www.qualiMaster.eu

QualiMaster

A configurable real-time Data Processing Infrastructure mastering

autonomous Quality Adaptation

Grant Agreement No. 619525

Deliverable D 3.4

Work-package	WP3: Optimized Translation to Hardware
Deliverable	D3.4: Optimized Translation of Data Processing Algorithms to Hardware
Deliverable Leader	Telecommunication System Institute (TSI)
Quality Assessor	H. Eichelberger
Estimation of PM spent	6
Dissemination level	Public (PU)
Delivery date in Annex I	31.12.2016
Actual delivery date	09.01.2017
Revisions	3
Status	Final
Keywords:	QualiMaster, Adaptive Pipeline, Reconfigurable Computing, FPGA Computing, Hardware, Support Vector Machines (SVM), Count Min, Exponential Histogram, Hayashi-Yoshida Correlation Estimator, Mutual Information, Transfer Entropy, design automation tool.

Disclaimer

List of Authors

Partner Acronym	Authors
TSI	E.Sotiriades, G. Chrysos, P. Malakonakis, I. Papaefstathiou, A. Dollas
MAX	T. Becker, N. Voss

Table of Contents

Executive summary

The present deliverable, D3.4 is the last deliverable of WP3 in the QualiMaster project, and it completes the project goals at the end of year 3. Despite this deliverable having only a very short person-effort (6 person-months), several significant contributions were made, these being that:

- The design automation tool that Maxeler was to develop was completed (in beta version) and it works as expected
- The design automation tool was evaluated by TSI and it was found to afford design efficiency that was impossible to have until now,
- There was further refinement on the Mutual Information (MI) and Transfer Entropy (TE) implementations in hardware

An additional contribution of the QualiMaster hardware team (TSI/MAX) during this project period was the development of additional interfaces towards further integration of the custom hardware to the QualiMaster adaptive pipeline, however, these are described in other project deliverables.

As this is the final WP3 deliverable for the QualiMaster project we feel that we need to briefly state the overall contributions of WP3 to the QualiMaster adaptive pipeline, as well as the potential for further research and exploitation opportunities: Tight integration of custom hardware with software has not been done or reported in this scale to date – custom hardware traditionally runs computationally-intensive kernels in a fairly standalone fashion. Integrating custom hardware to software in an adaptive application platform and in a distributed environment at that necessitated the development of many interfaces the addressing of a multitude of research and development issues. The mere fact that the QualiMaster pipeline can change from software execution to hardware in an adaptive and programmable way is an attestation of the value of this approach.

1 Introduction

Throughout the QualiMaster project the WP3 aimed at the analysis of algorithms, their direct mapping to hardware in a series of evolving architectures (aiming at ever increasing performance), performance evaluation, incorporation to the QualiMaster pipeline, and design iterations for improvements throughout the project. In addition, and this is the main contribution of D3.4, a tool was developed (called Kernel Merger), both in order to help the designer take better advantage of the hardware resources vs. hand coding, and, by utilization of the same resources in different instantiations to be able to map larger designs onto the same hardware. These capabilities are achieved through utilization of resources by multiple designs as long as they are not needed simultaneously, as determined by graph analysis (performed by the tool). The deliverable D3.4 presents Kernel Merger's capabilities, usage, and performance evaluation. Besides the tool, deliverable D3.4 summarizes improvements and performance evaluation on previous results, as well as synergies with other QualiMaster work packages (mostly with WP5) in order to complete the seamless QualiMaster pipeline. The tool itself will be further exploited, as described in WP7, whereas many of the previously developed applications will become open-source modules, per D7.4.

More specifically, in D3.4 the architecture and structure of the Kernel Merger tool are detailed, including the way in which it is used, followed by experiments which demonstrate its usefulness and capabilities. The fact that this is a new tool in beta version means that not all libraries are presently available to the designer however, many of the QualiMaster-developed algorithms as well as additional ones were evaluated and they demonstrated the capabilities of the tool. The total number of designs which were evaluated exceeds the goals of the project, as described in the Technical Annex (DoW). All these goals were achieved despite D3.4 having a total of only 6 person-months for all partners.

With the extensive presentation of the Kernel Merger tool and the hardware mapping of all suitable algorithms and their connection to the QualiMaster pipeline (in conjunction with WP5) the WP3 is completed, having achieved all of the goals which had been set for the project. This was no small achievement, as the QualiMaster project is the first project to incorporate reconfigurable hardware (the Maxeler nodes) into a reconfigurable and adaptive pipeline, meaning that there is very tight coupling of software and hardware, with each component changing dynamically and adaptively during operation – the standard usage of reconfigurable hardware is to run computationally intensive kernels on its own. Despite the technical challenges which are described in all previous deliverables, e.g. incorporation into the Storm environment, I/O issues, MaxCompiler issues, the QualiMaster project met its goals and demonstrated that in the kind of applications for which it was designed, (a) the usage of dedicated hardware is necessary and (b) the dynamic adaptation of software and hardware is both desirable and feasible to do.

As hardware designers, we consider the greatest achievement of the QualiMaster project was the demonstration of the seamless operation of vastly different software and hardware technologies in a single adaptive pipeline in order to deliver adaptive and workload-determined high performance. The QualiMaster project was clearly ahead of its time (as attested by industrial projects emerging now but not having some of the QualiMaster adaptive pipeline capabilities), and it was in the correct direction: in the course of its duration two significant high-visibility projects from industry show that custom hardware (and reconfigurable computing in particular – the QualiMaster technology of choice) are becoming mainstream. Microsoft already runs its search engine Bing on reconfigurable platforms, thus integrating software with custom hardware in a computationally efficient (but not adaptive) environment, and, Amazon has announced that it makes custom hardware (also reconfigurable, a.k.a. FPGA) available to users in a cloud-like manner, however, it completely lacks the tools in order for the user to take advantage of the hardware resources. Not only QualiMaster has demonstrate in actual, real-time applications how the custom hardware can be switched in and out of the adaptive pipeline, but also through the project a new tool has been developed (and described in the present deliverable) which re-uses hardware resources among applications, making designs more efficient. The above results indicate that the EU can be competitive in this, emerging form of computing, by developing new tools and services, some of

which will be developed through further exploitation of the QualiMaster WP3 results by its industrial partners.

2 Mutual Information and Transfer Entropy Overview

This section clarifies the reasons why the performance of the hardware implementations of Mutual Information and Transfer Entropy are not competitive against the software version implemented on Storm. In D5.4 we presented the performance results of MI/TE hardware implementations and they are shown to be not as efficient as the Streaming Software version.

2.1.1 MI/TE hardware architectures summary

In this section, we present a summary of the architectures implemented for MI and TE. This section is included so the reader does not need to refer to previous deliverables (D3.2, D3.3).

2.1.1.1 MI hardware architectures summary

The first architecture was presented in D3.2 and its main characteristic was that if consisted only of 2 MI calculation cores. The PDFs, p(x, y), p(x), p(y), are streamed into hardware for the calculation of Mutual Information. The length of the streams depends on the number of Bins that is R^2. Each stream of the p(x) and p(y) is streamed R times, with R being the number of Bins, in order to match the size of p(x, y). In order to stream the data into the DFEs in the correct order either the values in p(x) or p(y) have to be copied R times each, while the other is streamed R times. The processing is fully pipelined meaning that each clock cycle the appropriate values of p(x,y), p(x) and p(y) have to arrive in the cores at the same time for the correct results to be produced. The MI calculation utilizes 3 of the 8 streams available on the Maxeler System. In order to further accelerate the application, by utilizing the parallelization, the PDFs were divided by 2, providing 6 streams, 5 streams more precisely. Basically, the stream that is streamed R times is now streamed R/2 times. Also 2 hardware cores are responsible of calculating the partial MI results. This allowed double bandwidth utilization and thus even better performance. Also, even with two cores the FPGA resource Utilization remains at about 10% of the total available resources. That architecture provided speedups of up to 9x over the equivalent software implementation.

The 2nd architecture is an improvement of the first architecture. Some of the improvements of the 2nd architecture could be applied to the software which improved its performance significantly (up to 4x speedup over the initial software). The main addition of the 2nd architecture is that now there are 3 cores responsible of calculating the MI, instead of 2 that the previous version had. Further efficiency was achieved by dividing the PDF sequences p(x) and p(x,y) by 3, thus producing sub-streams of sizes R/3 and R^2/3 respectively. Each R^2/3 sub-stream is streamed R/3 times into the FPGA for alignment purposes. Thus, our proposed solution reaches almost the optimal bandwidth utilization of 7 out of 8 streams, with respect to the physical restrictions of the system and the nature of the input streams based on the computational measurement. The use of 3 parallel processing hardware components leads to a significantly better performance. However, even with the deployment of 3 cores, the occupied FPGA resources were less than 12% of the total available resources. The performance improvement over the new software was of about 5x. The Multi-DFE architecture was also presented in D3.3 which provided a 15x speedup over the software.

The final architecture was implemented during the visit at Maxeler, where we were introduced on the way to iterate over data written in the LMEM. Basically, the usage of the LMEM, in order to take advantage of its bandwidth, allows an increase in performance. This architecture is implemented on the next generation Maxeler platform, the Maia. The difference between the previous architecture (Version 2) is that the p(x) and p(y) PDFs are streamed from LMEM (DDR Memory) instead from PCIe. This allows the whole PCIe bandwidth to be utilized by the p(x,y) PDF. So, we can use all the 8 streams from the pc to the DFE for p(x,y). First the p(x) and p(y) are written to the LMEM of the DFE from the host. This includes a call to the DFE to write the LMEM with the provided data. Then the p(x,y) is divided into 8 streams and is streamed to the DFE via the 8 available PCIe streams. This allows 8 MI calculation cores to be placed into the DFE that work in parallel. The 8 partial results are then accumulated in hardware and the result is streamed to the host. This architecture provided speedups of about 7x vs. the Version 2 software.

2.1.1.2 TE hardware architectures summary

Similarly to Mutual Information we have 3 architecture generations for Transfer Entropy. The 1st architecture was also presented in D3.2 The TE calculation starts with the PDF estimation, which is done on software, then the PDFs are streamed to the DFE where TE is calculated. The PDFs p(x), p(x,y), $p(x_{n+1},x)$ and $p(x_{n+1},x,y)$ are streamed to the DFE. The length of the streams is R^3, with R being the number of bins, and R^3 is the size of $p(x_{n+1},x,y)$. In order for the rest PDFs/streams to match the stream size of $p(x_{n+1}, x, y)$, p(x) is copied R times and streamed R times, p(x, y) is streamed R times and $p(x_{n+1}, x)$ is copied R times. These array copies increase drastically the memory requirements of the applications. These streams utilize 4 of the 8 streams available on the Maxeler System. In order to further accelerate the application, the PDFs were divided by 2, providing 8 streams, actually 6 streams are used as 2 streams remain the same for both cores, p(x) and p(x,y), which are streamed R/2 times. Also 2 hardware cores are responsible of calculating the partial TE results. This allowed more bandwidth utilization and thus even better performance. Also, even with two cores the FPGA resource Utilization remains at about 10% of the total available resources, indicating that the limiting factor remains the PCIe bandwidth, just like in MI. This architecture provided a speedup of up to 5.5x.

The Version 2 implementation followed (D3.3) with the 2nd architecture. The same optimizations that were used in MI are also implemented for TE. Initially the optimizations were done on hardware, and then they were integrated also in software, providing the equivalent performance increase. The basic architecture difference between the 2 hardware versions is that the second has 3 TE calculation cores instead of 2. Another basic difference is that the data of the 1D PDFs don't have to be copied R^2 times and the 2D R times, in order to match the $p(x_{n+1},x,y)$. This reduced the memory needs of the application for the hardware implementation by a factor of 4. Also, some more optimizations were applied in order to optimize the memory accesses, which were also applied to the software version that also provided an increase in performance. Similarly, to MI, TE computation begins with estimating the required PDF values. This operation is performed on the CPU. Then, the PDF estimations p(x), p(x,y), $p(x_{n+1},x)$ and $p(x_{n+1},x,y)$ are streamed to the FPGA. The length of each stream has to be equal to R^3. In order to align the different sized PDF sequences, we streamed the PDF $p(x_{n+1},x,y)$ of size R^3 once, the PDF p(x) of length R repeatedly every R clock cycles and this was performed R times, the PDF p(x,y) of size R^2 repeatedly every R clock cycles and the PDF $p(x_{n+1},x)$ of size R^2, R times. Given that the system is fully pipelined the final output is produced after approximately R^3 clock cycles, since it is the time required for the entire PDF $p(x_{n+1},x,y)$ estimation to become accessible by the FPGA. Further efficiency was achieved by dividing sequences $p(x_{n+1},x,y)$ and p(x,y) by 3, thus producing sub-streams of sizes $R^3/3$ and $R^2/3$ respectively. Each $R^2/3$ sub-stream is streamed R/3 times to guarantee alignment. In addition to optimal bandwidth utilization, 3 TE hardware cores were mapped to the FPGA for parallel processing to take place. However, these processing cores only consumed a small percentage of the total available resources, a percentage less than 13%, as the parallel processing is restricted by the number of the available PCI streams that can be used to send data to the DFE. The Version 2 implementation provided a speedup of 5.9 vs. the Version 2 software. Finally, the Multi-DFE architecture was implemented, which with 4 DFEs provided a speedup of 17x.

