TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Computer Systems

Natalia Cherezova 194204IASM

# HLS-based Optimization of Tau Triggering Algorithms for LHC@CERN Application

Master's Thesis

Supervisor: Artur Jutman
PhD
Co-Supervisor: Dmitri Mihhailov
PhD

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutisüsteemide instituut

Natalia Cherezova 194204IASM

# Kõrgtasemesünteesil põhinev tau-vallandumise algoritmide optimeerimine CERNi Suurele Hadronite Põrgutile

Magistritöö

Juhendaja: Artur Jutman
PhD
Kaasjuhendaja: Dmitri Mihhailov
PhD

Tallinn 2021

# Declaration of Originality

Declaration: I hereby declare that this thesis, my original investigation and achievement, submitted for the Master's degree at Tallinn University of Technology, has not been submitted for any degree or examination.

Author: Natalia Cherezova

May 10, 2021

# Abstract

With the current increase of the data produced by the Large Hadron Collider (LHC) at CERN, it becomes important to process this data in a corresponding manner. To begin with, to efficiently select events that contain relevant information from a massive flow of data. This is the task of the tau lepton decay triggering algorithm developed by the National Institute of Chemical Physics and Biophysics (NICPB) group and Tallinn University of Technology (TalTech). The task of the TalTech group is to implement the algorithm on the FPGA board in accordance with time and area constraints.

The implementation is based on the High-Level Synthesis (HLS) approach that allows to generate a hardware description of the design from the algorithm written in high-level programming language like C++. HLS tools are intended to decrease the time and complexity of the hardware design development, however, their capabilities are limited. Development of an efficient application requires substantial knowledge of the hardware design and HLS specifics.

The thesis describes the optimizations introduced to the algorithm that improved latency and area and more importantly solved the problems with the routing, making it possible to implement the algorithm on the FPGA fabric.

The thesis is in English and contains 49 pages of text, 5 chapters, 22 figures, 8 tables.

# Annotatsioon

CERNis olev Suur Hadronite Põrguti (Large Hadron Collider — LHC) toodab üha enam uusi andmeid, mistõttu on tekkivate andmete korrektne ja ajakohane töötlemine muutunud üha olulisemaks. Esmalt tuleb suurandmetest efektiivselt üles leida sündmused, mis sisaldavad endas olulist informatsiooni. Üheks selliseks sündmuseks on tau-lepton osakeste lagunemise trigeri algoritmi arendamine, millega tegelevad Keemilise ja Bioloogilise Füüsika Instituudi (NICPB) ja Tallinna Tehnikaülikooli (TalTech) teadlased. Tallinna Tehnikaülikoolis oleva grupi eesmärgiks on luua ja rakendada vastav algoritm programmeeritava ventiilmaatriksi (FPGA) peal, järgides sealjuures kitsendusi nii ajale kui suurusele.

Loodav lahendus põhineb kõrgtaseme sünteesil (HLS), mille abil saab luua riistvara disaini kirjelduse kasutades algoritme, mis on kirjutatud kõrgtaseme programmeerimiskeeltes nagu C++. HLS tööriistade eesmärgiks on vähendada riistvara loomiseks kuluvat aega ning keerukust. Paraku on selliste tööriistade võimekus on limiteeritud, mistõttu on efektiivse lahenduse loomiseks vaja teadmisi nii riistvara disainist kui ka HLS tööriistade eripäradest.

Antud lõputöö kirjeldab algoritmide optimeerimisi, mis aitavad vähendada nii loodava disaini viiteid kui ka suurust. Kõige olulisem osa lõputööst on lahenduste leidmine marsruutimisprobleemidele, mis võimaldab algoritmi rakendada FPGA peal.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 49 leheküljel, 5 peatükki, 22 joonist, 8 tabelit.

