generate this probabilistic model. There are multiple algorithms that can implement Inference such as Variational Inference. Gibbs Sampling etc. M. Shah et al. [20] implemented a speech emotion recognition framework based on Variational Inference using FPGAs, which presented good results both in terms of accuracy (80%) and performance.

In the sections below three different implementations of Inference are presented. The last method named sparseLDA will be analyzed further as it is very efficient in terms of performance.

5.6.1 Study of Inputs and Outputs

LDA takes as input a set of documents and posits that each document is a mixture of a small number of topics and that each word's creation is attributable to one of the document's topics.

The applications that implement LDA take as input a filtered set of documents, from stop words, common used words etc., or filters a set of documents. They then create an index table with the vocabulary and use the indexes in processing. The training step of the algorithm produces a number of topics (input by user). The basic idea is that documents are represented as random mixtures over latent topics, where each topic is characterized by a distribution over words. The model produced by the training is then used to characterize new documents. Each new document is characterized by a different set of topics.

5.6.2 Data Sets

For testing the LDA implementation we collected and prepared data sets consisting of text files where each line is a bag of terms from a document within the particular dataset. Following data sets are available:

- *Flickr emotions dataset* (47MB): This dataset consist of a Flickr image metadata crawl where emotional tags like “angry”, “happy” were used as queries. Each document in this dataset is a concatenation of a particular image title, tags and description.
- *Global warming dataset* (64MB): This dataset is a cropped dataset to a Wikipedia article about global warming. Cropping is a technique where given a set of documents (the Wikipedia article in our case) as a seed, a set of similar documents can be selected. Thereby key phrases are extracted from the initial set and used as text queries (to Wikipedia again in our case) to obtain more similar documents and expand the initial set. Thus the dataset consist of Wikipedia articles related to the topic “global warming”. Each document in this dataset is a paragraph from one of the articles.
- *Jesus dataset* (75MB): This dataset was constructed in exact the same way as the predecessor but with the Wikipedia article about “Jesus” as a seed. -
- *Newsgroups dataset* (9.3MB): Newsgroups: The 20-Newsgroups dataset was originally collected by K. Lang . It consists of 19,997 newsgroup postings and is usually divided into 7 categories for supervised learning covering different areas: “alt” - atheism, “comp”- computer hardware and software, “misc”- things for sale, “rec” - baseball, hockey, cars, and bikes, “sci” - cryptography, medicine, space, and electronics, “soc” - Christianity, and “talk” - religion, politics, guns, and the middle east. The number of postings for these categories varies from 997 to 5,000.
- *CS Proceedings dataset* (30MB): We collected scientific publications within the Computer Science domain. We gathered 2,957 scientific documents from proceedings of the conferences in the following areas: “Databases” (VLDB, EDBT), Data mining (KDD,ICDM), “E-Learning” (ECTEL, ICWL), “IR” (SIGIR,ECIR), “Multimedia” (ACM MM, ICMR), “CH Interfaces” (CHI, IUI), and “Web Science” (WWW, HYPERTEXT). The number of publications for these categories varies from 179 to 905. We limited our selection to conferences that took place in the years 2011 and 2012 and publications with more than four pages and we removed references and acknowledgment sections.
- *WWW proceeding dataset* (26MB): We collected conference proceedings from the ACM WWW conferences in the time period between 2001 and 2012, capturing the evolution of expert knowledge in the context of the World Wide Web. We grouped the resulting 1,687 documents by pairs of consecutive years and obtained the following six categories: “2001-2002”, “2003-2004”,: : :, and “2011-2012”. The number of publications for these temporal categories vary from 139 to 637.
- *Movie dataset* (7.2 MB): From the Wikipedia movies-by-genre lists we extracted all available movies belonging to one or more of 10 main genres such as “Action”, “Comedy” and “Drama”, amounting to 11,070 movie descriptions. The number of movies in the different categories ranges from 656 to 2,364. From each of the movie pages we only considered the content of the sections “Plot” or “Description” and omitted the surrounding meta data information.

5.6.3 Algorithm Profiling

Variational inference

The first implementation of LDA considered was the Princeton implementation by David M. Blei et.al. [18][19]. This application was chosen as it was written in C and would be compatible with the hardware tools (the Maxeler host code has to be written in C/C++). The profiling showed that the most time consuming function was the digamma function (78% of the processing time). This function is called multiple times by the Variational inference algorithm. This implementation wasn't studied further as it is not the one used by the community. This study has been done as Variational Inference is the basic algorithm and the first LDA implementation.

Gibbs sampling

The next implementation considered was based on Gibbs Sampling. It is the algorithm implemented in R project [21] and Mallet [22] tools. Mallet is the most commonly used tool for topic modeling and has many different Inference algorithm implementations for the LDA. The basic LDA implementation is based on Gibbs Sampling.

The profiling of the Java implementation showed that more than 80% of the execution time is used by the Gibbs Sampling stage. The algorithm iterates through the documents (the number of iterations is given by the user) and for each word, iterates through all the topics (the number of topics is given by the user) and updates the appropriate values. This algorithm looked promising as there are various parallel implementations of Gibbs sampling on GPUs and multicore platforms [23, 24 , 25]. The most time consuming function is the sample TopicsForOneDoc(); that Implements the Gibbs sampling. It is called for each document for the number of iterations. The function iterates through the word of each document and updates the topic distributions.**Sparse LDA.**

Both the previous implementations are deprecated as the SparseLDA presented in [26] is mostly used by the community, as it is at least 20x faster than the standard Gibbs Sampling implementation. In their work they evaluate the performance of several methods for topic inference in previously unseen documents, including methods based on Gibbs sampling, variational inference, and a new method inspired by text classification. The classification based inference method produces results similar to iterative inference methods, but requires only a single matrix multiplication. Results indicated that SparseLDA can be approximately 20 times faster than traditional LDA and provide twice the speedup of previously published fast sampling methods, while also using substantially less memory.

As with the previous implementations, the most time consuming function is the sampleTopicsForOneDoc() function. This function is called for each document on the training dataset, number of iterations (given by user) times. This algorithm is faster than the previous Gibbs sampling methods as it doesn't need to iterate on all topics for each word. Also it is important to mention that with the increase of the number of topics SparseLDA can be more than 20 times faster than the simple Gibbs sampling implementation.

5.6.4 Important Data Structures and Operations

Variational Inference and standard Gibbs sampling are mentioned as these are the basic algorithms for LDA. Sparse LDA achieves performance improvement up to 20x over these algorithms, using a new algorithmic approach. For that reason it is not worthy to analyze further the Variational Inference and standard Gibbs sampling, and so we focus only on SparseLDA.

SparseLDA

SparseLDA is a lot faster than other implementations due to the data structures used in order for the algorithm to not iterate over the number of topics. In [26] by rearranging terms they changed

$$P(z = t|w) \quad \propto \quad (\alpha_t + n_{t|d})\frac{\beta + n_{w|t}}{\beta V + n_{\cdot|t}}$$

into

$$P(z = t|w) \propto \frac{\alpha_t \beta}{\beta V + n_{\cdot|t}} + \frac{n_{t|d}\beta}{\beta V + n_{\cdot|t}} + \frac{(\alpha_t + n_{t|d})n_{w|t}}{\beta V + n_{\cdot|t}}$$

and by splitting those three parts they get

$$s = \sum_t \frac{\alpha_t \beta}{\beta V + n_{\cdot|t}}$$

$$r = \sum_t \frac{n_{t|d}\beta}{\beta V + n_{\cdot|t}}$$

$$q = \sum_t \frac{(\alpha_t + n_{t|d})n_{w|t}}{\beta V + n_{\cdot|t}}$$

Also the expression for q can be broken into two components

$$q = \sum_t \left[\frac{\alpha_t + n_{t|d}}{\beta V + n_{\cdot|t}} \times n_{w|t}\right]$$

The first coefficient can therefore be cached for every topic, so calculating *q* for a given *w* consists of one multiply operation for every topic such that $n_{w|t} \neq 0$.Then they present the data structures that allow the rapid identification topics such that $n_{w|t} \neq 0$ and also that are able to iterate over non-zero topics in descending order. For this reason they encoded the tuple (t, $n_{w|t}$) in a single 32 bit integer by dividing the bits into a count segment and a topic segment. The number of bits in the topic segment is the smallest m such that $2^m \geq T$. They encode the values by shifting $n_{w|t}$ left by m bits and adding *t*. They recover $n_{w|t}$ by shifting the encoded integer right *m* bits and *t* by a bitwise and with a “topic mask” consisting of *m* 1s. This encoding has two primary advantages over a simple implementation that stores $n_{w|t}$ in an array indexed by *t* for all topics. First, in natural languages most word types occur rarely. As the encoding no longer relies on the array index, if a word type *w* only occurs three times in the corpus, an array of at most three integers is needed for it rather than *T*. Second, since the count is in the high bits, the array can be sorted using standard sorting implementations, which do not need to know anything about the encoding.