The final architecture for TE was also designed and implemented during the visit at Maxeler Technologies. The usage of the LMEM, in order to take advantage of its bandwidth, allows an increase in performance just like in the Mutual Information case. The increase is even more visible in TE as the execution time is much larger and thus the write LMEM function calls take a very small portion of the execution time. This architecture is implemented on the next generation Maxeler platform, the Maia. The difference between the previous architecture is that the p(x), p(x,y) and $p(x_{n+1},x)$ PDFs are streamed from LMEM instead from PCIe. This allows the whole PCIe bandwidth to be utilized by the $p(x_{n+1},x,y)$ PDF. So we can use all the 8 streams from the CPU to the DFE for $p(x_{n+1},x,y)$. First the p(x), p(x,y) and $p(x_{n+1},x)$ are written to the LMEM of the DFE from the host. This includes a call to the DFE to write the LMEM with the provided data. Then the $p(x_{n+1},x,y)$ is divided into 8 streams and is streamed to the DFE via the 8 available PCIe streams. Also, p(x,y) is divided into 8 streams in order to have the correct data sequence for the calculation. Actually 8 streams are initiated from different addresses of p(x,y) from LMEM. This allows 8 TE

calculation cores to be placed into the DFE that work in parallel. The 8 partial results are then accumulated in hardware and the result is streamed to the host. This architecture provided speedups of up to 10x vs. the Version 2 software.

2.1.2 MI/TE hardware architectures tradeoffs

2.1.2.1 MI hardware architectures tradeoffs

In this section, we will evaluate in brief the architecture versions of our implementations for Mutual Information from the initial version till the final architecture that provided the best results with respect to performance. The initial implementation was based on the first software implementation of the MI algorithm. This architecture provided a SpeedUp of up to 9 vs the initial software implementation. This performance increase was not significant, while the architecture used only 10% of the available resources with 2 calculation cores. Thus, the target of the Version 2 implementation was the more efficient utilization of the available bandwidth. This implementation required less memory, while the bandwidth to the DFE was better divided between the PDFs. The Version 2 architecture was 2x faster than the initial version, partly due to 3 calculation cores and partly due to more efficient bandwidth utilization. The Version 2 implementation led to the implementation of the Version 2 software, using similar techniques, which led to better cache memory usage and provided a 2x faster software. The final architecture utilizes the LMEM (DRAM) available on the Maia cards. The 1D PDFs are stored on the LMEM and are streamed from there to the DFE, while the 2D PDF is streamed through the PCIe. This allowed the utilization of the LMEM bandwidth in parallel with the PCIe. The final architecture is 1.6x faster than 2nd architecture and 7.7x faster than the Version 2 software implementation. The final architecture has 8 calculation cores allowing much higher parallelization.

2.1.2.2 TE hardware architectures tradeoffs

In this section, we will describe in brief the generations of our implementations for TE from the initial version till the final architecture that provided the best results with respect to performance. The initial implementation was based on the first software implementation of the TE algorithm. This architecture provided a speedup of up to 5.5 vs the initial software implementation. This performance increase was not significant, while the architecture used only 12% of the available resources with 2 TE calculation cores. The conclusion was that we should make an architecture that utilizes more efficiently the available PCIe bandwidth. This leads to the Version 2 implementation. This implementation required less memory, as we do not need to make the 1D and 2D PDFs as big as the 3D PDF, while the bandwidth to the DFE was better divided between the PDFs. The Version 2 architecture was about 2x faster than the initial version, partly due to the 3rd core integration and partly due to bandwidth and memory optimizations. The Version 2 implementation lead to the implementation of the Version 2 software, using similar techniques, which led to better cache memory usage and provided a 1.8x faster software. The V2 hardware implementation is up to 5.9x faster than the V2 software, while the multi DFE implementation with 4 DFEs is 17x faster. The final architecture utilizes the LMEM (DRAM) available on the Maia cards. The 1D PDF as well as the 2D PDFs are stored on the LMEM and are streamed from there to the DFE, while the 3D PDF is streamed through the PCIe. This allowed the utilization of the LMEM bandwidth in parallel with the PCIe. The final architecture is 1.8x faster than Version 2 architecture and 9x faster than the Version 2 software implementation. The main reason is the parallelization by 8, as the final architecture integrates 8 TE calculation cores, instead of 3 of V2.

2.1.3 MI/TE Evaluation

In D5.4 the hardware implementations of Mutual Information and Transfer Entropy performance results are shown to be less efficient than the Streaming Software versions. The basic reason of these results is the nature of the data from the stock markets. Actually, the data rates are very

small per market player in order to justify the use of a hardware server. In the initial implementation, we considered time series of lengths from 10^5-10^9. This is the reason why we investigated large number of bins, in order to be able to have as accurate as possible pdf's (probability density functions) for such lengths of time series. Also, the hardware implementation is competitive for large number of bins. The initial approach was dictated mainly by Maxeler consultants and the previously published work (by a very respectable research group), presented in [1] for Transfer Entropy. In [1] they use large datasets in order to estimate the pdfs the number of bins used is from 100-1200. Also, the pdfs are stored in arrays just like in our case. If we consider these and the fact that we take one value for each market player each second, we would need 10^5-10^9 seconds in order to store the amount of data needed to create the pdfs.

The Storm/Streaming implementation of the algorithms takes advantage of the nature of the data by using hashmaps in order to maintain the pdfs. In the real data case the hashmaps are very small as the highest amount data end up on the same bins. This reduces the amount of mathematical operations and memory accesses for the MI/TE calculations.

In our attempt to make the hardware implementation competitive again we considered the hashmap approach with, unfortunately, not so promising results. The problem starts from the Maxeler server requirement that the data have to be consecutive on the memory, and also the different streams from the host to the DFE have to be synchronized. As such, even though the resulting hashmaps were very small, they would have to be copied into arrays, and in the correct order, in order for the pdfs to be streamed in the correct order to the DFEs. We compared the time needed for the copy from the hashmaps to the arrays and the actual calculation of the MI/TE values (that would be done on hardware) and the execution times were almost identical, sometimes resulting in the copy operation being even slower than the calculation of MI/TE. After also considering the overhead of streaming the data to the DFEs, as well as the hardware call we deemed that such a solution would be inefficient.

Unfortunately, the data rates are small (for each market player) which makes a software solution more applicable for such cases of datasets. Especially the Streaming implementation is very efficient with respect to its real-time capabilities while on the other hand it needs a lot of memory in order to keep all the different hashmaps available constantly. The streaming implementation is memory intensive which is the disadvantage of this approach. All the hashmaps have to be kept in memory and are updated in a streaming way. In the case of datasets which result into dense pdfs (>50%) the amount of memory required per market player pair is dramatically increased resulting in the incapability of the streaming software to tackle such test cases.

The advantages of the hardware implementation are that it can calculate the MI/TE of any number of market players as the memory is needed only to store the time series, which should be smaller than storing all the pdfs. Also, whatever the nature of the data it has the same execution time whether the data are sparse or dense. The hashmap pdfs are getting larger as the amount of data increase and the values of the data change significantly per time point. The array pdfs remain the same. The disadvantage is that even if we take into account the overhead of the hardware call of 10ms per market player pair, for 1000 market players the hardware would require almost 3 hours to produce the 1 million MI/TE results.

Finally, we conclude that the hardware implementations of MI/TE are not as efficient as the software Streaming version for the real-time stock market test case. If the data rates (per market player) increase dramatically or in the case of historical datasets (10^9 length time series) the hardware can provide better performance.

3 Kernel Merger - Automatic Dataflow Design Optimizations

In this deliverable we consider designs that use Maxeler Dataflow Engines (DFEs) as hardware accelerators. Such designs are typically developed by porting an existing software implementation into a dataflow implementation using Maxeler's MaxCompiler. After the porting the design goes through an optimization phase to maximize the performance. While it is very challenging to automate the translation of a sequential software implementation into a dataflow implementation (or any other parallel implementation for that matter) we can target a number of typically time-consuming optimizations for tool automation to improve the overall productiveness of the translation process.

The purpose of the Kernel Merger tool is to provide a number of automatic optimizations to a dataflow design that would otherwise take considerable time and expertise by the developer. The main focus is merging different parts of a dataflow graph that are not active at the same time, but a number of further optimizations are also included. As an example, consider an algorithm that computes in hardware either a * b or a + b depending on a condition c. Both, the adder and the multiplier must be instantiated even though only one of the results will be selected by a multiplexer (MUX) at the output. However, if the computation is either a * b or a * d then instead of creating two multipliers we can use a single multiplier with a MUX selecting between inputs b or d. This is a more efficient hardware implementation as a MUX uses far fewer resources than the second multiplier.

In practice, this situation can be found when applications offload several compute intensive parts, often referred to as kernels, to the DFE. Since only one kernel is active at any given time, the other kernels remain idle. However, since they still have to be implemented on the DFE, a large portion of the chip is not in use most of the time. The solution for this problem is to reuse arithmetic units between kernels to minimize the area overhead introduced by simultaneously implementing multiple kernels. Manually identifying the arithmetic units that can be shared and creating the required control logic would be very time consuming.

For this reason, we created a dedicated pre-processing optimization tool called Kernel Merger. It is implemented as a software library for the Maxeler high-level synthesis tool-chain MaxCompiler and it is able to automatically merge multiple calculations and thereby significantly reduces the hardware area requirements without any additional user interaction.

3.1 Principles of dataflow graph merging

Designs targeting Maxeler DFEs are developed in a high-level language which describes a dataflow graph. A dataflow graph is a directed graph where the nodes represent the basic operations and the edges between them the specific paths data elements follow. A dataflow graph can be easily translated into a hardware circuit, since every node directly corresponds to a hardware unit that has to be allocated on the chip surface and every edge represents a wire between two units. Another advantage is that the level of abstraction can be raised within the same graph. One node can either represent the change of a single bit in a signal or a complex algorithm. Even though the amount of hardware created is very different, both nodes can be used in the same dataflow graph without any problems. For this reason we decided to build our tool on top of dataflow graphs. First, each kernel of the application is described as a dataflow graph, then our tool optimizes these graphs and automatically merges them. Finally it produces the required hardware blocks.

We give a simple example of the tool's functionality. The dataflow graphs in Figure 1 represent $(x + y)^2$ and $(x + y) * z$. Assume that only the result of one of the two calculations is used at any given execution cycle. It is obvious that implementing both dataflow graphs uses double the necessary resources. A merged, optimized implementation is shown in Figure 2. This implementation reduces the hardware requirements significantly since only one addition and one multiplication have to be implemented.

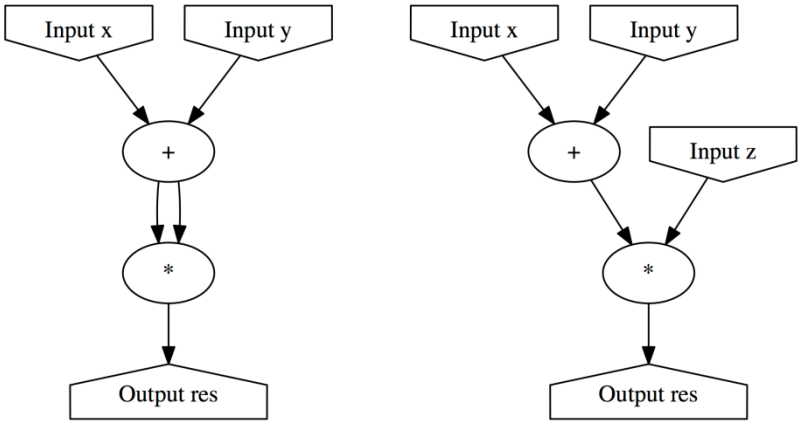

Figure 1: Two input graphs with identical operators that can be merged.

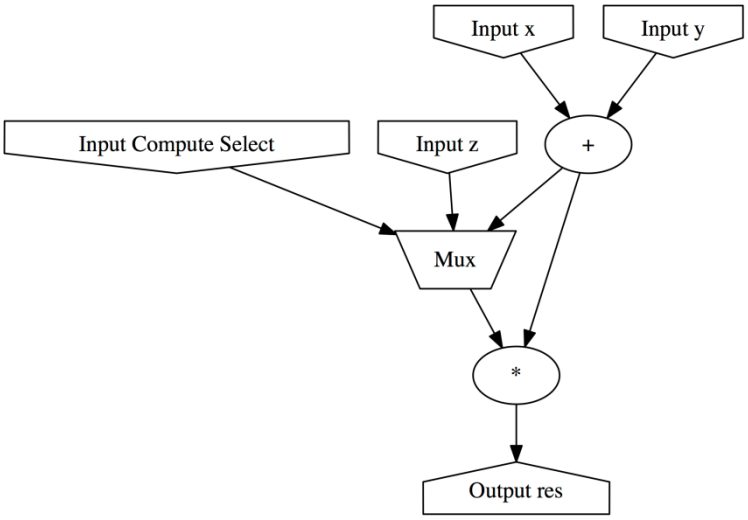

Figure 2: The graphs from Figure 1 merged to save hardware resources.