# List of abbreviations and terms

| | |
|---|---|
| BRAM | Block Random Access Memory (RAM) |
| CDFG | Control and Data Flow Graph |
| CPU | Central Processing Unit |
| DSP | Digital Signal Processing |
| FF | Flip-Flop |
| FIFO | First In, First Out |
| FPGA | Field-Programmable Gate Array |
| FSM | Finite State Machine |
| GPU | Graphical Processing Unit |
| HDL | Hardware Description Language |
| HLS | High-Level Synthesis |
| II | Initiation Interval |
| IP | Intellectual Property |
| IR | Intermediate Representation |
| LHC | Large Hadron Collider |
| LUT | Look-Up Table |
| MUX | Multiplexer |
| RTL | Register-Transfer Level |
| SIMD | Same Instruction Multiple Data |
| VHDL | VHSIC (Very High Speed Integrated Circuits) HDL |

# Contents

# List of Figures

# List of Tables

# 1. Introduction

With the current increase of the data produced by the Large Hadron Collider (LHC) at CERN, it becomes important to process this data in a corresponding manner. Firstly, to efficiently select events that contain relevant information from a massive flow of data. This is the main objective of the joint project of the National Institute of Chemical Physics and Biophysics (NICPB) group and Tallinn University of Technology (TalTech). Within the scope of the project a tau lepton decay triggering algorithm should be developed and implemented on the FPGA (Field-Programmable Gate Array) board in accordance with time and area constraints. Hardware implementation of the algorithm is the task of the TalTech group.

The implementation is based on the High-Level Synthesis (HLS) approach that allows to generate a hardware description from the algorithm written in high-level programming language like C++. HLS tools are intended to decrease the time and complexity of the hardware design development, which is especially useful in case of compute- and data-intensive applications like the developed algorithm for CERN. However, the capabilities of HLS tools are limited. Proper optimization of the algorithm requires the knowledge of the hardware design and HLS specifics.

The algorithm is developed using Vivado HLS and will be eventually implemented on Xilinx Virtex UltraScale+ FPGA VCU118 Evaluation Kit.

The task of the thesis is to optimize the critical parts of the algorithm in order to solve problems with the timing, area or routing. Considering the selected development approach, in order to optimize the algorithm two tasks should be solved: first, an optimized design development and, second, its implementation in a correct way so that Vivado HLS can synthesize it accordingly.

Several design optimizations were introduced during the work on the project that improved the latency, decreased the resource usage of the algorithm and more importantly solved the problems with the routing, making it possible to implement the algorithm on FPGA fabric.

The thesis is organized in the following way. Chapter 2 presents the main concepts of high-level synthesis and its current state and also gives the overview of the whole algorithm. Chapter 3 provides an overview of the hardware optimizations methods available and describes the optimizations introduced to the algorithm. Chapter 4

gives an overview of the sorting algorithms from the hardware point of view and presents the sorting algorithm developed for the project. Conclusions are given in the Summary section.

# 2. Background

## 2.1. High-Level Synthesis

### 2.1.1. Motivation for HLS

With the increased popularity of the FPGAs (Field-Programmable Gate Array) and heterogeneous systems combining processor units and programming logic on one platform, there arises the necessity to introduce new programming methods for the FPGAs that will make them more accessible for the broader public. FPGAs are integrated circuits consisting of programmable logic blocks and reconfigurable interconnects that run without operating systems and processor-like instructions and can be configured by the user to implement specific applications, hence the name field-programmable.

FPGAs are programmed with hardware description languages (HDL) such as VHDL (Very High Speed Integrated Circuit Hardware Description Language) and Verilog. Hardware description languages use RTL (Register-Transfer Level) abstraction that represents the hardware circuit as a flow of digital signals between storage elements in sequential logic and arithmetic or logical operations performed on those signals in combinatorial logic. RTL design is usually implemented as a datapath controlled by FSM (Finite State Machine). Datapath describes the data flow through the circuit. FSM describes the states that control the data flow and operations on the data. In RTL description, they should be defined explicitly.

FPGAs have certain benefits over the traditional computational platforms like CPU and GPU. Due to their natural parallel computing capabilities, they are faster than CPU. While GPUs are capable of parallel computing as well, FPGAs are characterized by lower cost and much lower power consumption.

However, the complexity of programming that requires the substantial knowledge of the underlying architecture and hardware specifics and increased time to develop the product makes FPGA platforms less accessible for the broader public. Using IP (Intellectual Property) cores, ready to use functional blocks, is one way to solve the problem, they allow to build an application in a Lego-like fashion by combining needed blocks together. Nevertheless, not every functionality can be implemented

using IPs. More complex algorithms require tailored solutions designed specifically for the application.