By storing encoded values of (t, $n_{w|t}$) in reverse-sorted arrays, it can rapidly calculate *q*, sample *U*, and then (if $U > (s+r)$) usually within one calculation find the sampled *t*. Maintaining the data structure involves reencoding values for updated topics and ensuring that the encoded values remain sorted. As $n_{w|t}$ changes by at most one after every Gibbs update, a simple bubble sort is sufficient.

6 Mapping algorithms in Hardware

This chapter presents the algorithms mapping to hardware procedure. Firstly, an overview of the design procedure is given. Next, the following sections analyze the designs that have eventually been mapped to hardware and their corresponding initial performance results vs. the official software solutions.

6.1 Design Methodology

This section describes the methodology that it is followed to map efficiently an algorithm to specific hardware technology, as shown in Figure 15. This method is generic in the sense that the steps are technology independent as method, and mapping technology affects aspects of the modeling and not the method itself. The sequence of steps to get from a specification to a working hardware prototype is called a design flow, and each step of the design flow is associated with specific Computer Aided Design (CAD) tools, such as simulators, synthesis tools and place and route tools.

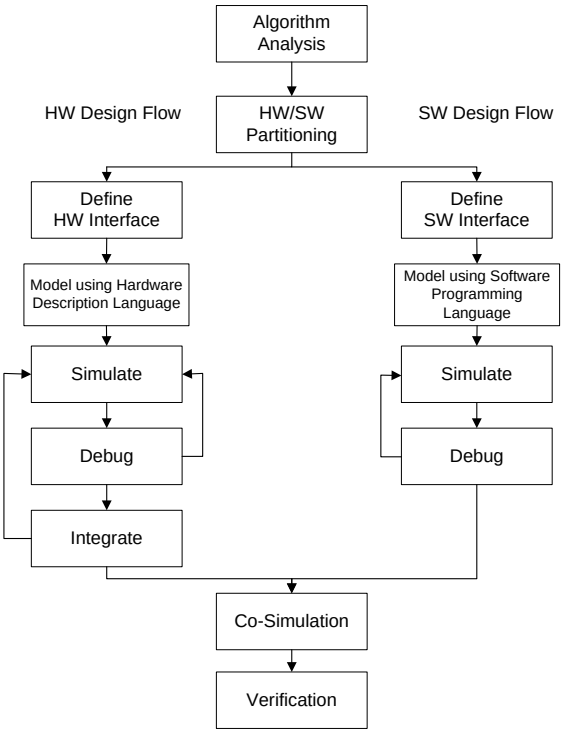


Figure 15: Design flow for mapping an algorithm on hardware

6.1.1 Top Down analysis

Depending on the algorithm analysis the first step is to map the algorithm in abstract block diagrams identifying the interface between them. For each block there is a description of the functionality and its interface with other blocks and external systems. Functionality can be described in an abstract manner but it has to be checked that each task is assigned to be executed to a unique block. The total functionality of the block tasks is the task of the complete system.

This procedure is at system level, and it combines software and hardware. The first major system division is on software and hardware components(parts). Depending on the platform that the designer wants to map his system interfaces between software and hardware are usually predefined. Usually at the mapping of the algorithm in hardware what remains in software does not change from its official version and design continues for the hardware part.

At the hardware component level, the functionality of each block can be described in a formal manner using a programming language as C/C++, Matlab, or even using formal verification methods depending on the system complexity and size. It can also be described in a less formal manner but it has to be very precise.

As a rule of thumb the number of subsystems is up to four but it depends on system complexity and size which is crucial for such a decision. This procedure is repeated for every subsystem, and following the same rule of thumb up to three times for a moderate system.

Subsystems functionality and interfaces which include physical connections and the protocol of communication, is crucial to be well defined at this procedure.

At the end of that phase a quite detailed block diagram of the system and its subsystems is available on the designer.

Depending on the platform and technology several design decisions have to be taken at this phase for each block. For example the memory system is usually very important for a computing system. Using FPGAs, designer has to think of the internal very high bandwidth memory (BRAM) and the way of communicating with the external memory. With such an approach block functionality is closer to what can be actually be implemented in a system.

6.1.2 Bottom Up Modeling

Using the block diagrams of the Top Down analysis, for the hardware part, the designer starts to model each block. Modeling is done with a Hardware Description Language which can be low level as VHDL or Verilog or a modern high level language as Maxeler Java Extension[Maxeler], Vivado C/C++ [Vivado], SystemC [SystemC] etc. The designer can also use modules from module libraries, such as protocol implementations, DDR controllers, video controllers, several filters or even a general purpose processor. The designer can also use other design tools as Xilinx Coregen[Coregen],MatLab Simulink [Simulink], which produce modules for specific technology. With such tools the designers usually produce memory controllers, floating point arithmetic units or even modules implementing more complex arithmetic operations.

In this procedure an equivalent functional module is built for each block. The module is tested for the equivalent functionality vs. the initial model. The results of the tested modules are validated vs. the results that are produced from the corresponding software solution. After the testing phase the integration procedure commences. Usually two tested modules are connected as a subsystem and the functionality of the subsystem is proved to be equivalent to the reference system. Then, a new tested module is added and with this procedure is repeated adding a new block. In that manner designer follows the reverse procedure of the Top Down analysis, building the complete system using subsystems as in the block diagram.

Modular modeling and integration are really useful to the design procedure as several designers can work in parallel, following the block diagram and the interface descriptions. In that manner the design procedure is significantly faster vs. a serial implementation of the hardware components. The independent working designers procedure proves how crucial is to have a proper and well defined Top Down analysis, as any functional overlap between the blocks, or any ambiguous description of interfaces can lead to block diagram revision and consequently to new block modeling for several blocks.

6.1.3 Debugging Approaches

Debugging stages follows the Bottom Up Modeling stages, and the integration phase. At every single module that have been build a test bench applies in order to certified that the its functionality is equivalent to the desired as it was described at Top Down analysis and the bottom up modeling.

Several debugging tools are used, the most common of which are simulators as ISE simulator or ModelSim. More sophisticated tools are used for complex platforms as Vivado Simulator, Maxeler Simulator, Convey Software-Hardware Simulator. These tools usually provide to the designer a visualized representation of the system functionality.

Modules debugging is a demanding procedure. It is very important to have the proper testing vectors for each module. Debugging and test vectors creation is better to be created from designers that have not be involved in modeling phase. This independent procedure helps to detect more problems as wrong or missing functionality. In order to have the proper test vectors, and for complex subsystems, the designer makes small applications to produce them.

Simulations should be exhaustive for small modules and extensive for the most complex ones. Exhaustive simulation examines and proves the complete functionality of the system but it is very time demanding procedure. For this reason, for more complex systems an extensive simulation is usually performed. The system is examined at several different states for several thousand or millions clock cycles. A careful selection of states and test vectors should be done in order to test the complete module functionality. This procedure is repeated for every module that has been build. At integration phase it is repeated for every module added to the subsystem.

The simulation phase starts with the functional model which specifies only the functionality of the model without any timing data, such as set up / hold time for example (these are timing constraints for proper clocking of sequential circuits). When the proper functionality has been checked for each module, a new simulation has to be done where a timing model for the target technology is included to test the module or the subsystem with timing parameters. The test vectors are the same, without any changes from stage to stage.

After this procedure the designers have a confidence on their model that it will work in real world. This functional model, also, defines a very abstract resource utilization and the clock speed achieved by the full system design. There are two factors for a system to work in simulation and not in real world. The first case (most typical) is not the extensive testing of a module, which can lead to functional failures. The second factor is that functional models simulate the real world but they are missing some important parameters, e.g. delays at inputs or external noise, which can lead to timing problems and thus to the complete wrong functionality.

6.1.4 Verification Issues

When the complete model has been build, the simulation procedure has to be used for verification. As an algorithm or a software have been modeled the hardware designer has to prove that it produces the same results, or for approximate methods results with a standard error.

The designed system runs using as inputs several typical data sets and the outputs are compared against an official distribution of software or a reference software-based implementation running the same inputs. In order to compare outputs(as output can be a very large file), the designers can implement software to compare automatically the hardware data results vs. the reference system or the desired behavior as it is determined by simulators.

6.2 Count Min (CM) Design

This section presents our initial proposed hardware-based architecture for the Count-Min algorithm. Our proposed solution was mapped on a high-end Maxeler MPC-series platform using a single FPGA device.

6.2.1 Top Down analysis

First, we had to define the parameters as far as the size of the sketch data structure. The typical values for both ε and δ are in the space [0.05, 0.2]. In our first implementation, we used the typical values $\varepsilon = 0.085$ and $\delta = 0.05$, which lead to a sketch that has d = 3 different hash functions and w = 32 elements in each row. Our proposed system is fully parameterizable, which means that we can change dynamically the dimensions of the sketch data structures according to the needs of our application.