While the merging seems to be trivial for small graphs of the size presented above, the search for optimal solutions is far more complex when graphs with thousands of nodes are considered.

3.2 MaxCompiler

The KernelMerger tool is an add-on to MaxCompiler, a tool developed by Maxeler Technologies for static dataflow hardware generation, which also relies on an internal dataflow graph representation. MaxCompiler internally supports several graph passes for dataflow graph generation and optimization such as scheduling; however, these target kernels individually. The presented optimizations in the following sections are novel functionalities.

In MaxCompiler, a Java based meta-language called MaxJ is used to describe the dataflow graph generation process and is employed by us to create the tool. The MaxCompiler API also supports creation of backward edges in the dataflow graph to handle loops under hardware constraints and implement more complex control flow structures. To support the above, automatic stalling of the dataflow graph inputs is supported. More complex I/O operations like memory accesses are

handled by dedicated nodes in the generated dataflow graph. MaxCompiler then creates massively pipelined hardware structures from the dataflow graphs.

3.3 Algorithmic optimizations

In addition to graph merging, the tool implements multiple further optimizations as distinct graph passes. This means that they operate on a dataflow graph and transform it in order to save hardware resources. While many passes just optimize regular dataflow graphs, some of them are unique to the problem of reusing nodes between separate graphs.

All these graph passes run in a loop until the graph remains unchanged by an iteration. While this increases compilation time, it boosts the number of optimizations applied. We discovered that starting with a more optimized dataflow graph often leads to improved performance in terms of area reduction by the merging algorithms, thus we implemented many optimizations not specific to the problem of merging dataflow graphs to preprocess the graphs. For this reason after each node merging step all optimization graph passes are repeated.

3.3.1 Graph structure

As we mentioned in the previous section that a dataflow graph is used as the internal representation of the hardware that has to be implemented. In order to write clear and efficient graph passes, a simple graph representation is key. For this reason the graph is implemented as a list of nodes.

All kernels first create their own dataflow graphs, which are then combined into a single graph. While they may not be connected initially, all graphs are stored in the same data structure. The advantage of such approach is that all graphs can be optimized at the same time rather than added successively. This makes the merging of multiple graphs far more efficient, since we can decide which nodes should be merged on a global level. We also store the kernel-to-node mapping information.

3.3.2 Dead Code Elimination

The first graph pass removes all unnecessary nodes from the graph. This graph pass is very simple since it only has to remove nodes where the output is not connected to any other node in the graph.

3.3.3 Constant Folding

The second graph pass is a constant folding graph pass. The target is to eliminate as many unnecessary operations from the dataflow graph as possible. If a node has only constant inputs, the result is also a constant. Replacing the node with a constant saves hardware and simplifies the dataflow graph.

While this optimization is already implemented in MaxCompiler this graph pass it necessary as a preprocessing step. If constant folding is not done, a node that has only constant inputs may be merged with a node that has variable inputs. In this case pruning the node after the merge is impossible and resources might be wasted due to the second node no longer being available for merging.

3.3.4 Associative Operations

The tool uses a graph pass which automatically combines associative operations. If an associative operation has a second associative operation of the same type as input, they can be combined into a single node as shown in Figure 3.

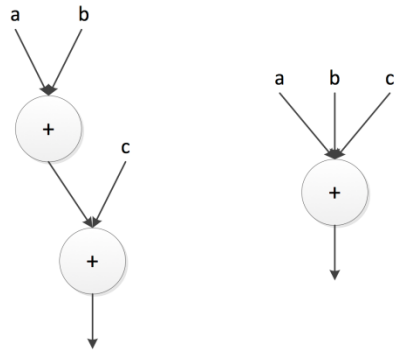

Figure 3: Two associative operations can be combined into a single node

This has multiple advantages. The number of nodes in the graph is reduced, which speeds up all other graph passes and makes it easier to perform their optimizations. Furthermore this technique enables additional associative optimizations.

Another advantage is that the operations can be rearranged into trees very easily. Normally multiple operations are represented as a simple chain of operations. If they are rearranged as a binary tree, as shown in Figure 4, the latency of the entire computation is reduced significantly while the number of operations is preserved.

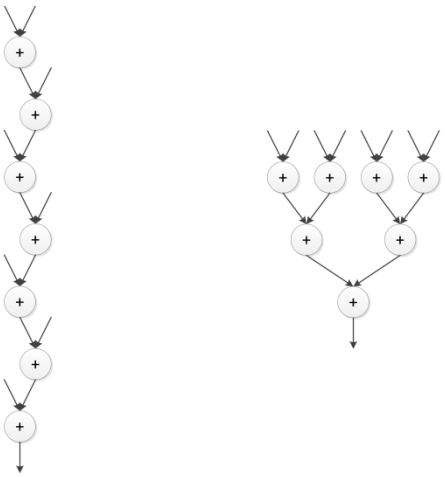

Figure 4: Same number of adders arranged in linear order and as a binary tree.

If the operations are arranged as a chain, the latency of each operation will force the inputs to be scheduled at different times. In a tree the inputs will be scheduled closer together resulting in shorter FIFOs, which in its turn saves some of the additional hardware resources consumed by the scheduling.

There are certain optimizations that can be applied to some types of operations if they are arranged as a tree. For and, or, minimum and maximum operations all duplicate inputs can be deleted since they will not change the result. If an integer exponent power is calculated all inputs to the multiplication are connected to the same node. If such a pattern is detected this operation can be optimized as Figure 5 shows.

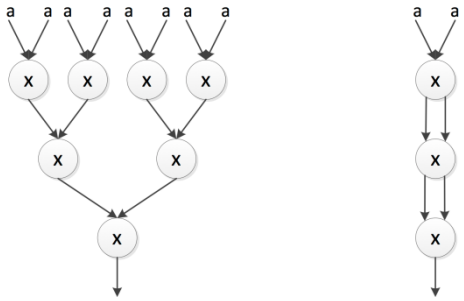

Figure 5: Optimization of the calculation of a^8

This optimization is implemented in a way that this pattern of inputs is also spotted if only some inputs to a multiplication node are connected to the same output. In this case the commutative property is used in order to rearrange the operations within the tree to find and optimize all possible powers.

3.3.5 Common Sub-expression Elimination

The common sub-expression graph pass looks for redundant computation and prunes all duplicated nodes. To determine which computations are redundant the inputs to nodes of the same kind are analyzed. For example, if two additions have the same inputs, they obviously compute the same sum. This means that only one hardware implementation is necessary and the second node can be removed from the graph. All nodes using the result from the second addition can instead use the result from the first addition.

The tool is aware which operations are commutative, so the expressions a + b and b + a will be detected as duplicate operations. Since all associative operations that are directly connected are automatically combined into a single node, this allows less obvious duplicates to be removed. If, for example, the user writes a + b + c + d and c + b + a + d, the tool will recognize this as a duplicate.

Furthermore, similar computations are extracted from associative operations. For the expressions a + b + c + d and a + b + c + e the expression a + b + c is extracted into a separate node which occurs in the graph only once. This result can then be added to d or e respectively. Again, commutativity will be able to detect duplicates.

3.3.6 Other Mathematical Optimizations

Several more complex arithmetic properties are used to optimize the dataflow graph. The distributivity law is used to reduce the node count. The expression a * b + c * b can also be written as (a + c) * b which saves one multiplication. This is applied to all multiplications as well as divisions that have a common divisor.

Since it is very expensive to implement division in hardware, a technique is used when the same divisor appears in multiple equations. Suppose we want to divide a/c and b/c. We can then define d=1/c and calculate a*d and b*d. While in this case the total number of operations is increased, the hardware resource usage will be actually reduced.

3.3.7 Improved Floating Point Addition

Floating point addition requires a tremendous amount of hardware resources. The reason for this is the need to normalize one of the inputs and to renormalize the result. Since the number of consecutive additions is known, the amount of renormalizations can be reduced by only applying them at the inputs and outputs of the consecutive operations instead of at every step in between.

The first step in order to achieve this reduction is to reorder the graph as a tree. Now only the final output of the tree has to be renormalized. Instead of normalizing one input of every node all inputs at the first tree layer have to be normalized. As a result n normalizations instead of n-1 are needed

for a tree of n-1 operations, however, since only one renormalization is needed significant overall area reduction is achieved.

The negation of numbers in floating point is cheap since only the sign bit has to be toggled. This means that all subtractions can be performed as additions where the second input is negated. This enables all optimizations normally only executed on associative and commutative operations for subtractions, which also allows subtractions to use the improved floating point addition and save additional resources.

3.3.8 Constant Multiplication and Division

If the divisor is constant, a division can be transformed into a multiplication with the inverse of the constant. Since division is far more expensive than multiplication, this saves a significant amount of hardware resources.

Furthermore fixed point multiplications by a power of 2 do not have to use any hardware at all, since only a reinterpretation of the bit significance has to be performed. A multiplication by a constant, which, represented in binary, has only a limited number of bits set, can also be optimized into a series of additions to save resources.

For floating point numbers the multiplication with a power of two can be optimized, since only the addition of a number to the exponent is needed while the mantissa stays the same.

3.3.9 Merging

The actual sharing of hardware resources is implemented as a merging of nodes in the dataflow graph. In order to merge two nodes the inputs to the merged node have to be multiplexed so that either the inputs to the first node or the inputs to the second node are selected. The output of the merged node has to be connected to all nodes that use the result of either node. This process is shown in Figure 6.

There are multiple conditions that must be checked before two nodes can be merged. Obviously both nodes need to implement the same operation, but they also have to operate on the same data type. Another condition is that both nodes have to be created in different kernels.

To understand why this condition is necessary the properties of the dataflow graph have to be recalled. The actual data flows along the edges through the nodes in a dataflow graph. If the graph gets transformed into hardware, on every cycle new data has to be read from the sources and sent to the sinks in order to maximize throughput and thereby efficiency. If a node is now used twice in the same kernel it has to calculate two different results on the same cycle, which is not possible.

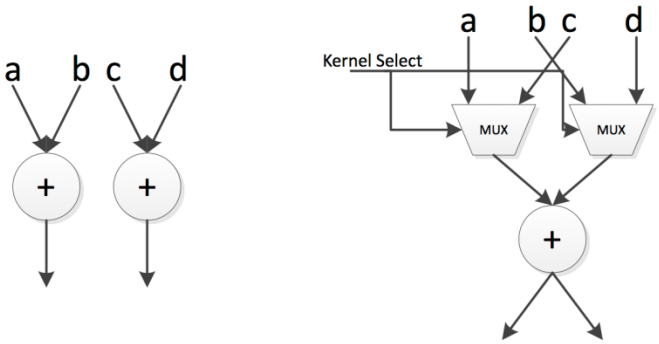

Figure 6: Two add nodes are merged together by multiplexing the inputs.

The last condition is that we do not create loops in the graph, since in many cases this would also mean that nodes have to calculate two different results in the same cycle especially if we want to control which kernel runs on any given cycle.

Since all associative operations are represented by a node with multiple inputs it is very unlikely that a second node exists with the same number of inputs in a different kernel. For this reason we may merge associative operations with a different number of inputs. In order to do this all inputs of the node that cannot be multiplexed with the inputs of the other node are multiplexed with the identity element of the given operation. Although we now perform more operations than necessary for the smaller tree, the total is reduced, since all operations of the larger tree have to be implemented anyway (see **Figure 7**).

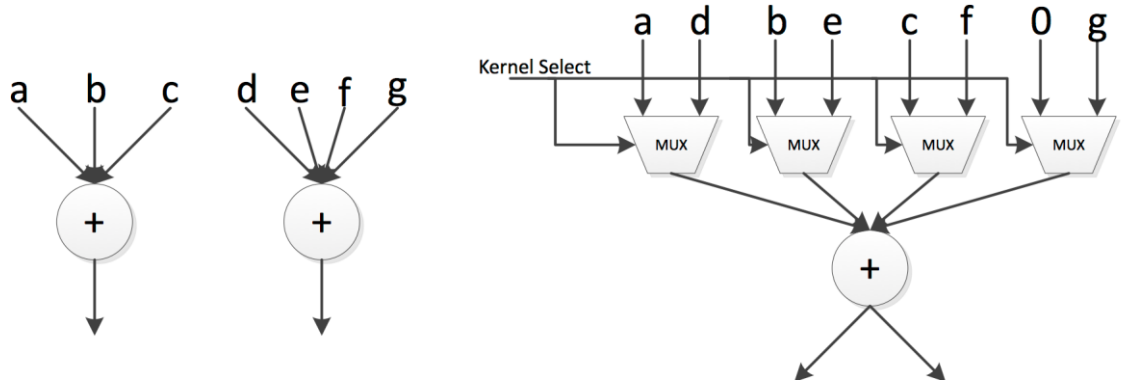

Figure 7: Two trees of different sizes are merged.