A better way to solve the problem is the High-Level Synthesis (HLS) tools that introduce a higher level of abstraction for the hardware programming, namely they allow developers to write algorithms in high-level languages like C and C++ that will be automatically translated by the tool into HDL specification taking into account characteristics of the selected board. This way, HLS tools allow non-hardware specialists to program FPGAs. Additionally, they increase portability and maintainability of the application, code written in C/C++ can be easily re-synthesised for different target platforms, debugged and changed in case there is a need. Since input code for HLS describes the algorithm, what the program does, rather than how exactly it is implemented on the target platform.

### 2.1.2. HLS overview

High Level Synthesis, also called Behavioral Synthesis and Algorithm Level Synthesis, consists of two major steps: control logic and datapath extraction and generation of RTL specification. During the first step, a Control and Data Flow Graph (CDFG) is created from the input code. CDFG is a directed graph where each node or basic block represents a statement or a sequence of statements without branches and each edge represents a control flow. Statements include logical and arithmetic operations and assignments, in other words, the data flow. Based on the CDFG a standard architecture with FSM and datapath can be implemented [1, 2].

During the second step, three tasks should be performed: scheduling, allocation and binding. Scheduling refers to scheduling or assigning operations to the specific clock cycles. Scheduling takes into account extracted control flow and datapath, as well as the user constraints. Allocation refers to the allocation or selection of hardware resources necessary for the design: functional units, storage components, buses. The types and amount of elements required to implement the algorithm are defined during the allocation. Binding refers to assigning operations to the specific functional units, variables to the storage elements and data transfers to the connectivity components. Binding stage utilizes the detailed documentation of the target platform architecture: availability and characteristics (such as area, delay and power consumption) of the hardware components.

Simple scheduling and binding example from [3] is presented in Figure 1.

```
int foo(char x, char a, char b, char c) {
 char y;
 y = x*a+b+c;
 return y;
}
```

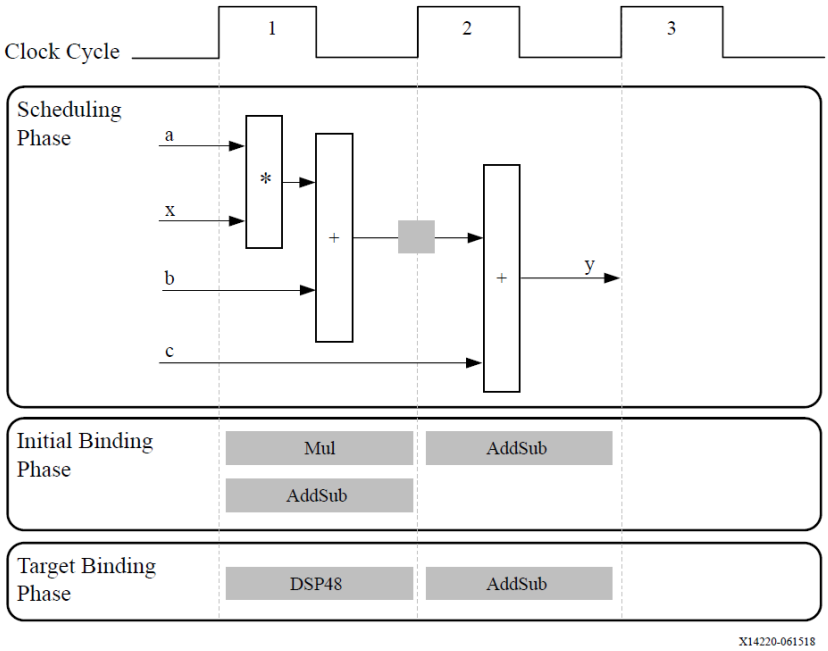


Figure 1. Scheduling and binding example [3]

Allocation, scheduling and binding are often constrained by time or resources, since high-level synthesis approach is oriented towards optimization of the design. HLS task can be considered as an optimization problem to minimize the cost function for one of the design metrics with constraints on the other [4, 5]. High-level synthesis also performs interface synthesis, implementing appropriate interfaces (data signals and hand-shaking control signals) for the data transfer between the periphery and the application, storage components and functional units.

The generalized workflow of the HLS tools is presented in Figure 2. Control and data-path extraction is represented by the first two tasks: compilation and transformation of the input code using optimization directives.

### 2.1.3. Brief history of HLS tools

Early research on high-level synthesis started in the 1970s at Carnegie Mellon University. Martin et al. [7] calls it “groundbreaking research”, however, notes that it did not have much effect on the electronic design automation (EDA) industry.