Figure 16 presents our proposed top level architecture. In our initial approach, we mapped the update function on the reconfigurable hardware, while the query process is resolved in software. The system starts with initializing the Count-Min tables on hardware platform. Next the system accepts a stream of (id, value) tuples. These tuples are batched and then we pass them into the reconfigurable part where the sketch update takes place. When a query arrives, the system uses the sketch that is built in reconfigurable platform and answers the query.

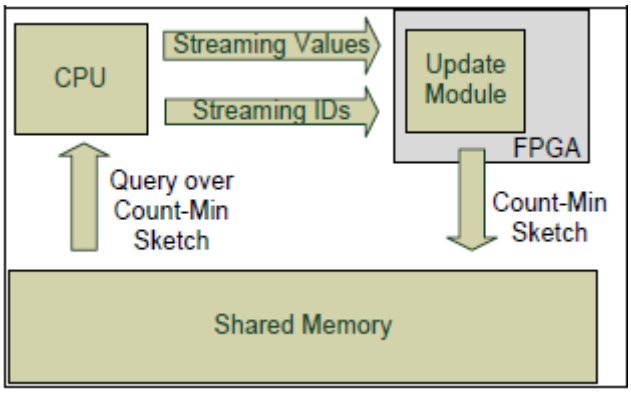


Figure 16: Count-Min Top level Architecture

6.2.2 Bottom Up Modeling

This section describes the individual components that are presented in Figure 16.

The CPU device executes the initialization steps of the Count-Min data structure. Also, it is responsible for sending the streaming data to the reconfigurable part of the system. Last, the CPU can read the Count-Min data structure from the shared memory and it resolves any query over it.

Figure 16 shows the mapping of the update function on reconfigurable hardware. The implemented module takes as input the streaming ids and their corresponding values. The hash functions are implemented as lookup tables in reconfigurable hardware, where the precomputed values have been loaded before the start of the processing. The lookup tables take as input the streaming IDs and output the corresponding values from the hash functions. These values are used as index to the memories, as presented in Figure 17. Each memory module corresponds to a single row of the sketch data structure. The values are updated and stored again in Block Rams (BRAMs). When the processing finishes, the values of the memories return to the shared memory, which can be accessed by the CPU, too. The query processing takes place from the CPU. When a new query

arrives, the CPU reads the CM sketch data structure from the shared memory and returns the query estimation.

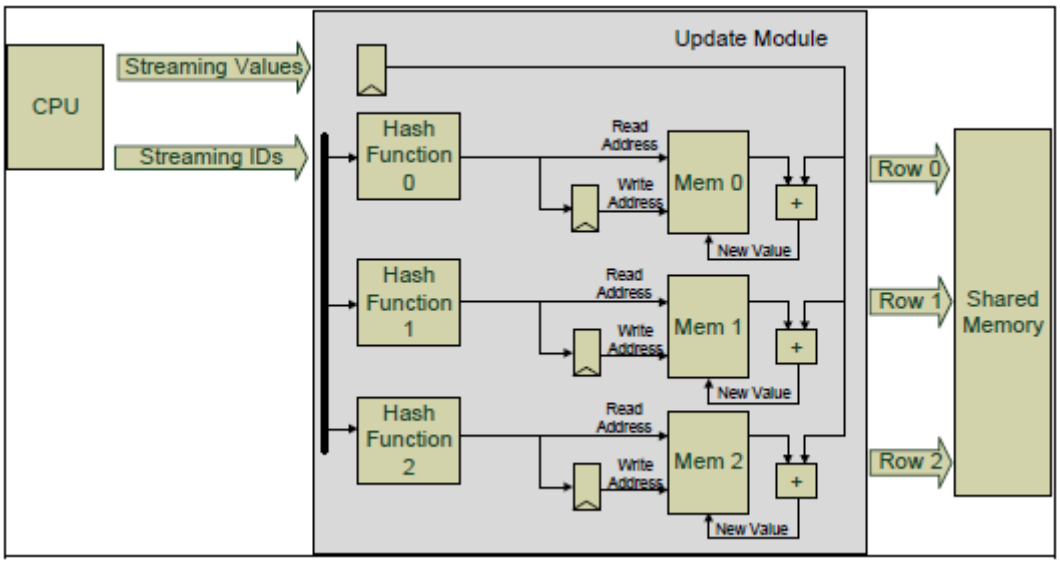


Figure 17: Update function Architecture

6.2.3 Debugging Issues

The software code, which initializes the Count-Min data structure and resolves the queries, was ported in the MaxIDE platform. The reconfigurable part was implemented using the the Max Java language. The system was simulated, using the Maxeler simulator, keeping both the software and the simulated hardware running.

6.2.4 Verification Issues

The MassDAL Public Code Bank[40] offers a C version code that implements both the update and the query processing of a Count-Min data structure. The results were verified vs. the results from the officially distributed code of the Count-Min algorithm from the MassDAL Public Code Bank.

6.2.5 Performance Evaluation

Our proposed system was mapped on a single DFE without taking into account the parallelization that the multi-DFEs can offer us. The testing dataset was random items that reached up to 10^{10}. This initial performance comparison showed that our system outperforms the single threaded official software solution for about 7 times. This our first implementation mapped only the update function of the Count-Min data structure while the estimation function takes place in software. In our future plans, we aim to propose an architecture that will implement both the update and the estimation function in reconfigurable platform. Also, we will take advantage of the coarse grained parallelism that the algorithm can offer, where we will map Count-Min data structures in independent FPGA devices, which will be combined in order to conclude to the final Count-Min data structure.

6.3 Exponential Histogram (EH) Design

This section presents our proposed hardware-based architecture of the Exponential Histogram algorithm. Our proposed solution was mapped on a Maxeler server using a single FPGA device.

6.3.1 Top Down analysis

First, the top level analysis of the system that implements the Exponential Histogram algorithm on hardware is presented. As referred in Section 5.3, we both implemented the functions of updating the EH data structure and estimating the number of the elements with value 1 on the reconfigurable hardware. The system starts with initializing the EH data structure and its corresponding counters. Next, the streaming values with their corresponding timestamps are passed to the reconfigurable device for updating the EH structure. The estimation of a specific timestamp or batches of timestamps are passed from another streaming port to the reconfigurable module and the corresponding results return back to the CPU. It is important to mention that for this first approach only one of the two streaming processes (streaming a new value or estimating a new timestamp) at each clock cycle can take place. Figure 18 presents the top level system architecture.

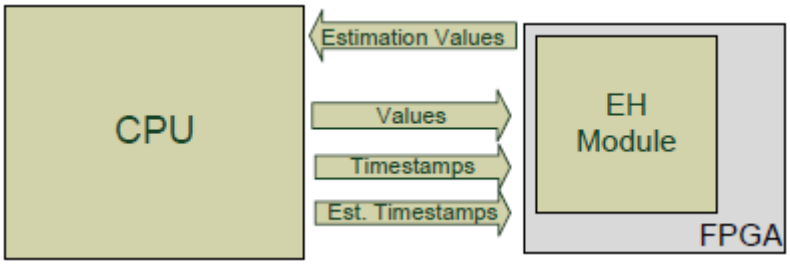


Figure 18: Exponential Histogram top level architecture

As defined in Section 5.3, the EH data structure consists of non-overlapping buckets. The buckets have increasing sizes. The first bucket has size k+1 while the rest buckets have size k/2 + 1. The variable k is bounded by the 1/ε value, where the typical values for the error value ε are between [0.05, 0.2]. We used ε= 0.1 in our implementation, thus the bucket of the first level consists of 11 elements and the remaining ones consist of 6 elements. The window size defines the number of buckets of the EH data structure. Our proposed architecture is fully parameterizable, which means that we can change dynamically the dimensions of the EH according to the needs of the application.

6.3.2 Bottom Up Modeling

First, the CPU resolves the building of the interconnection between the CPU and the reconfigurable part. Second, the CPU sends a signal that initializes the EH structure and the corresponding counters. Next, the reconfigurable module takes either a stream of elements with values 1s or 0s with their corresponding timestamps or a stream of timestamps for estimation. The update process is separated into two stages: the first stage omits the expired data from the processing bucket while during the second one a new value from the input or the previous bucket is put in the bucket. In case of a new input, the timestamp of the new value is passed into the first level bucket. As shown in Figure 19, the buckets are 1-D arrays in range of [6, 20], as analyzed in Section 5.3.3, which work like a complex shift-register. In other words, when a new timestamp-value arrives at the input of a bucket all the previous values are shifted to the right for one position. After the insertion completes, there is specific logic which checks for merging condition for the last two elements of the bucket. If a new merged value needs to be passed to the next level, it is stored in the pipeline registers and the process continues the second level during the second clock cycle. The important issue here is that our implementation is fully pipelined which means that each level can serve the insertion/merge of a different timestamp. In other words, our proposed system exploits the fine grained parallelization that the hardware can offer by processing in parallel N different input values (like the number of total levels).