It is important to note that not all nodes in the graph are worth merging. For example, most logical operators like and, xor or equals use fewer resources than the MUXs that would be needed to merge them. By default these nodes are not merged at all, but since merging them might enable other optimizations, the user can change how expensive nodes have to be before they are merged.

3.3.10 Heuristics to Select Nodes for Merging

The merging of two nodes is not complicated; it is far more important to decide which nodes have to be merged. One reason for this is that the change of the structure of the graph may allow further optimizations. For example, it might be possible that the common sub-expression elimination can now be applied to nodes whose inputs have been merged.

However, the more important reason is that if two nodes are merged, new dependencies are introduced into the graph, and further merging may be prevented. If, for example, the first node in one kernel is merged with the last node in a second kernel, this would prevent merging of all other nodes in these kernels since this would introduce a loop in the dataflow graph.

Since the targeted problem sizes are usually very high (> 1,000 nodes) it is important to keep the execution time of the tool in mind. For this reason, it was decided to use heuristics in order to determine which nodes should be merged. The most important cost driven heuristic decides in which order nodes are merged. The algorithm always tries to merge the most expensive node first. This greedy approach makes sure that large area reductions are not prevented by merging nodes that only provide a small reduction.

Once the node that has to be merged first has been determined, a list of all possible merge candidates is created. If there are no merges possible the procedure will be repeated for the next most expensive node.

If the list contains multiple candidates another heuristic has to be applied in order to decide which nodes are actually merged. If we merge associative operations with multiple inputs the first criterion is to find a node of the same type where the number of inputs is as similar as possible. The other heuristic to decide which possible match is chosen is based on the position of the nodes in the graph. In order to decide how deep in the graph a node is located we calculate the distance from the inputs and try to find nodes where this distance is as similar as possible.

There is one exception where the previously described heuristics are not applied in the same way. If two nodes can be merged and both share one input, they get merged earlier. The greedy heuristic of merging the most expensive nodes still applies, but the improvement achievable for merging nodes which share one input is multiplied by 1.2. This small bias makes sure that they are merged before other nodes of the same class are merged. This behavior is desired, since the shared input makes sure that both nodes have a quite similar position in the graph and thereby the chance of preventing other merges is decreased. Since when this shared input is a constant the above prediction is less strong, a bias of only 1.1 is used to represent this case. When multiple nodes share the same input and all of them could be merged, the distances to the input are evaluated in order to determine the optimal match.

3.3.11 MUX Collapsing and MUX Input Minimization

If the same node is merged multiple times, which often happens if a large number of kernels are merged, multiple layers of MUXs are created in front of the inputs. Obviously this is suboptimal in terms of area usage and it is better to combine these MUXs into a single MUX. Since they all share the same select signal this process is straightforward and only the inputs have to be connected to the appropriate positions.

If many kernels are merged, the MUXs used to select which kernel is currently executed will have a huge number of inputs. If, for example, fourteen kernels are merged, each MUX must have fourteen inputs to select from. These MUXs are expensive in terms of hardware usage. Since in many cases only a subset of these inputs is used, this hardware usage can be reduced drastically.

For example, for four kernels the select signal is two bits wide and the MUX has four input ports. If only kernels one and three are used the MUX is twice as wide as needed. To avoid this a lookup table is generated which reduces the select signal to one bit so that the MUX only has to have two inputs.

This lookup table can be implemented as another MUX. Of course, this MUX has to have the same number of inputs as the MUX that is about to be replaced, but since the input is only a few bits wide instead of the width of the compute data-type, the area savings achievable are significant. Once such a lookup table is created it can be reused for other MUXs which select between the same kernels.

3.3.12 Timing Optimizations

The throughput of a hardware design always depends on two factors. The first is the number of data items that can be processed in one cycle and the second is the clock frequency at which the design can be run.

Since this tool introduces new abstraction layers it is actually quite hard, if not impossible, to improve the timing characteristics of a design manually. For this reason, a functionality was added to the tool which tries to improve the timing characteristics automatically.

If the same output has to be connected to many other nodes it is often hard to meet timing since it may not be possible to place all the nodes in close proximity which means that the wire length will increase. In order to avoid this problem, it is possible to automatically insert registers. This is done in a tree like fashion where each register has a maximum number of outputs. This is especially needed for the signal that decides which kernel has to be executed currently, since it can be

connected to hundreds of MUXs. For this reason the tool automatically inserts registers after each node that has a large fanout.

4 Tool Evaluation

This section presents the evaluation methodology that we followed for the new tool, i.e. the Kernel Merger tool, which was implemented by Maxeler in the context of the QualiMaster WP3 needs. Each algorithm that is mapped on the reconfigurable chip of a Maxeler server is implemented as a separate kernel IP. One of the more common problems when mapping a kernel IP on reconfigurable technology is the restriction of the available resources. The Maxeler Kernel Merger tool attempts to solve this restriction by merging the mapped kernels. Such solution leads to lower resource utilization due to the implementation of common algorithmic parts on common reconfigurable resources. As a consequence, the low resource utilization can lead to even higher parallelization levels in the DFE chips, thus to even higher performance achievements. As described in the previous sections, the tool attempts to find common computational parts in each mapped kernel. In our case, we moved towards different evaluation test cases, which are described in the following sections. The evaluation of the final mapping is based on the resource utilization of the merged architectures vs. the initial non-merged solutions.

4.1 Algorithm Selection

The tool has been developed by Maxeler Technologies partner in the context of the QM WP3. The Kernel Merger tool is based on the needs and the feedback from the hardware mapping of the selected QM algorithms. The new tool supports a satisfying number of functional operations but not all the methods available through the MaxCompiler are available. Based on the project requirements and the available functional operations, we moved forward to algorithm selection for the final tool evaluation. We moved towards two different directions. First, algorithms with common parts were selected in order to show the advantages of using the tool. Second, the QM algorithms were selected in order to show the advantages of hardware merging in the context of the QualiMaster project.

The initial version of the algorithms that were implemented for the QualiMaster project used functional components that are not available at this first version of the Kernel Merger tool. This reason led us to small changes on the algorithms in order to manage to map them efficiently on the DFEs. After these changes, we managed to merge the implemented algorithms and use different scenarios, which shows the advantages of the Kernel Merger tool to the final QualiMaster project.

More specifically, we mapped different algorithms using the Kernel Merger tool. First, we used the implementation of the color to grayscale conversion algorithm in order to get used to the Kernel Merger tool functionalities. Next, we moved towards the mapping of the QualiMaster algorithms based on various application scenarios. Initially, we mapped multiple kernels of the Hayashi-Yoshida algorithm on a single DFE chip, where each one of them used different time sensitivity. Next, we mapped the Mutual Information (MI) and Transfer Entropy (TE) algorithms. These two algorithms consist of common mathematical functions that the Kernel Merger could merge offering low resource utilization and a higher parallelization level in the available DFEs. Last, our final scenario was the merging of the basic QualiMaster hardware-based algorithms, i.e. Hayashi-Yoshida, Mi and TE into a single DFE. Such solution would increase by far the capabilities of the QM hardware platform. In more details, as each Maxeler server consists of 4 DFEs, thus such scenario would lead to 12 available kernels (1 kernel of each algorithm on each DFE) on a single Maxeler platform.

As described above, different mapping scenarios were implemented in order to show the advantages of the Kernel Merger tool to final QM reconfigurable implementations. Considering that the Technical Annex indicated that we would test two algorithms on the new tool, we believe that the actual testing of Kernel Merger by far exceeds the requirements, and it showed some capabilities which are very useful to the designer and can be exploited by Maxeler beyond the QualiMaster project’s scope.

4.2 Evaluation Cases

This section describes the different evaluation cases that were used for evaluating the Kernel Merger tool. First, different colored to grayscale conversion algorithms were mapped into a single DFE chip. This scenario was used mainly for becoming familiar with the Kernel Merger tool. Next, we evaluated the tool on the QM algorithms. We used different combinations of the algorithms with interesting results that are presented in the next sections. Last, this section presents the evaluation results of the Kernel Merger tool, when it is used for mapping on DFE chips the various scenarios.

4.2.1 Evaluation case 1: Colored Image to Grayscale Conversion

4.2.1.1 Implementation

As an initial attempt, we implemented two kernels for color image to grayscale conversion. This test implementation was done in order to get familiar with the Kernel Merger tool. The two kernels consist of the simple average grayscale conversion and the weighted average conversion. The functions that represent the grayscale conversion are:

Average Grayscale: (R + G + B)/3

Weighted Grayscale: 0.2126 *R + 0.7152*G + 0.0722*B

The original kernel graphs for the grayscale conversion are presented below. From these graphs, we can see in Figure 10 that the merger connects the two independent kernels.

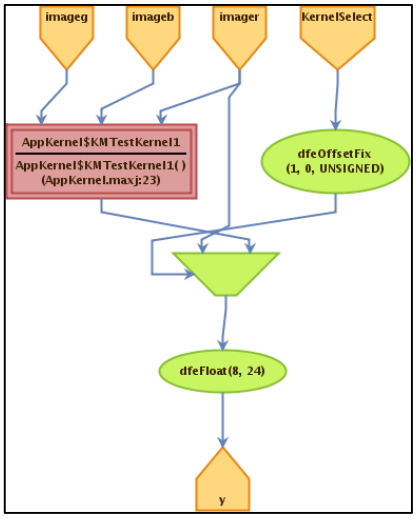

Figure 8: Original Grayscale Average kernel

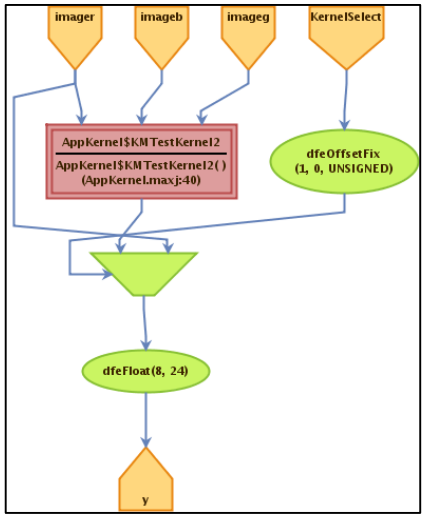

Figure 9: Original Grayscale Weighted average kernel

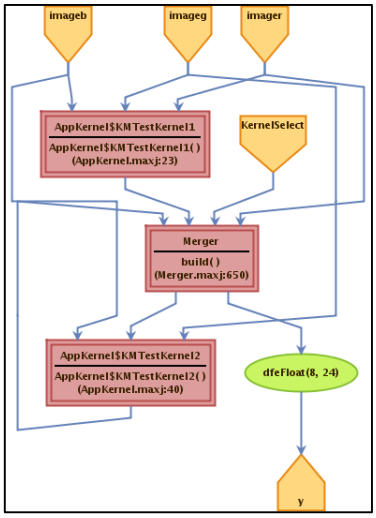

Figure 10: Original Grayscale Merged kernels Graph

4.2.1.2 Results

Next, we move towards the evaluation of the Kernel Merger tool based on the above implementation. The resource utilization results provide a significant conclusion for the usability and effectiveness of the Kernel Merger tool. According to Table 1, if we add the resources of both grayscale converters the LUTs would take 6.48% of the LUT resources while the merger gives the same functionality with only 3.65%. The difference is that in the first case both conversions can be called at the same time, while in the merged case only one after the other. As a conclusion, the tool can effectively merge two kernels by reusing the resources, allowing for the implementation of complex algorithms within the same bitstream, even if on the first parse they do not seem to fit to the FPGA resources.