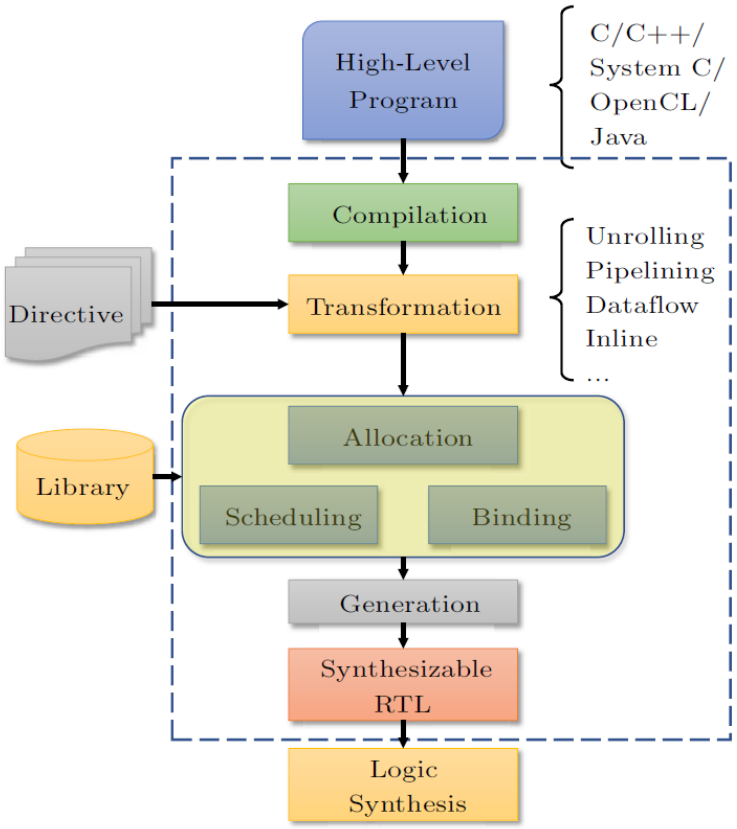


Figure 2. High-level synthesis workflow [6]

In the 1980s HLS gained more interest from the research community. Work done at the time established a strong foundation for the future development and research, however, it was not until the second half of the 1990s when the first commercial HLS tools appeared: Behavioral Compiler from Synopsys, Monet from Mentor Graphics and Visual Architect from Cadence accepted behavioral HDL code as an input. Behavioral HDL code describes the algorithm in a higher level of abstraction than RTL. The new approach was considered very promising [8], however, it was reported that the first generation of HLS tools required "a deep knowledge of their internal synthesis methods and architectural models" [9]. Those tools did not perform architecture optimization, rather synthesizing design that satisfied requirements specified by the designer (amount and type of functional units and memory elements, latency of the design, etc) [4]. And the quality of generated designs was very low [7]. In the end this approach was not able to replace RTL design flow [10, 11] and was considered "a commercial failure" [7].

In the 2000s appeared the new generation of HLS tools that accepted input code in C-like languages and used an iterative optimisation approach. Among the first tools were Forte Cynthesizer, Mentor Graphics Catapult C Synthesis, Celoxica Agility compiler, Bluespec Compiler, NEC CyberWorkBench [4].

Second generation of the HLS tools was more successful for various reasons [7, 10].

1. Improved quality of results.
2. Change of the input language that moved the algorithm description closer to the algorithm specification and allowed system and algorithm designers to use the new tools without the necessity to learn hardware description languages. Additionally, it allows for easier hardware and software co-simulation.
3. The increased deployment of the FPGAs and heterogeneous systems with programmable cores allowing for comparatively simple acceleration of the whole application or some part of it.

The success of the second generation of HLS tools drew a lot of attention to the topic. Recent years have been very prolific for the HLS community, bringing many new tools to the market [12]. Research community published many studies on the design optimization.

Nane et al. [12] reports 17 currently active tools that can be divided into commercial and academic, tools that accept input code written in domain-specific languages and general-purpose languages, tools for general purpose applications and limited to specific domains, like image processing or streaming applications. Numan et al. [13] updates the list, reporting 16 active commercial and academic tools: only in 5 recent years several new tools appeared (Intel FPGA SDK for OpenCL), some old tools have merged (Forte Cynthesizer and Cadence C-to-Silicon Compiler have merged into Stratus HLS), and some lost their importance (eXcite, NAPA-C).