Moreover, our proposed system implements the estimation processing either for the total window or for a specific timestamp. As shown in Figure 19, the EH module takes as input the timestamp that we want to estimate the number of elements with value 1. In case, that we want to calculate the 1's estimation value of the complete processing window, we pass the timestamp value -1. During the estimation processing, the value passes to the first level, where the estimation module calculates the estimation of this level. At the next clock cycle, the estimated value of the present level with the estimation timestamp passes to the next level bucket. The processing finishes when the score reaches to the last level and it returns back to the CPU. It is clear that our proposed architecture is fully pipelined taking advantage of the hardware fine grained parallelization.

6.3.3 Debugging Issues

The initial software code that initializes the EH sketch data structure was ported in the MaxIDE platform. The reconfigurable part was implemented using RTL coding, i.e. VHDL. The hardware-based code was mapped on a single DFE device of the Maxeler platform.

6.3.4 Verification Issues

We tested our system with various input datasets and different configurations, i.e. number and size of buckets. We queried our system for the estimation of different estimation timestamps and in different times and the results were verified vs. the results of the official Java code of the EH data structure[35].

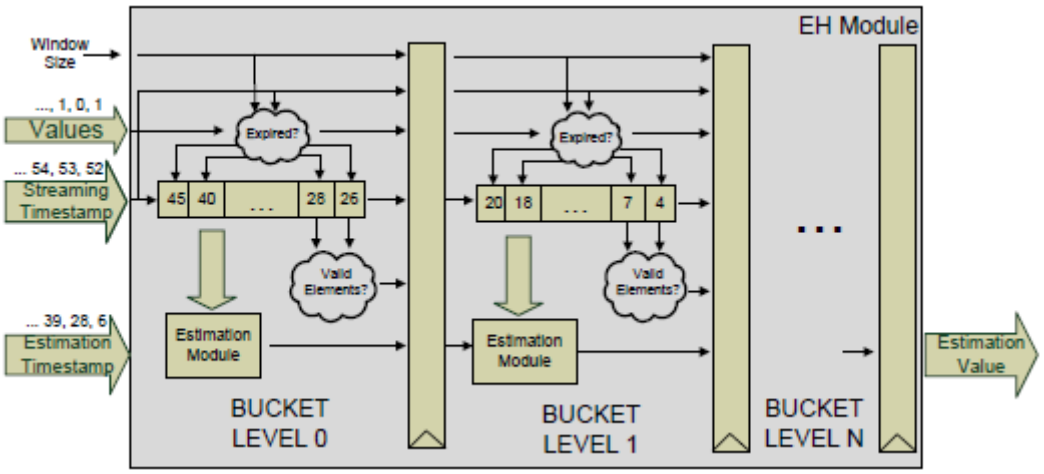


Figure 19: Architecture of Exponential Histogram module

6.3.5 Performance Evaluation

We compared the performance of the hardware-based solution for the EH algorithm vs. the official software solution. The initial results showed that our proposed system can offer 4 times faster processing of the EH data structure vs. the official code of the EH data structure[35]. It is important to mention that the presented architecture is not fully optimized as we do not take advantage of the parallel nature of the algorithm. Lastly, we aim that our next system architecture will take advantage of the coarse grained parallelization implementing EH data structures on independent streams taking advantage of the low resource utilization (only 4% of the resources of the FPGA device are used) that our proposed architecture offers.

6.4 Correlation Design

This section presents a first hardware-based architecture for the HY estimation algorithm. Our proposed solution was mapped on a Maxeler server using a single FPGA device.

6.4.1 Top Down analysis

Our proposed architecture is a software-hardware co-design system. The software part builds the data in proper data structures while the hardware part implements all the computations. The system starts with the initialization of the internal data structures and the correlation matrix that is stored in the shared memory of the Maxeler platform. Next, the streaming of input data begins. The software reads all the transactions for a single timestamp and updates the arrays that were described in Section 5.4.4. In more details, the software updates the transaction values of the overlapping time intervals for all the pairs of the input stocks(only if there was transaction for one of the two stocks of the pair). Also, it updates the table with the transactions of the single stocks. Then, these tables are streamed to the reconfigurable part of the system, which computes the new coefficients for the HY correlation estimator. Last, the software is responsible for calculating the correlation matrix using the intermediate results, i.e. coefficients that have been computed by the hardware and stored in the shared memory. Figure 20 presents the top level system architecture.

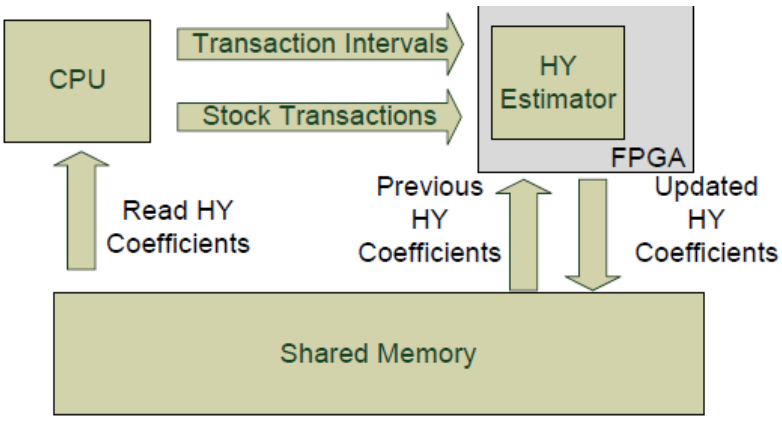


Figure 20: Top level system architecture

6.4.2 Bottom Up Modeling

As referred above, the proposed architecture is a software-hardware co-design system. The presented architecture is an initial attempt to map the HY correlation algorithm on Maxeler platform. The CPU initializes the internal data structures and the correlation matrix. Also, it reads all the transactions for a single timestamp and updates the corresponding data structures that are transmitted to the reconfigurable part for further processing. The reconfigurable part of the system, i.e. HY module, calculates the HY coefficients for the calculation of the HY correlation estimator. The HY Module is mapped on a single FPGA device and it calculates the HY estimation value for each one of the market stocks pairs. As shown in Figure 21, it takes as input the transaction values of a pair of stocks and computes the HY covariance value at each clock cycle. Also, it takes as input the transaction prices for each one of the input stocks and calculates the denominator of the HY estimator. The final values are stored in shared memory. Lastly, the CPU reads these values from the shared memory and computes the final correlation matrix.