Table 1: Resource Utilization for Greyscale Conversion

Resources	Average grayscale	Weighted grayscale	2 grayscale converters combined	Merged kernels	Resource Utilization Reduction
Logic Utilization	9542 (3.21%)	9724 (3.27%)	19266 (6.48%)	10859 (3.65%)	44 %
BRAMs	38 (1.79%)	38(1.79%)	76 (3.58%)	38(1.79%)	50%
DSPs	2 (0.10%)	6 (0.30%)	8 (0.40%)	6 (0.30%)	25%

The final kernel graphs for the grayscale conversion are presented in Figure 11 and 12. Figure 13 presents the merged graph for both the average and the weighted average kernels. The complexity of the graph shows that merging two kernels, even as simple as the grayscale ones, is not a trivial process. The size of the produced graphs make them really hard to be presented and analyzed. On the other hand, the graphs are automatically produced by the Kernel Merger tool and they only are presented in this deliverable in order to show the complexity

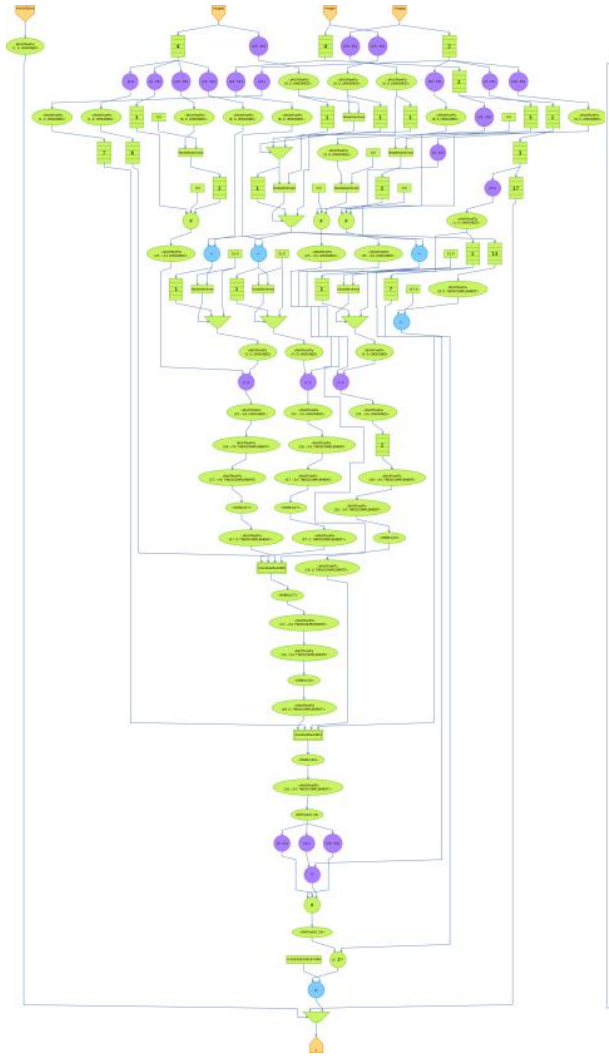

Figure 11: Graph for the Grayscale average kernel

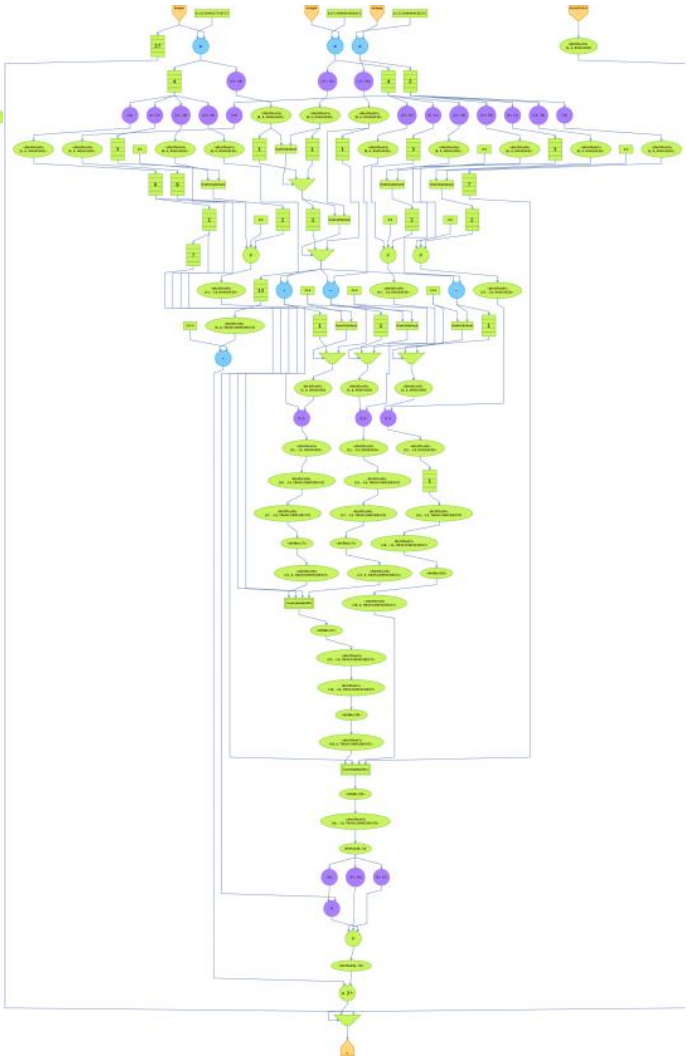

Figure 12: Graph for the Grayscale Weighted average kernel

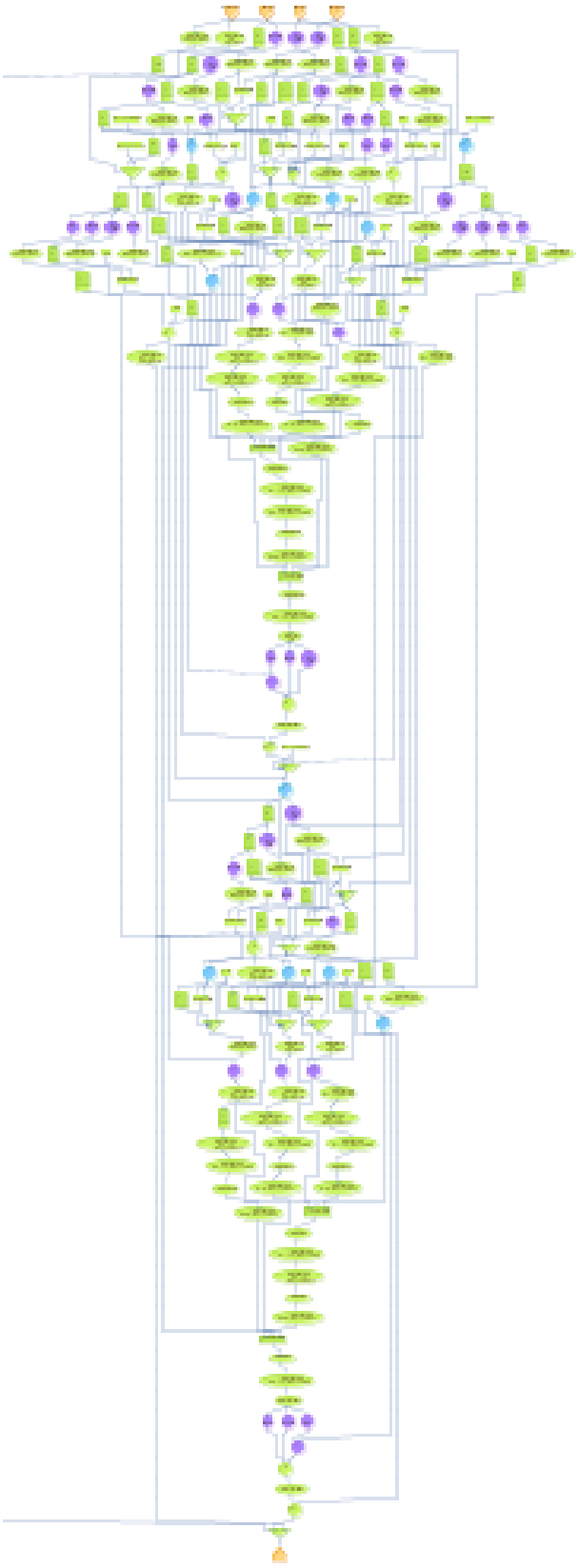

Figure 13: Final graph of the merged Greyscale kernels

4.2.2 Evaluation case 2: Multiple Correlation computations

4.2.2.1 Implementation

This section describes the mapping of the Hayashi-Yoshida algorithm using to the new Maxeler Kernel Merger tool. As presented in previous Deliverables, we mapped the HY estimator on reconfigurable logic using an additive method over a specific time window. The sensitivity of the HY estimator is based on the size of this time window, whereas the functionality of the algorithm does not change. In the QM pipelines, we used the HY estimator to calculate the correlation between the processed market players based on their transactions. As the transactions of the market players do not follow the same frequency, thus, the correlation over different time window sizes would lead to a much better financial analysis.

Based on the above observation, we implemented an evaluation scenario in order to show the advantages of the Kernel Merger tool, which was implemented in the context of the QM project. As described above, the analysis over various time window sizes offer a better financial analysis. Such scenario needs the mapping of multiple reconfigurable HY module on a DFE chip. Each HY module will calculate independently the correlation of the processed market players over a different specific time window size. On the other hand, as analysed in previous Deliverables and it is shown in the Results section the mapping of a single HY module on a DFE uses high percentage of the available resources. Thus, we used the Kernel Merger tool, which merges the mapped architectures and reduces the resource utilization, **a capability which we could not have without the Kernel Merger tool.**

A new version of the Hayashi-Yoshida algorithm was implemented in order to be compatible with the requirements of the evaluation scenario. In more detail, we moved towards three different changes. First, we removed the time window size from the input arguments. The window size for this new version is hardcoded in order to be independent for the different multiple mapped HY modules. Second, we changed the shift right and left operations with the corresponding multiplications and division processes in order to increase the common mathematical patterns of the mapped algorithms. Last, we changed the cast functionality with bitwise operations, i.e. bitwise and/or, for compatibility reasons.

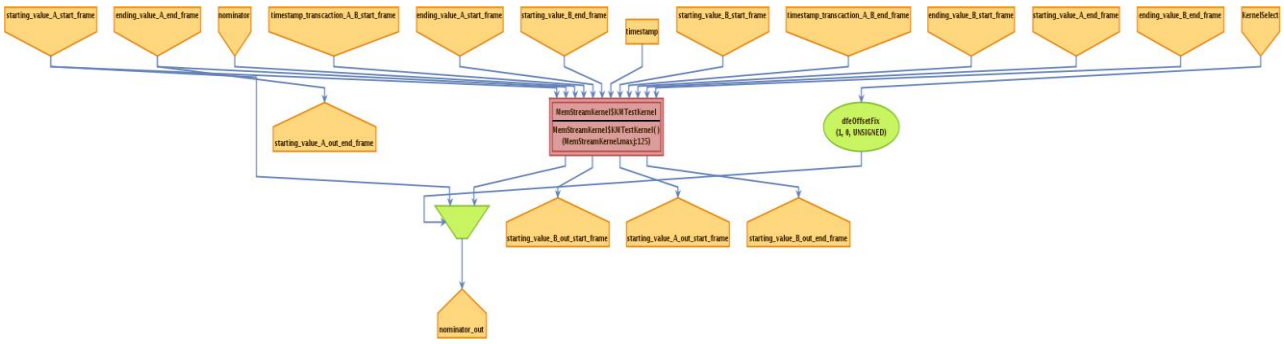

Figure 14: Architecture of the Hayashi-Yoshida module

The output results of the new version of the Hayashi-Yoshida reconfigurable module were validated against the output results of the initial Hayashi-Yoshida module and the official software solution. Figure 14 shows the architecture of the new Hayashi-Yoshida module when mapped on DFE.

The implemented scenario maps 4 of the new version Hayashi-Yoshida modules on a single DFE chip. Each HY module calculates the correlation of streaming market players values based over different time windows, i.e. 1 minute, 15 minutes, 30 minutes and 1 hour. The mapping took place using the new Kernel Merger tool. Figure 15 presents the architecture of the final reconfigurable system for our proposed scenario. The final architecture consists of 4 HY modules mapped and a single Merger component that is used to combine the above components.

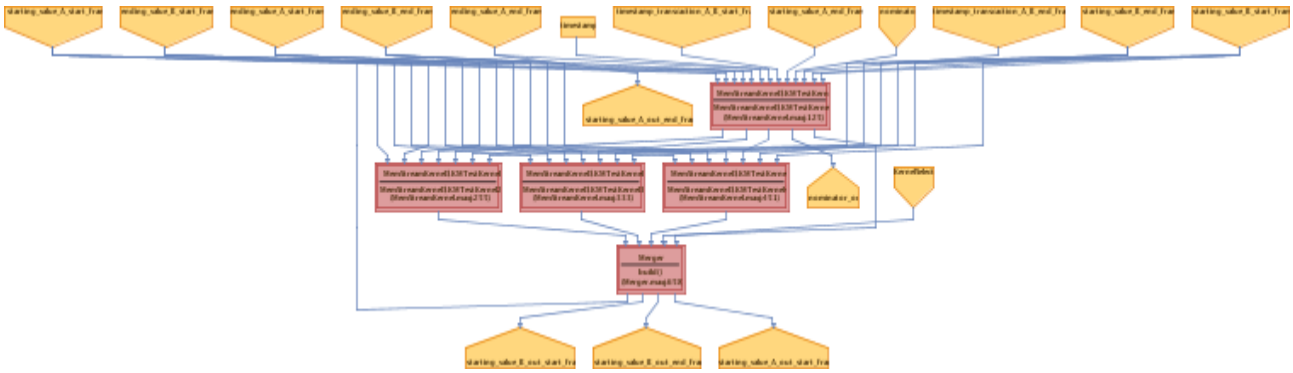

Figure 15: Architecture of the parallel mapping of Hayashi-Yoshida modules using the Kernel Merger tool. The rectangular shapes are the merged kernels and the Kernel Merger component that combines all the above.

4.2.2.2 Results

This section presents the results when multiple new HY modules are mapped on a single DFE of a Maxeler server. First, we analyzed the resource utilization when a single HY module is mapped and we concluded that the mapping of a single HY module on an FPGA chip uses high percentage of the available resources. In more detail, Table 2 presents that a single HY module uses up to 16% of the available slices and the 31% of the available BRAMs (on-chip static memories) of a single DFE chip. In addition, Figure 16 presents the graph that is produced by the tool when mapping a single HY module. According to the utilization results, we can conclude that the maximum number of mapped independent HY modules on a single DFE can be up to 3.

Table 2: Resource utilization when a single, 4 independent HY modules and 4merged HY modules are mapped on a DFE of Maxeler Vectis server

Resources	1 HY module	4 HY Modules (Projection)	4 HY modules (Merged with Kernel Merger tool)	Resource Utilization Reduction
Logic Utilization	79104 (26.58%)	316416 (**106.32%**)	80055 (26.90%)	74.7%
BRAMs	658 (30.92%)	2632 (**123.68%**)	658 (30.92%)	75.0%
DSPs	14 (0.69%)	56 (2.76 %)	14 (0.69%)	75.0%

The implemented scenario for this test case focuses on mapping four different HY modules on a single DFE chip. We projected the resource utilization of the single HY module and we calculated the resource utilization when 4 HY modules are mapped on the same DFE chip. As the numbers in Table 2 indicate, such implementation **would not be feasible without the use of the Kernel Merger tool**, as the resource utilization is over 100%. Table 2 presents the resource utilization when the 4 parallel HY modules are mapped on a single DFE using the Kernel Merger tool. The mapping of 4 HY modules using the Kernel Merger tool uses up to 17% of the available logic and up to 31% of the available BRAMs. According to the resource utilization results, the Kernel Merger tool manages to group the four different HY modules into a single datapath using a few more resources than the single HY module when mapped on reconfigurable logic. The utilization results are impressive, as according to Table 2 no more than 3 HY modules could be mapped on a single DFE. Whereas, the use of the Kernel merger tool seems to group the HY modules mapping offering much better resource utilization, i.e. up to 75%. Lastly, Figure 17 presents the graph that is produced by the Kernel Merger tool when multiple, i.e. 4, HY modules are mapped on reconfigurable logic.

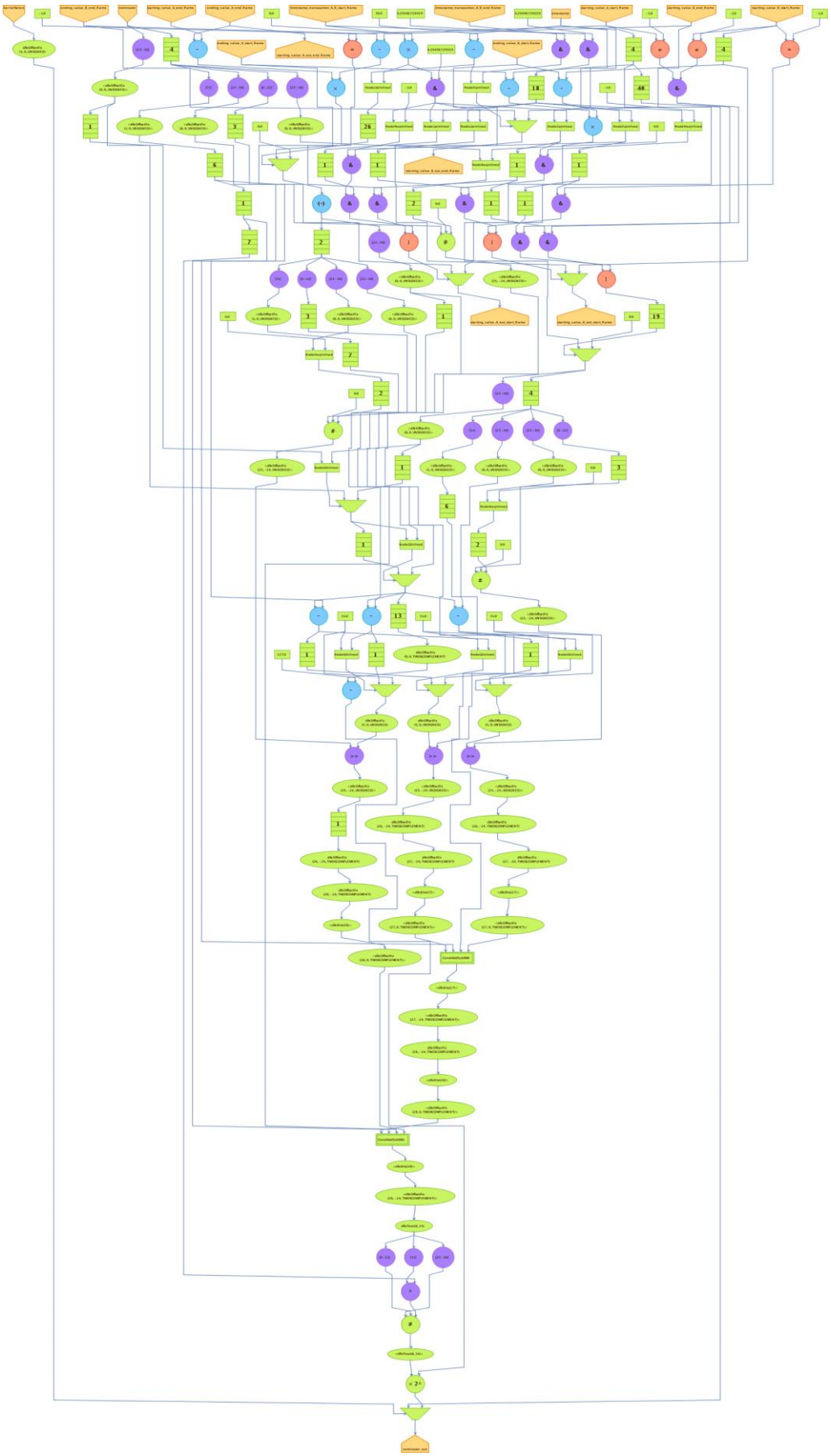

Figure 16: Final graph-based architecture of the mapping of a single Hayashi-Yoshida module

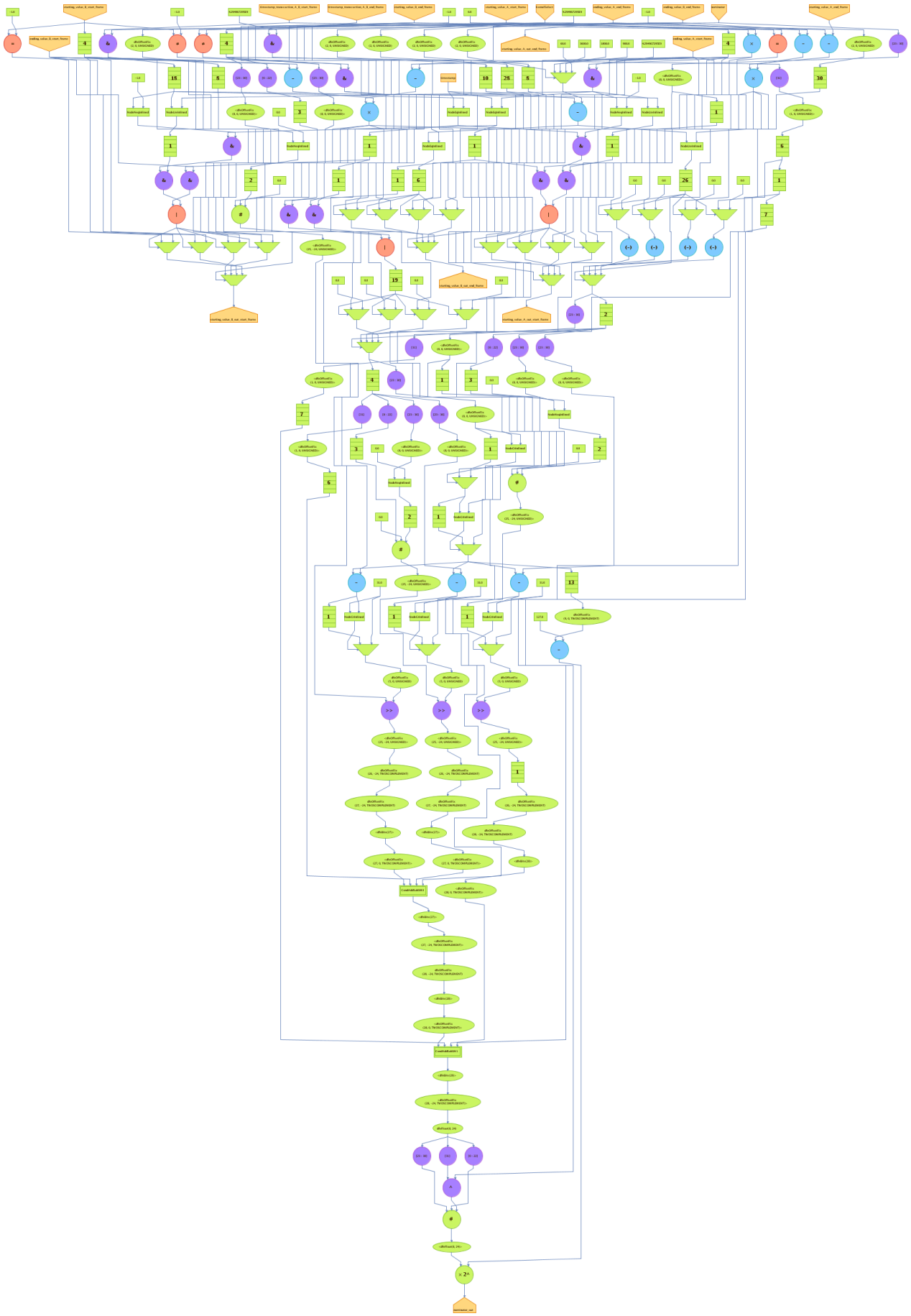

Figure 17: Final graph-based architecture of the mapping of four Hayashi-Yoshida modules

4.2.3 Evaluation case 3: Mutual Information and Transfer Entropy

4.2.3.1 Implementation

This section presents the merging of the Mutual Information and Transfer Entropy Kernels. As shown by their mathematical presentation the operations required are very similar:

MI: $I(X;Y) = \sum_{x,y} p(x,y) log \frac{p(x,y)}{p(x)(y)}$

TE: $T_{X \to Y} = \sum_{y_{n+1}, y_n, x_n} p(y_{n+1}, y_n, x_n) log \frac{p(y_{n+1}, y_n, x_n) p(y_n)}{p(y_{n+1}, y_n)(y_n, x_n)}$

As a result, we expected great resource merging by the Kernel Merger between these two algorithms. The main issue that we had to face during merging the above algorithms was the incompatibility of the Kernel Merger tool with the loopback that both algorithms need for their computation. Thus, we changed the mapped algorithms to a simpler version, which support the simple dataflow processing scheme. Consequently, the results present two slightly changed versions of the MI and TE algorithms with similar resource utilization as the original kernels.

4.2.3.2 Results

As shown in the Table 3, the resources that are utilized by the merger are the maximum resources from MI or TE per resource category. A few more LUTs are utilized by the multiplexers needed in order to map the differences between the two algorithms. Mapping both the algorithms in a single kernel would need double the resources, while the merger manages to map them on the same FPGA resources. Merging such algorithms, with so similar processing, proves to be very effective. Also, by calling one of the TE/MI the bitstream will be downloaded to the DFEs (100ms-1s overhead) and when the next algorithm is called the download overhead is reduced significantly to the 5ms-10ms of the hardware call, as the bitstream is already on the DFEs. The effective resource utilization reduction is at about 50% on the LUTs case. Also by merging multiple cores of these algorithms the resource reduction is a lot more significant as proved by the previous test case where the resource reduction reaches up to 75%.

Table 3: Resource Utilization comparison for MI and TE

Resources	MI	TE	MI & TE (projection)	Merged MI & TE	Resource Utilization Reduction
Logic Utilization	24745 (8.28%)	24250 (8.15%)	48995 (16.43%)	24793(8.33%)	49%
DSPs	30 (1.49%)	26 (1.29%)	56 (2.78%)	30 (1.49%)	46%
BRAMs	62 (2.91%)	45 (2.11%)	107 (5.02%)	67 (3.15%)	37%

The final kernel graphs MI and TE and the merged MI & TE kernels are presented below. The merged kernel graph contains both MI and TE kernels, Figure 20, and thus is more complex than the separate kernel graphs.