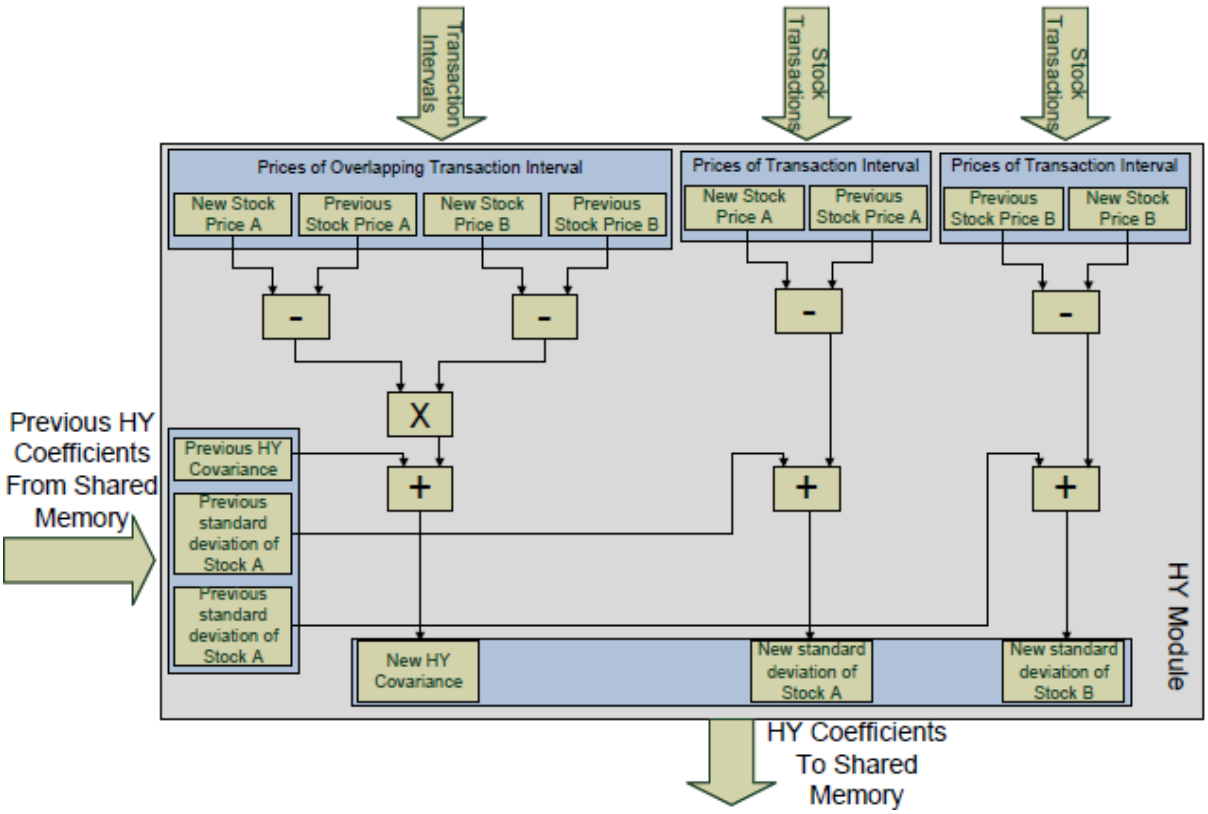


Figure 21: Architecture of Hayashi-Yoshida Estimator module

6.4.3 Debugging Issues

First, the software code, which initializes the internal data structures, calculates the correlation matrix and receives the stock market transactions, was ported in the MaxIDE platform. The reconfigurable part was implemented using the Max Java language. The hardware-based module was mapped on a single DFE device of the Maxeler platform.

6.4.4 Verification Issues

We tested our system with various input datasets, i.e. number of market stocks. The results of the hardware-based system were verified vs. a reference software-based implementation.

6.4.5 Performance Evaluation

We evaluated our system with a test dataset, which consisted of the transactions from 40 different market stocks. Our results showed that we can calculate the correlation matrix of the input market stocks in maximum 8ms, which means that it is almost in real time taking into account that the transactions arrive at our systems every second. Our system will offer really good performance for higher numbers of stock markets, taking into account that our system does not utilize the full communication bandwidth between the CPU and the DFE and it does not take advantage of the full parallelization level for these initial performance results. Also in our future plans, we aim to transform the current system in order to calculate the correlation estimator for a big number of stock markets over a sliding time window.

6.5 SVM Design

This section describes the design of the SVM Training method for hardware, given that we are using Maxeler technologies. Thus, we provide a top down analysis that contains a general block diagram of our architecture, we describe in detail the implemented blocks in the bottom up

modeling section, and finally we present debugging and verification issues that were met during this process.

6.5.1 Top Down analysis

In general, a top down analysis provides all necessary information to describe the different components of our implementation, as well as how they are interconnected. In particular for the SVM Training method we created the following design.

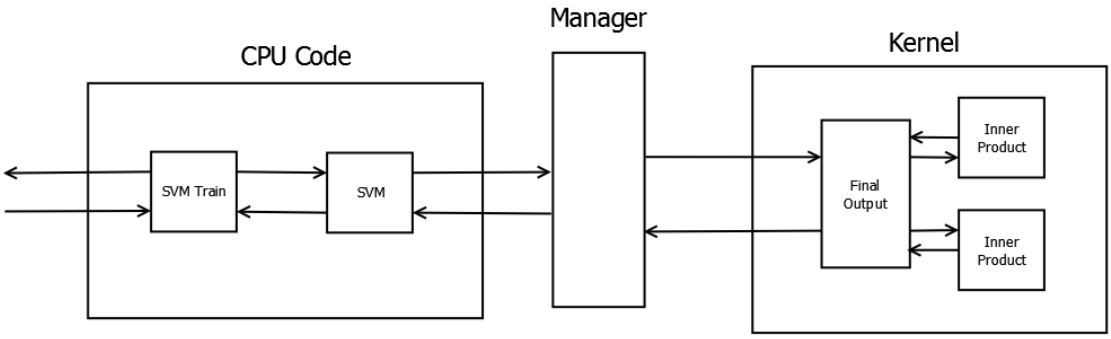


Figure 22: Abstract SVM training method block diagram for Maxeler platform

A detailed description of the implementation of our architecture follows in the Bottom Up modeling section. In this section we present the general contribution of each block to the overall system.

CPU Code

The CPU Code block contains software code that is written in C and C++. It contains two modules the SVM Train module and the SVM Module. The first receives as input the data file, processes its elements and creates the **svm_problem** structure described in 3.2.4. Then, the SVM module utilizes this structure in order to solve the dual optimization problem. Each iteration of the optimization problem recalculates and updates certain variables. Among these computations the most time consuming one involves the computation of the expression $y_i y_j K(x_i, x_j)$ presented in Section 5.5.4. Thus, we let the hardware side produce the expression's outcome so as to exploit all potential parallelism.

Note that the software side should provide the hardware side with all the necessary material. Thus, the output of the CPU code module consists of the training data, as well as of the size of the data in bytes.

Manager

In general, the Manager provides a predictable input and output streams interface to the Kernel. In particular, it comprises a Java API that allows configuring connectivity between Kernels and external I/O. That said it is evident why we have placed the specific module between the CPU code and the Kernel. The CPU code and Kernel 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, and the latter should return the final outcome.

Kernel

As mentioned above, the Kernel module uses the data instances of the software to produce the outcome of $y_i y_j K(x_i, x_j)$ presented in 3.2.4. However, due to the fact that it is not suitable to

calculate the exponent on the hardware side, we assign to the hardware side only the computation of the expression in the exponent:

$$gamma * dot(x_i, x_i) + dot(x_j, x_j) - 2 * dot(x_i, x_j).$$

We stress that function dot(x_i,x_j) computes the dot product of vectors x_i and x_j, and that gamma is a constant number. Briefly the main difference between implementing the aforementioned formula on hardware and implementing it on software is that in the first case the results of the three dot products can be produced simultaneously. Furthermore, the execution of the dot product function implies performing *n* multiplications and *n* totals, where *n* is the size of the vectors. However, multiplications can be produced in parallel and the same applies for calculating pairwise totals. Once the kernel has produced the final outcome of a pair of vectors it outputs the result back to the CPU code module via the Manager module.

So far, we have described the architecture of the SVM Training method, by presenting the general components, their respective subcomponents, as well as I/Os of the system. In the next section we will deepen in each component separately and we will completely illustrate the execution of the SVM Training method.

6.5.2 Bottom Up Modeling

In this section we begin by describing in detail each individual component of Figure 22 and then we mention how they are interconnected.

CPU Code

Both components included in the CPU Code module have been provided by LibSVM. Nevertheless, we have applied several modifications to the source code in order to allow the integration of software and hardware.

SVM Train

The SVM Train component is written in C. We have not applied any modifications to the corresponding LIBSVM code as its contribution to the SVM Training method is trivial. More specifically, SVM Train receives as input a file with data instances and sequentially parses this file in order to copy and organize all data into a data structure. The description of the file's format has been presented in Section 5.5.1. Furthermore, the created data structure is the **svm_prob** structure described in Section 5.5.4. Recall that this structure contains a two dimensional list with all elements of the input file, a list with their respective labels and an integer that denotes the number of the elements (rows in the file). In addition, SVM Train receives as input possible user inputs. These may define the type of kernel computation (rbf, linear, polynomial, sigmoid), certain constant values (gamma, degree, coefficient), or the type of classification that we want to execute. However, if the user does not provide any input, the program sets default values for these variables, if they are needed.

Once, **svm_prob** is created, SVM Train calls the *svm_train()* function, which is located in the SVM component. This function receives as input the **svm_prob** structure, as well as a structure that contains parameters provided by the user, or their respective default values. Moreover, *svm_train()* returns the SVM Training model which is the solution of the optimization problem, i.e., the result of the SVM Training method.

SVM

The SVM component is also part of the LIBSVM package and comprises the most essential steps of the SVM Training method. In particular, it uses the **svm_prob** structure to find the support vectors that will be used during the SVM Classification phase. We do not present a detailed description of the execution flow of the algorithm, as the respective theory is described in Section 4.4. However, note that during the training phase of the algorithm, LIBSVM solves the following primal optimization problem.

Solving an optimization problem implies an arbitrary number of iterations which depends on the size of the input data set. At the end of each iteration the variables of the primal optimization problem are updated based on the new calculations. Specifically, for the LIBSVM implementation the ***α*** values and the ***Q*** coefficients are reconstructed. This process is repeated until a stopping condition is met, or the algorithm converges. The processing for the optimization problem takes place in function *Solve()* of the SVM file. At this point we will continue with the description of our contribution to LibSVM. In order to do so, we need to mention that the functions described in Section 5.5.3 are used for the computation of coefficient ***Q*** and this is why they occupy a significant percentage of the execution time. Reconstructing the ***a*** values is not time consuming due to the fact that it only requires comparisons operations between relatively small vectors.