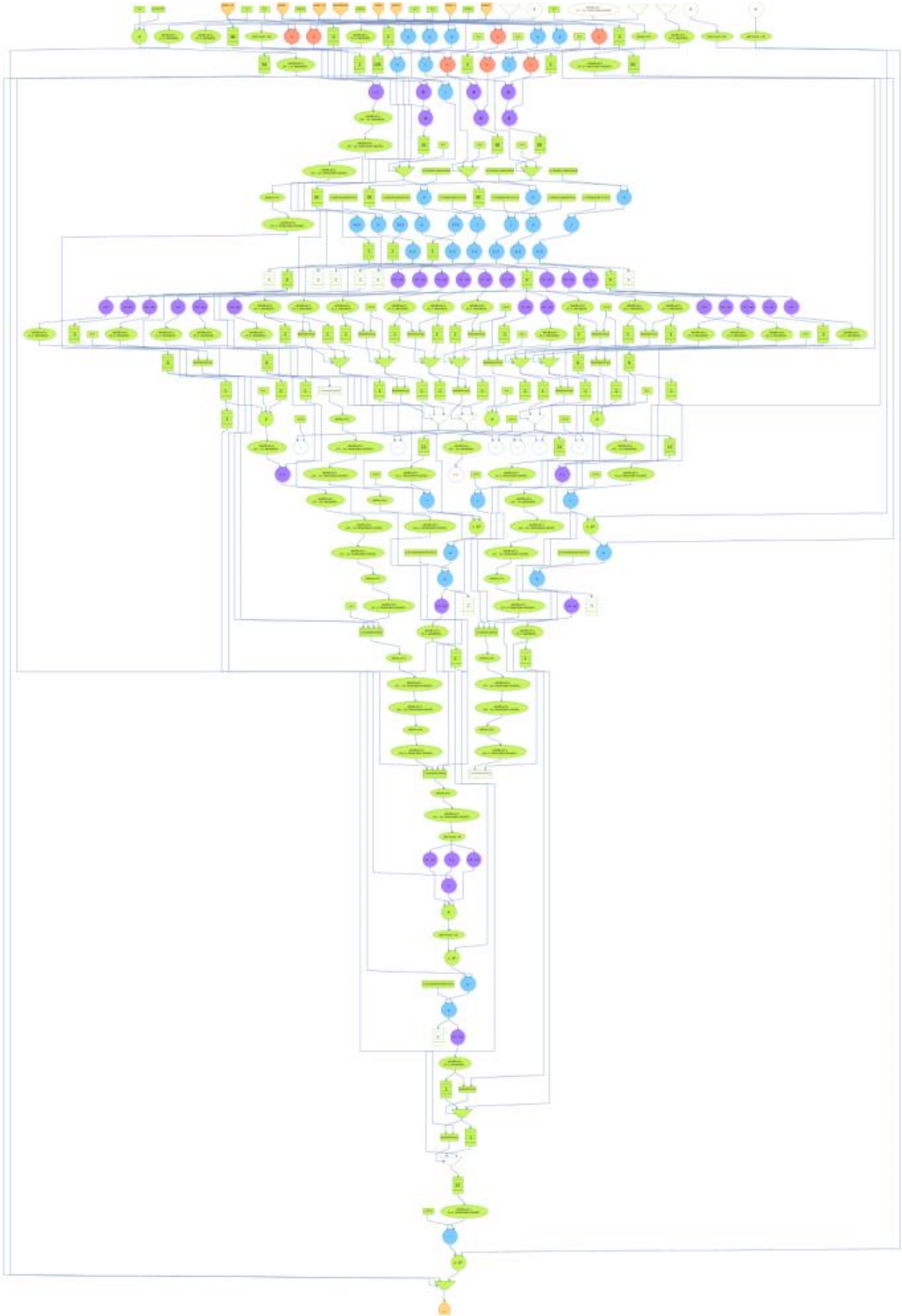

Figure 18: Final kernel graph for MI

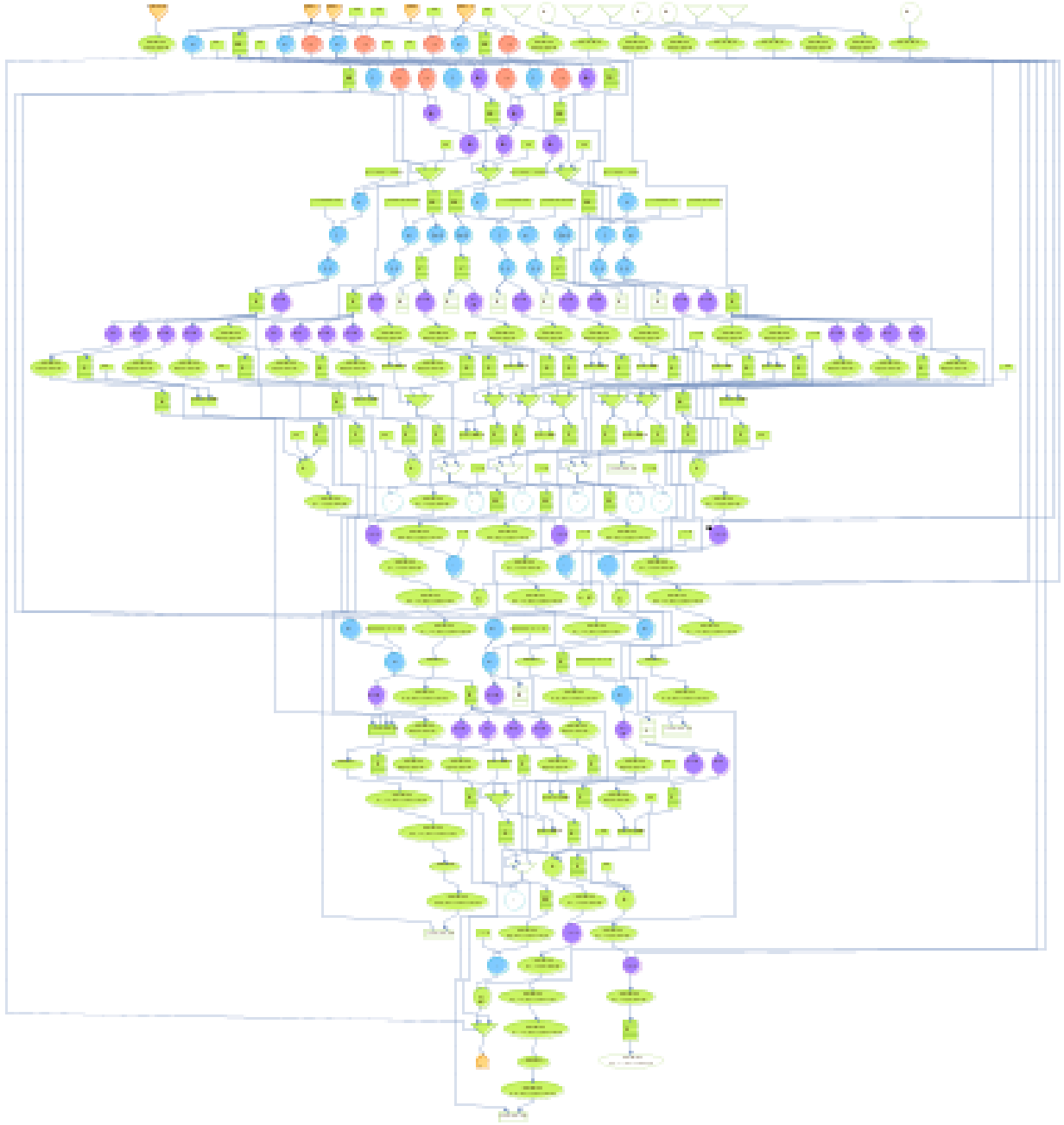

Figure 19: Final kernel graph for TE

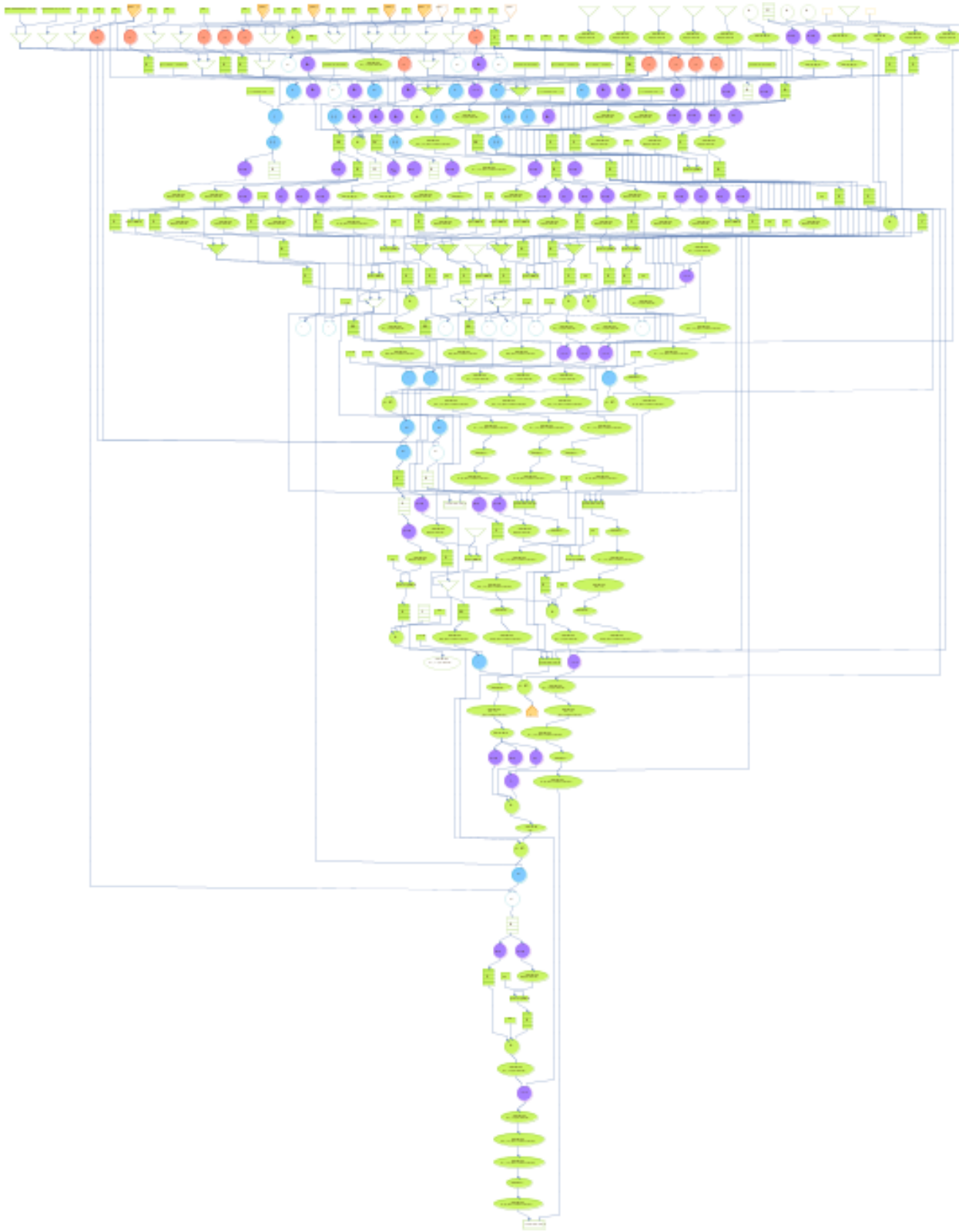

Figure 20: Final merged kernel graph for MI &TE

4.2.4 Evaluation case 4: QM Financial Algorithms

4.2.4.1 Implementation

The fourth scenario was based mainly on the algorithms that are implemented in the context of the QM project. The previous three scenarios mapped very similar algorithms, thus, we showed that the Kernel Merger tool can offer impressive results. This scenario evaluates the Kernel Merger tool in a more difficult test case. In this scenario, we used the Kernel Merger tool for mapping the three main reconfigurable algorithms, i.e. Hayashi-Yoshida, Mutual Information and Transfer Entropy, which were implemented in the context of QM project. These architectures have many architectural differences among them, thus, it will be difficult for the Kernel Merger tool to find common patterns and merge them.

The changes that took place in order to map all these three algorithms into a single DFE chip were mainly based on the I/O issues. In more details, we had to join the different I/O ports of mapped algorithms under a common frame so that all the mapped algorithms to have the same I/O ports. In addition, all the compatibility changes that took place in the previous scenarios were used in this test case, too. The final architecture consists of 1 HY module, 1 TE module and 1 MI module which are combined by a Merger component.

4.2.4.2 Results

This section presents the resource utilization results when all the implemented hardware-based algorithms in the context of the QM project are mapped on a single DFE chip. The resource utilization for each one of the implemented algorithms are presented in the previous sections and they are summarized in Table 4. The resource utilization results show the different nature of each one of the mapped algorithms, i.e. especially between the HY algorithm and the MI/TE algorithms.