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\boldsymbol{\alpha}^T Q \boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \le \alpha_i \le 1/l, \qquad i = 1, \ldots, l, \\ & \boldsymbol{e}^T \boldsymbol{\alpha} \ge \nu, \quad \boldsymbol{y}^T \boldsymbol{\alpha} = 0, \end{aligned}$$

The computation of ***Q*** is executed in the *get_Q()* function which receives as input variables *i* and *length*. Both variables are integer numbers and the first denotes the index of a data instance selected during the specific optimization loop, whereas the latter variable shows the total number of the data instances. Note that only during the initialization phase of the optimization problem index *i* takes all values between 0 and length i.e. all data instances are considered by *get_Q()*. Furthermore, *get_Q*() outputs a vector of float numbers with size at most equal to *length*. We stress out that the output of *get_Q()* equals ***Q*** and more specifically:

$$Q_{i,j} = y_i y_j * K(x_i, x_j)$$

As mentioned in Section 6.4.1 due to the existence of the exponential in the selected kernel function (rbf) it is not efficient to implement the entire expression in hardware. Therefore, the final outcome of the above expression is computed in *get_Q()*. Moreover, *get_Q()* calls the functions that are implemented on hardware. However, hardware functions cannot receive as inputs multi-dimensional data structures, or structures of arbitrary size. Therefore, we reconstructed the two dimensional arbitrary sized data structure **svm_node** **x, described in Section 5.4.4, into a fixed size one dimensional vector. Let us call this vector **x_array.** In particular, its size equals the number of data instances*maximum feature number, which is the biggest possible size **svm_node** **x can have. In addition, when we call a hardware function we need to declare the fixed size of the input structures in bytes. Therefore, in *get_Q*()we compute this number, with respect to memory alignment constraints. Finally, once the hardware execution has produced the appropriate value, the remaining computations are performed in *get_Q()* and the results of the above expression are

stored in a vector of float numbers. This vector contains all the reconstructed Q values and is returned back to *Solve()* function.

Kernel

The Kernel component is written in Maxeler Java-like code and contains hardware implementations that perform numerical operations on data. We stress out that data is transferred from software to hardware one-by-one. More specifically, during one unit of time called a tick, the Kernel executes one step of computation, consumes one input value and produces one output value. Thus, neither can hardware receive all data at once, nor can it produce all outputs together. We could not overcome this limitation, which affects significantly our current architecture.

Final Output

To begin with, the Final Output hardware component receives as input a single data element of the**x_array** vector described in the SVM component, every tick. Similarly, it outputs a result every tick. A simple architecture of the *Final Output* is presented in Figure 23. Block *Data* represents the input stream and scalars, Inner Dot Product denotes the Inner Product component, and the computations of the figure indicate the calculation of the expression that we will describe immediately after.

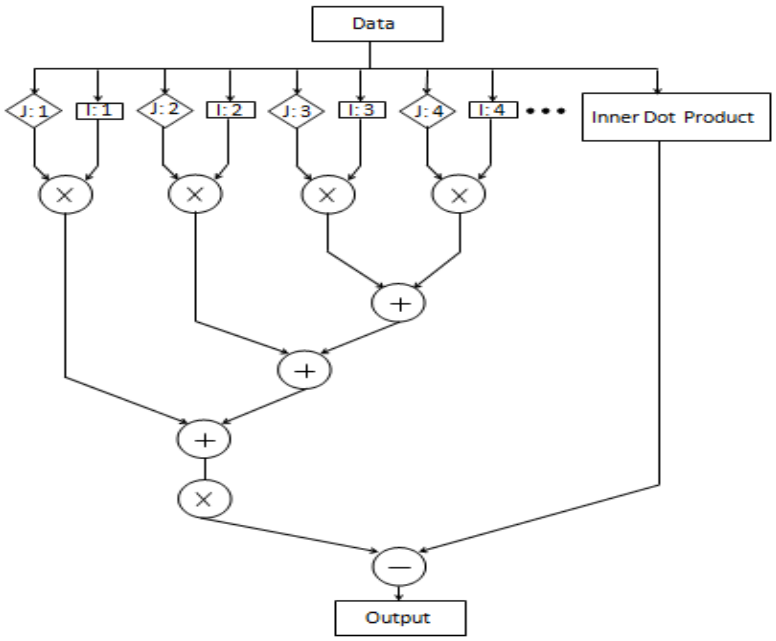


Figure 23: Block Diagram on Inner Dot Product Calculation

In order to clarify the role of hardware in the SVM method, we present the loop that is being unrolled on hardware.

$$for(j = 0; j < size\ of\ x\ array; j++)$$
$$\{$$
$$result[j] = -gamma * \left(x\_i + dot\left(x_{array\ [j]}, x_{array\ [j]}\right) - 2 * dot\left(x_{array\ [i]}, x_{array\ [j]}\right)\right);$$
$$\}$$

As we have previously mentioned, gamma is a constant coefficient that depends on the user. Moreover, the loop runs for number of times equal to the size of **x_array** described in the SVM module. An intuitive explanation of this loop is that **x_array** contains all elements of the input data

set. More specifically, cells 1 to the maximum number of the dataset features (max) correspond to the first data instance. The same applies for the rest of the data instances, i.e. the second data instance will be located from max + 1 to 2*max, the third data instance from 2*max+ 1 to 3*max and so on. Note that the loop is independent of index *i* , which means that all variables associated with this index remain fixed throughout a loop. For instance, x_i is the dot product of element **x_array**[i] with itself. Due to the fact that x_i remains the same during the whole loop, we can avoid computing it on hardware. Another element that remains constant throughout the loop is the data instance *i*, that corresponds to **x_array**[i]. In case max is relatively small, then we can assign each feature of data instance *i* to a variable and input these variables to hardware, instead of sending data instance *i* as a stream. Thus, we avoid the initialization overhead of an extra stream.

Moreover, due to the fact that hardware can receive and output only a single element at a time, we produce an actual result[j] output every max ticks. A more detailed explanation follows in the *InnerProduct* component. We are currently designing an architecture that will take into account this case and that will allow a bigger number of features, which will calculate more than a single result in parallel.

Inner Product

The Inner Product component only comprises numeric operations, since it calculates the dot product of two equal size vectors.

Given A = [A_1, A_2, … , A_n] and B = [B_1, B_2, … ,B_n] the dot product is defined as:

$$\boldsymbol{A \cdot B = \sum_{i=1}^{n} A_i B_i = \underbrace{A_1 B_1}_{1} + \underbrace{A_2 B_2}_{2} + \underbrace{A_3 B_3}_{3} + \underbrace{A_4 B_4}_{4} + \cdots + \underbrace{A_n B_n}_{n}}$$

$$\underbrace{\qquad\qquad}_{5} \qquad\qquad \underbrace{\qquad\qquad}_{6} \qquad \cdots \qquad \underbrace{\qquad\qquad}_{k}$$

In Section 5.4.3 we described how the computation of a dot product can be parallelized. Here, we present an illustrated version of the same description. More specifically, a parallel version of the dot product computes in parallel all products from **1** to **n.** Once these are produced, calculating the sum takes place. In particular, the sum of **1** and **2** (**5**) is calculated in parallel with the sum of **3** and **4** (**6**) and so on.

Although we would expect to produce all products in a single tick, this assumption does not apply due to the fact that we can only receive a single data input of the stream every tick. More specifically, in the first tick we will get element A_1, in the second tick we will get element A_2 and in the n tick we will have gotten all elements. Thus, the final outcome is produced after n ticks have passed, instead of in a single tick and this introduces delay to the hardware implementation.

Nevertheless, once the dot product of the two vectors is computed the result returns to Final Outcome and the outcome produced by hardware is returned to software.

6.5.3 Debugging Issues

As a first step the software needs to be ported and executed through the MaxCompiler platform. Then the Kernel that calculates the dot products was implemented, along with the appropriate Manager. The project was simulated, using the Maxeler simulator, keeping both the software and the simulated hardware running. The design was tested using small and constructed datasets, as

the simulation needs a significant amount of time to complete (100 times more than software. These slow run times are times for the simulated run of the design using CAD tools and not run times on actual hardware – only complete and debugged designs are run on actual hardware, because instrumentation of the design is easier in the CAD tool platforms (e.g. monitoring internal signals to see if data is progressing, if synchronization of resources is correct, and if operations are done properly) and in order to protect the valuable hardware resource from a design which could erroneously short-circuit a signal and damage the system. The simPrintf() debug function was used in order to print the intermediate results and prove correctness of the functions implemented. This function can be included in the hardware design coding, but is executed only on simulation mode.

6.5.4 Verification Issues

The verification of our design was done by executing the software for a certain dataset while gathering the results produced by the dot product function. The hardware implementation was tested with the same dataset and the results were compared one by one with the software's. The dataset had to be small enough in order for the simulation to complete in short amount of time. Larger datasets were also tested, but for a small amount of their products. We also used constructed (small) datasets in order to test boundary conditions.

6.5.5 Performance Evaluation

We evaluated the performance of the current hardware-based design with various numbers of input items but with a small number of features for each one of them. Our performance results showed degradation up to one order of magnitude vs. the performance achieved by the single thread software code. This is due to the low parallelization level achieved with the small number of features.

6.6 LDA Design

The SparseLDA algorithm (described in Section 5.6.4) is very efficient both on performance and on memory utilization. Basically, because the execution of the most time consuming function (sampleTopicsForOneDoc ()) is very efficient on software. Implementing an architecture for FPGAs would be meaningful only if we could execute the same operations faster.

The FPGAs gain performance over software implementations by utilizing parallelization as much as possible (resources are the limiting factor) and by allowing the implementation of really deep pipelines.

After careful inspection of the operations of the SparseLDA, and more specifically of the sampleTopicsForOneDoc() function we concluded that the operations executed are sequential and can't be parallelized. Mainly the bubble sort parts, which run in small parts of the topic array, cannot be pipelined. Basically the algorithm bubbles one value up the array and does not sort the whole array (which could be effective on FPGAs). The code that bubbles the new value up is shown below:

$$
\begin{aligned}
&while(i > 0 \;\&\&\; currentypeTopicCounts[i] > currentTypeTopicCounts[i-1]\\
&\{\\
&\quad int\ temp = currentTypeTopicCounts[i];\\
&\quad currentTypeTopicCounts[i] = currentTypeTopicCounts[i-1];\\
&\quad currentTypeTopicCounts[i-1] = temp;\\
&\quad i--;\\
&\}
\end{aligned}
$$

There is some parallelization that can be utilized, mostly on arithmetic operations for the probabilities calculations (pipelined floating point cores), but it cannot hide the sequential operations' execution time.

The SparseLDA has a parallel version where each thread samples different documents. This parallelization can be utilized by streaming multiple docs in the FPGA (up to 8 streams on Maxeler machines). This parallelization factor isn't enough for the FPGA to be faster than the software, as the clock frequency for the Maxeler system is 150 MHz which is at least 20 times lower than the clock frequency of a modern high end CPU. Lastly, the sequential nature of the SparseLDA algorithm led us to the conclusion that even an efficient hardware-based system will have lower execution times than the corresponding software solution.

6.7 Hardware based system evaluation

The previous Sections presented four initial different hardware-based systems, i.e. the SVM algorithm, the Count-Min data structure, the Exponential Histogram sketch data structure and the Hayashi-Yoshida correlation estimator. We tested our systems with various input datasets. As our initial non-optimized performance results indicate, Table 2, our systems offer good performance achievements vs. the corresponding software solutions. In our future plans, we aim to take advantage of the full parallelization level that the DFEs can offer and the full communication bandwidth for the data I/O in order to take much better performance results.

Lastly, the LDA algorithm was analyzed and an initial hardware-based solution was evaluated. We reached to the conclusion that an hardware implementation of such complex and low-parallelized algorithm would not give better performance results.

Algorithm	HW-based systems performance
SVM algorithm	One order of magnitude worse performance vs. SW
Count-Min algorithm	About 7x better performance vs. SW
Exponential Histogram	About 4x better performance vs. SW
Hayashi-Yoshida Correlation Estimator	About 8ms for computing Correlation Matrix for 40 Stock Markets

Table 2: HW-based Systems Performance

It may appear that the results in the table, above, are rather poor, however, there are several reasons why they are actually very good: the initial version of each design aims at correct execution of the corresponding algorithm, so that it will serve as a reference design. Subsequent versions exploit parallelism with techniques including pipelining, multiple functional units, data forwarding (i.e. setting up buffers to deliver data where it needs to be for processing, rather than do so through the memory), etc. The key factors that determine whether an initial design seems promising are three: (a) that the design does not fully utilize the resources of the hardware, i.e. there exist substantial resources that can be exploited in subsequent versions of the design, (b) that the system is not Input/Output limited, i.e. a speedup of the initial design will not lead to no performance benefit due to the impossibility of getting data in and out of the new design, and (c) the intrinsic parallelism of the algorithm, i.e. that there are aspects of the algorithm which can be executed in parallel, given the resources and an appropriate design. In the case of the four designs, above, and especially in the case of the last three algorithms we are highly optimistic that reconfigurable computing will offer very substantial speedups.

7 Conclusions

The QualiMaster project fuses several state-of-the-art technologies in order to process Big Data in streaming form, and extract in real-time useful information. The various processing requirements as well as the characteristics of the data require the processing to be done in an adaptive pipeline, the QualiMaster pipeline, an integral part of which is the reconfigurable hardware (FPGA computing) processing element. In reconfigurable computing algorithms are mapped directly to hardware, which is a different computational paradigm vs. that of algorithm execution on general-purpose computers (including single computers and multiprocessors, clusters, cloud computing, etc.)The reconfigurable processing element of the QualiMaster pipeline needs to operate seamlessly with the software, run the same algorithms, and be employed on-demand. The above considerations are the scope of the WP3 of QualiMaster. This present deliverable D3.1 details the year's progress in WP3, as well as the Work Package's interaction with all other Work Packages of the QualiMaster project.

A summary of the progress is that the algorithms chosen by the software partners, namely, Support Vector Machines (SVM), Latent Dirichlet Analysis (LDA), Count Min, Exponential Histogram, and Hayashi-Yoshida Correlation Estimator were all studied with respect to their potential for hardware implementation. Four of these algorithms (all, except for LDA) were found to be suitable to a greater or lesser extent (i.e. they seem promising for exceptional or good speedups vs. software execution, respectively). In addition, a MaxelerC-Series FPGA-based computer of the QualiMaster partner TSI was integrated seamlessly with the Storm environment which will be employed in the QualiMaster adaptive pipeline. No such result has been reported in the literature to date, and the ability to easily and seamlessly direct data into the Maxeler reconfigurable processor as needed will no doubt give the QualiMaster project very interesting capabilities for realistic experiments.

To conclude, WP3 is progressing according to plan and in a timely fashion, with a number of technical issues, such as the integration of the reconfigurable computing node with the Storm platform, already having been solved. The promising results in terms of potential for hardware (FPGA) implementation in four of the five algorithms chosen by the software partners allow for great optimism that the QualiMaster adaptive pipeline will demonstrate performance capabilities which cannot be achieved with conventional distributed computing methods, and work is underway to implement and test these algorithms on reconfigurable computing platforms. Last but not least, the very close cooperation of the software and the hardware partners of the project is very beneficial to all, as the software partners are exposed to computational capabilities which would otherwise be out-of-reach and hence they have more tools with which to implement their algorithms of choice, whereas the hardware partners have meaningful driving problems, complete with realistic Input/Output requirements, real-time operation requirements, and reference software designs for the assessment of their work.

References

[1] Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., &Zdonik, S. (2003). “Aurora: a new model and architecture for data stream management”. *The VLDB Journal—The International Journal on Very Large Data Bases*, *12*(2), (pp. 120-139).

[2] Chakravarthy, S., & Jiang, Q. (2009). *Stream data processing: a quality of service perspective: modeling, scheduling, load shedding, and complex event processing* (Vol. 36). Springer.

[3] Klein, A., &Lehner, W. (2009). “Representing data quality in sensor data streaming environments”.*Journal of Data and Information Quality (JDIQ)*, *1*(2), 10.

[4] Geisler, S., Weber, S., &Quix, C. (2011, November). “An ontology-based data quality framework for data stream applications”. In *16th International Conference on Information Quality*.

[5]Pang, B., Lee, L., &Vaithyanathan, S. (2002, July). “Thumbs up?: sentiment classification using machine learning techniques”. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10* (pp. 79-86).Association for Computational Linguistics.

[6]Vapnik Vladimir, N. (1995). *The Nature of Statistical Learning Theory*.Spring-Verlag, New York.

[7] Cortes, C., &Vapnik, V. (1995). “Support-vector networks”.*Machine learning*, *20*(3), (pp. 273-297).

[8] Drucker, H., Wu, S., & Vapnik, V. N. (1999). Support vector machines for spam categorization. *Neural Networks, IEEE Transactions on*, *10*(5), (pp. 1048-1054).

[9] Dumais, S., Platt, J., Heckerman, D., &Sahami, M. (1998, November). “Inductive learning algorithms and representations for text categorization”.In *Proceedings of the seventh international conference on Information and knowledge management* (pp. 148-155).ACM.