Table 4: Resource utilization for each one of the mapped algorithms, i.e. HY, MI and TE module

Resources	HY module	MI module	TE Module
Logic Utilization	79104 (26.58%)	24745 (8.28%)	24250 (8.15%)
BRAMs	658 (30.92%)	30 (1.49%)	26 (1.29%)
DSPs	14 (0.69%)	62 (2.91%)	45 (2.11%)

In order to show the advantages of the Kernel Merger tool use, we projected-added the resource utilization of the different implemented modules if they were independently mapped on a DFE. According to the above scenario, an “independent” mapping of the implemented modules would lead to an architecture, which would use 43.01% of the available logic, 33.7% of the available BRAMs and 5.71% of the available DSPs, as presented in Table 5. On the other hand, Table 5 shows that the merged design using the Kernel Merger tool uses up to 30.82% of the available logic, 31.95% of the available BRAMs and 1.98% of the available DSPs. According to the presented results, the use of Kernel Merger tool can offer a better resource utilization up to 28% for logic resources and 65.32% for DSPs! The reduction achieved is impressive, if we take into account that we merged such dissimilar algorithms. Another important remark is that the final merged design uses 40 DSPs, which are less than the number of DSPs that uses the MI module alone, i.e. 45 DSPs. According to the above, this means that the Kernel Merger tool attempts to merge parts of the same kernel, too. Lastly, Figure 21 presents only one part of the four different partitioned graphs that were produced by the tool, when we mapped this test case scenario.

Table 5: Resource utilization when combining the HY, MI and TE module

Resources	HY & MI & TE module (Projection)	HY & MI & TE module (Merged with Kernel Merger tool)	Resource Utilization Reduction
Logic Utilization	128099 (43.01%)	91714 (30.82%)	28.3%
BRAMs	714 (33.7%)	680 (31.95%)	5.2%
DSPs	121 (5.71%)	40 (1.98%)	65.32%

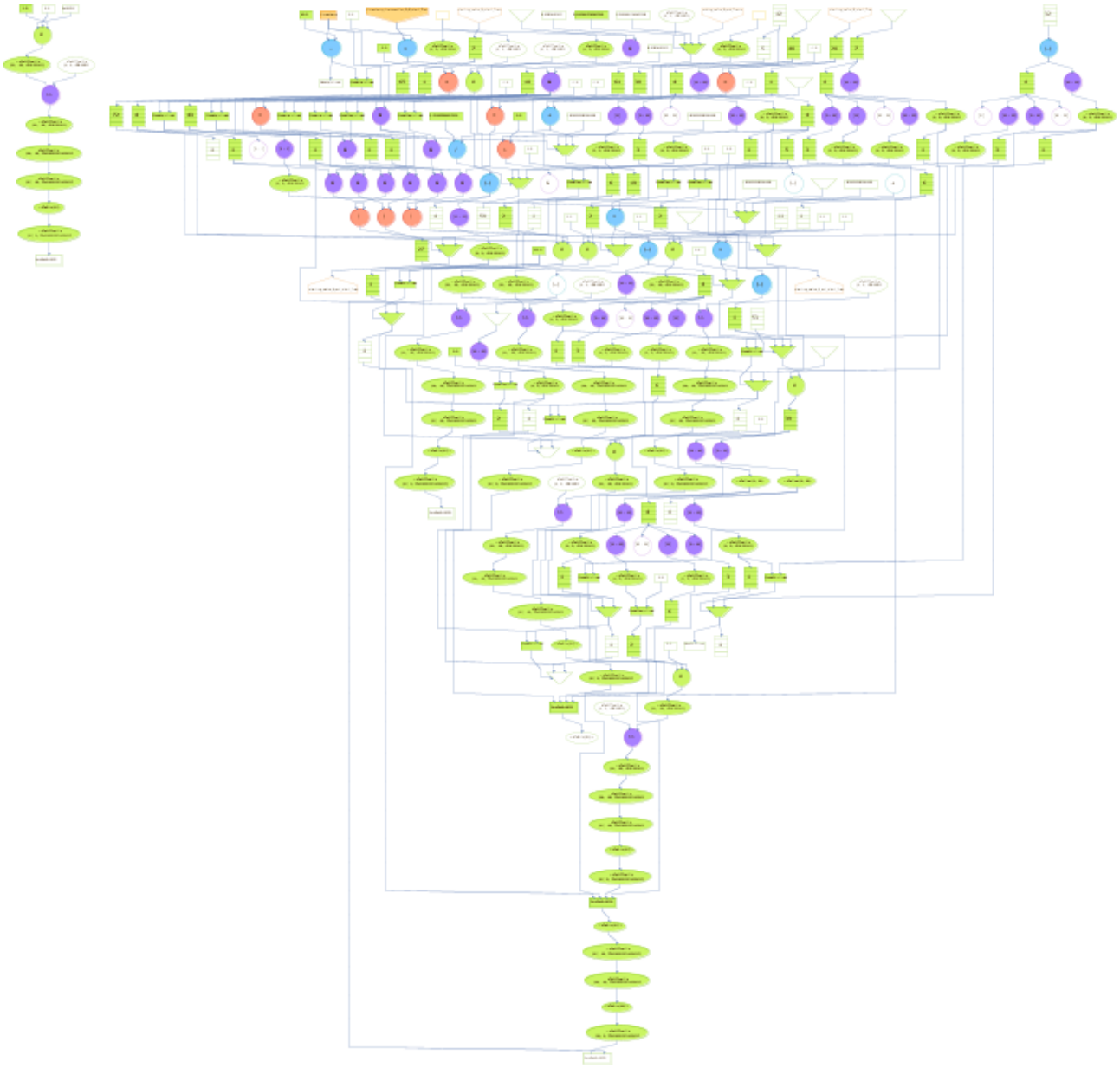

Figure 21: Subgraph of the final merged kernel graph for HY, MI and TE modules. It is important to define that the rectangular white boxes are embedded partitioned subgraphs of the final graph, too.

5 Conclusions

5.1 Guidelines for selecting appropriate classes of algorithms

The KernelMerger tool is a newly developed tool, which especially in the QualiMaster adaptive pipeline project proves that it can be highly effective. The question to be answered is: What is the kind of algorithms that would best take advantage of the tool? This question has many answers, not only in terms of the characteristics themselves, but also on the type of applications which to date cannot be executed on any platform of any vendor, and which this tool will enable. To illustrate, let's assume that we have 2-3 algorithms, which –depending on the application characteristics- are adaptively switched in and out of the custom hardware. To date there are only two ways in which this can be achieved:

- by reconfiguring the device in between algorithms, and
- by having some version of all of the algorithms inside the reconfigurable fabric, so that selection of one of them is merely a matter of routing the data to the appropriate module.

Both of the above approaches do have applications to date. We will give examples outside the realm of QualiMaster in order to emphasize the general usefulness of the KernelMerger tool. If, for instance, we have a network processor which checks for denial-of-service attacks, a periodic update of the patterns might be done daily or weekly, and in this case reconfiguring the custom hardware might have an acceptable overhead, as some microseconds (or even milliseconds) for reconfiguration is an acceptable tradeoff. For other applications, however, e.g. weak vs. strong encryption for the real-time processing of data might necessitate for all encryption algorithms to be on the reconfigurable fabric at all times, as the switch among them might be needed right at the time of en emergency situation, and thus there is no time for the reconfiguration. This latter case means that vast hardware resources remain idle at all times, and the cost of hardware is for much larger devices vs. what is actually needed at any given time. The KernelMerger tool solves this problem by reusing resources among different applications, as long as not all of them are required at the same time. Case-in-point for the QualiMaster project, was that as described in this deliverable, one could have either multiple algorithms at the same time with fewer resources than the sum total of the resources for each algorithm, or, have differently dimensioned versions of the same algorithm all residing on the hardware, so that the correct version will be available at all times without need for excessive resources.

5.2 Best-practices and approaches on the translation of data stream processing algorithms to hardware

The best practice for usage of the KernelMerger tool will evolve in time, as the tools capabilities are expanded, however, several general principles do apply: in general-purpose computing as well as in custom computing the vast amount of resources (quantitatively) are spent on the so-called datapath, i.e. the components that perform computation. Such components in a reconfigurable fabric may be the built-in DSP (digital signal processing) modules, which in turn are used for such computational modules as floating point arithmetic units. If Application A and Application B use floating point arithmetic but they do not both run at the same time, the KernelMerger tool will share this unit among the two applications, thus saving resources that would otherwise need to be duplicated.

The user can help the tool discover such reusable resources by having clean codes with well-defined interfaces among modules, as the KernelMerger tool analyzes graphs in order to achieve its goals. In addition, the usage of standardized libraries is essential in this process. If a module that could potentially be shared is a shared library component, the tool knows how to reuse it, however, if in Application A this component is called from a library and in Application B a functionally equivalent module is hand-coded, the KernelMerger tool cannot determine that it is the same, reusable module and hence it may not maximize reusability among modules.

5.3 Expert evaluations in WP6

This section presents the conclusions that can be summed up by the use of the Maxeler Kernel Merger tool. As the presented evaluation tests indicate, the Kernel Merger tool can offer really impressive results as far as the hardware resource utilization when multiple kernels are mapped on the same DFE chip. Especially, if the mapped kernels have high similarity, then the tool can reduce the resource utilization up to the resources used by the "biggest" mapped kernel. Last, it is important to mention that the kernel merger tool can merge resources internally into the same kernel module. In our evaluation cases, we used the Kernel Merger tool for various and we reached to some conclusions that are presented below:

- The tool can offer high advantages as far as the resource utilization especially in cases where the reconfigurable mapped designs use high percentage of the available resources.
- In addition, the tool can increase the parallelization level of the mapped algorithms increasing the final performance that the hardware implementation can offer.
- The more architectural similarities exist among the mapped kernel, the better resource merging takes place by the tool.
- The tool attempts to merge not only different kernels but common patterns that exist internally to each kernel.
- The tool automates the process of building complex reconfigurable architectures by combining complex kernels. The building reconfigurable architectures using the tool is very fast and productive.
- The merger also provides one more utility. The download of a bitstream to an FPGA increases the overhead of a hardware call. In more details, the first time a bitstream is downloaded, the whole process takes from 100ms to 1s. The Kernel merger tool can merge all the mapped methods in the same bitstream and thus changing from one to the other design there is no need for downloading a new bitstream.
- It allows the mapping of algorithms that would not fit on a single FPGA, e.g. by merging the resources of one step of the algorithm with the next. The merged steps of the algorithm have to run sequentially but there is no need of a new bitstream for each step, while the download overhead is eliminated.
- The installation of the tool needs some experience but on the other hand it is easy to use.
- Our test case scenarios show that this tool can be used to a wide range of applications with impressive results.
- As this tool uses a new framework, more libraries and methods need to be supported in order to increase its functionality.

Concluding, the Kernel Merger tool is a promising tool that can give really impressive results. The evolution of the tool could lead to an important product for the Maxeler company.

Concluding on WP3 we can state that all of its goals were achieved, and the emergence of technologies with similar approaches (such as Microsoft Bing and Amazon's "Xilinx FPGA Cloud"), which nonetheless remain behind the achievements of QualiMaster in terms of adaptivity, demonstrate that the state-of-the-art is not anymore in the execution of computational kernels in hardware, but in very tight integration of hardware and software recourses, in a distributed environment, and with adaptivity bot in terms of hardware (through reconfiguration) and system-level (through adaptive pipelines). This means that quite possibly one of the side effects for further exploitation (beyond what has been reported already) of the QualiMaster project might be in run-time environments for reconfigurable/FPGA cloud services.

References

[1] Shao, S., Guo, C., Luk, W., & Weston, S. (2014, December). Accelerating transfer entropy computation. In *Field-Programmable Technology (FPT), 2014 International Conference on* (pp. 60-67), IEEE.

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

[3] Dionisio, A., Menezes, R., & Mendes, D. A. (2004). Mutual information: a measure of dependency for nonlinear time series. Physica A: Statistical Mechanics and its Applications, 344(1), 326-329.

[4] Marschinski, R., & Kantz, H. (2002). Analysing the information flow between financial time series. The European Physical Journal B-Condensed Matter and Complex Systems, 30(2), 275-281.

[5] Dimpfl, T., & Peter, F. J. (2013). Using transfer entropy to measure information flows between financial markets. Studies in Nonlinear Dynamics and Econometrics, 17(1), 85-102.

[6] Shao, S., Guo, C., Luk, W., & Weston, S. (2014, December). Accelerating transfer entropy computation. In IEEE Field-Programmable Technology (FPT), 2014 International Conference on (pp. 60-67).

[7] https://cran.r-project.org/web/packages/entropy/index.html

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

[9] Cormode, G., & Muthukrishnan, S. (2004, April). An improved data stream summary: The count-min sketch and its applications. In Latin American Symposium on Theoretical Informatics (pp. 29-38). Springer Berlin Heidelberg.

[10] Datar, M., Gionis, A., Indyk, P., & Motwani, R. (2002). Maintaining stream statistics over sliding windows. SIAM journal on computing, 31(6), 1794-1813.