[10] Joachims, T. (1998). “Text categorization with Support Vector Machines: Learning with many relevant features”. *Machine Learning: ECML-98*, *1398*, (pp. 137-142).

[11] Manning, C. D., Raghavan, P., &Schütze, H. (2008). *Introduction to information retrieval* (Vol. 1, p. 6). Cambridge: Cambridge university press.

[12] Aggarwal, C. C., &Zhai, C. (2012). “A survey of text classification algorithms”. In *Mining text data* (pp. 163-222). Springer US.

[13]Aizerman, A., Braverman, E. M., &Rozoner, L. I. (1964). “Theoretical foundations of the potential function method in pattern recognition learning”. *Automation and remote control*, *25*,(pp. 821-837).

[14]Boser, B. E., Guyon, I. M., &Vapnik, V. N. (1992, July). “A training algorithm for optimal margin classifiers”.In *Proceedings of the fifth annual workshop on Computational learning theory* (pp. 144-152).ACM.

[15] Chang, C. C., & Lin, C. J. (2011). “LIBSVM: a library for support vector machines”. *ACM Transactions on Intelligent Systems and Technology (TIST)*, *2*(3), 27.Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[16] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). “The WEKA data mining software: an update”. *ACM SIGKDD explorations newsletter*, *11*(1), (pp. 10-18).

[17] Datar, M., Gionis, A., Indyk, P., &Motwani, R. (2002). “Maintaining stream statistics over sliding windows”.*SIAM Journal on Computing*, *31*(6), (pp. 1794-1813).

[18] www.cs.princeton.edu/~blei/lda-c/

[19] Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). “Latent dirichlet allocation”.*The Journal of machine Learning research*, *3*, (pp. 993-1022).

[20] Shah, M., Miao, L., Chakrabarti, C., &Spanias, A. (2013, May). “A speech emotion recognition framework based on latent Dirichlet allocation: Algorithm and FPGA implementation”. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on* (pp. 2553-2557).

[21] http://www.r-project.org/

[22] http://mallet.cs.umass.edu/

[23] Yan, F., Xu, N., & Qi, Y. (2009). “Parallel inference for latent dirichlet allocation on graphics processing units”.In *Advances in Neural Information Processing Systems* (pp. 2134-2142).

[24] Newman, D., Smyth, P., Welling, M., & Asuncion, A. U. (2007). “Distributed inference for latent dirichlet allocation”.In *Advances in neural information processing systems* (pp. 1081-1088).

[25] Smyth, P., Welling, M., & Asuncion, A. U. (2009). “Asynchronous distributed learning of topic models”. In *Advances in Neural Information Processing Systems* (pp. 81-88).

[26] Yao, L., Mimno, D., & McCallum, A. (2009, June). “Efficient methods for topic model inference on streaming document collections”. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 937-946).ACM.

[27] Pell, O., Averbukh, V. (2012). "Maximum Performance Computing with Dataflow Engines".*Computing in Science &Engineering* , vol.14, no.4, (pp. 98-103).

[28] Pell, O., Mencer, O., Tsoi, K. H., &Luk, W. (2013). “Maximum performance computing with dataflow engines”.In *High-Performance Computing Using FPGAs* (pp. 747-774).Springer New York.

[29] *Multiscale Dataflow Programming* ,Maxeler Technologies Ltd, London, UK, 2014

[30] *Programming MPC Systems*, Maxeler Technologies Ltd, London, UK, 2014

[31] https://storm.apache.org/

[32] SWIG: http://www.swig.org/index.php

[33] http://en.wikipedia.org/wiki/Network_socket

[34]http://www.cs.rutgers.edu/~muthu/massdal-code-index.html

[35] Papapetrou, O., Garofalakis, M., & Deligiannakis, A. (2012). “Sketch-based querying of distributed sliding-window data streams”. *Proceedings of the VLDB Endowment*, *5*(10), (pp. 992-1003).

[36] Hayashi, T., & Yoshida, N. (2005). “On covariance estimation of non-synchronously observed diffusion processes”.*Bernoulli*, *11*(2), (pp. 359-379).

[37] Arlitt, M., &Jin, T. (2000). “A workload characterization study of the 1998 world cup web site”. *Network, IEEE*, *14*(3), (pp. 30-37).

[38] http://en.wikipedia.org/wiki/Program_optimization

[39] Amdahl, G. M. (1967). “Validity of the single processor approach to achieving large scale computing capabilities”. In Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, ACM, (pp. 483-485).

[40] http://www.cs.rutgers.edu/~muthu/massdal-code-index.html

[41] Cormode, G., &Muthukrishnan, S. (2005). "An improved data stream summary: the count-min sketch and its applications. Journal of Algorithms, 55(1), (pp. 58-75)."

[42] Baeza-Yates, R., & Ribeiro-Neto, B. (1999). Modern information retrieval (Vol. 463). New York: ACM press.

[43] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA data mining software: an update. ACM SIGKDD explorations newsletter, 11(1), 10-18.

[44] Hsu, C. W., & Lin, C. J. (2002). A comparison of methods for multiclass support vector machines. Neural Networks, IEEE Transactions on, 13(2), 415-425.

[45] Joachims, T. (2006, August). Training linear SVMs in linear time.InProceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 217-226).ACM.

[46] Fan, R. E., Chen, P. H., & Lin, C. J. (2005). Working set selection using second order information for training support vector machines. The Journal of Machine Learning Research, 6, 1889-1918.

[47] Chang, C. C., & Lin, C. J. (2011). LIBSVM: a library for support vector machines. ACM Transactions on Intelligent Systems and Technology (TIST),2(3), 27.

[48] Cadambi, Srihari, et al. "A massively parallel FPGA-based coprocessor for support vector machines." Field Programmable Custom Computing Machines, 2009. FCCM'09.17th IEEE Symposium on. IEEE, 2009

[49] Papadonikolakis, Markos, and C. Bouganis. "A novel FPGA-based SVM classifier."Field-Programmable Technology (FPT), 2010 International Conference on.IEEE, 2010.

[50] Pina-Ramfrez, O., Raquel Valdes-Cristerna, and Oscar Yanez-Suarez. "An FPGA implementation of linear kernel support vector machines." Reconfigurable Computing and FPGA's, 2006.ReConFig 2006.IEEE International Conference on.IEEE, 2006.

[51] Lai, Y. K., Wang, N. C., Chou, T. Y., Lee, C. C., Wellem, T., &Nugroho, H. T. (2010, June). “Implementing on-line sketch-based change detection on a netfpga platform”.In 1st Asia NetFPGA Developers Workshop.

[52] Lai, Y. K., & Byrd, G. T. (2006, December). “High-throughput sketch update on a low-power stream processor”.In Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems (pp. 123-132).ACM.

[53] Thomas, D., Bordawekar, R., Aggarwal, C. C., & Yu, P. S. (2009, March). “On efficient query processing of stream counts on the cell processor”. In Data Engineering, 2009.ICDE'09. IEEE 25th International Conference on (pp. 748-759). IEEE.

[54] Wellem, T., Lai, Y. K., Lee, C. C., & Yang, K. S. (2011, October). “Accelerating Sketch-based Computations with GPU: A Case Study for Network Traffic Change Detection”.In Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems (pp. 81-82).IEEE Computer Society.

[55] Wellem, T., & Lai, Y. K. (2012, December). “An OpenCL Implementation of Sketch-Based Network Traffic Change Detection on GPU”. In Parallel Architectures, Algorithms and Programming (PAAP), 2012 Fifth International Symposium on (pp. 279-286). IEEE.

[56] Fowers, J., Brown, G., Cooke, P., &Stitt, G. (2012, February). “A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications”.In Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays (pp. 47-56).ACM.

[57] Qian, J. B., Xu, H. B., DONG, Y. S., Liu, X. J., & Wang, Y. L. (2005). “FPGA acceleration window joins over multiple data streams”. Journal of Circuits, Systems, and Computers, 14(04), 813-830.

[58] Ureña, J., Mazo, M., Garcıa, J. J., Hernández, Á.,& Bueno, E. (1999). “Correlation detector based on a FPGA for ultrasonic sensors”. Microprocessors and Microsystems, 23(1), 25-33.

[59] Fort, A., Weijers, J. W., Derudder, V., Eberle, W., &Bourdoux, A. (2003, April). “A performance and complexity comparison of auto-correlation and cross-correlation for OFDM burst synchronization”. In Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03). 2003 IEEE International Conference on (Vol. 2, pp. II-341).IEEE.

[60] Lindoso, A., &Entrena, L. (2007). “High performance FPGA-based image correlation”. Journal of Real-Time Image Processing, 2(4), 223-233.

[61] Liu, X., Sun, D. J., Teng, T. T., & Tian, Y. (2013). “FPGA Implement of Multi-Channel Real-Time Correlation Processing System”. Applied Mechanics and Materials, 303, 1925-1929.