### 2.1.4. Current state of HLS

Current generation of HLS tools has achieved substantial results and gained popularity within industry and academia. One of the reasons of their success, as it was mentioned before, is the use of high-level programming languages at the input. Working with the application written with high-level language, makes it easier to debug and do behavioural verification of the algorithm [14].

According to [15] a complex design utilizing one million gates requires around 300,000 lines of RTL code, however the same algorithm can be expressed in C language with

only 30,000-40,000 lines of code, thus reducing the complexity of the task and the workload.

However, HLS is not a panacea. It still requires domain specific knowledge to write well optimized code. HLS tools suggest different pragma directives for hardware-specific optimization of the algorithm. Nevertheless, those directives alone are not enough to synthesize optimal RTL implementation. They will not be able to improve the algorithm not suitable for the hardware implementation. In order to produce efficient hardware description, the input code should be restructured accordingly. Licht et al. [16] states that naive unoptimized HLS implementations show worser performance than naive software implementations. Matai et al. [17] presents a case study for insertion sort optimization, showing how properly written code can increase the performance of the algorithm, reducing the latency and resource usage. Restructured code shows the best result time- and area-wise compared to the software version of the algorithm optimized with different pragma directives. Results of the case study for comparison are presented in Figure 3.

| | Optimizations | II | Period | Slices | Category |
|---|---|---|---|---|---|
| 1 | L3: pipeline II=1 | 661 | 3.75 | 29 | slow/small |
| 2 | L3: unroll factor=2 cyclic partition array by factor=2 | 730 | 3.84 | 112 | slow/small |
| 3 | L2: pipeline II=1 | 1194 | 3.06 | 47 | slow/small |
| 4 | L2: unroll factor=2 and cyclic partition array by factor=2 | 1193 | 3.50 | 144 | slow/small |
| 5 | L1: pipeline II=1 and complete partition array | 1 | 440.85 | 27291 | faster/huge |
| 6 | Code restructuring | 64 | 2.90 | 374 | fastest/small |

Figure 3. Results of the case study from [17]

Li et al. [1] compares three optimization approaches: polyhedral framework, Vivado HLS optimizations and code refactoring method proposed in the paper. Comparison is based on the synthesis results of four benchmarks. According to the results, proposed method decrease the latency of the applications 5 times more efficiently.

Huang et al. [6] compares HLS-based development process to embedded systems development. Even though companies like STM and Arduino producing development boards provide great support for their products including frameworks for automated settings and board specific libraries in order to make the programming of the devices

easier, it still requires domain specific knowledge to write efficient applications and solve arising problems.

The main difference is that high-level languages used in HLS for input code are not designed to describe hardware specifications, they are designed to describe software instructions executed sequentially, which introduces a challenge of describing hardware constructs with software languages.

Additionally, it requires the knowledge of how HLS tools themselves work, in what order they implement directives, how they represent the algorithm internally, how they synthesize different constructs in order to restructure design of the application in a way that will produce a desirable result.

Even though HLS tools provide an opportunity to create a plethora of different hardware designs from the same input code using different directives and hardware-specific optimization techniques without the necessity to write thousands of lines of code in VHDL or Verilog, the development of the product using HLS approach is still time consuming. Synthesis, code refactoring and identification of the best approach for the product take significant time. A lot of works have been published by the academic community to address the time consuming problem of the design space exploration in HLS [18].

Several works report that quality of results (QoR) of the HLS generated hardware description is far behind that of manually written RTL design [6, 16, 19, 20, 21]. On the contrary, [10] shows an HLS generated design that reduces the resource usage by 11-13 % compared to the design written by RTL expert, proving that HLS can produce competitive results. Case study presented in [22] shows comparable results as well.

Another drawback of the HLS tools is that reported metrics of the design, such as latency and area, are only approximate, and in order to obtain the accurate metrics of the generated design it is necessary to run logic synthesis and implementation, which in turn increase the evaluation time of the design. Therefore, it is impossible to evaluate changes introduced to the algorithm purely from the HLS report and simulation.

A serious limitation of HLS tools is that physical layout and routing estimation is difficult on the HLS level and therefore requires again logic synthesis and implementation in order to identify whether synthesized design has routing problems and

congestions. Reported congested areas should then be mapped to the original code, which is a problem on its own, since the generated hardware description of the design differs drastically from the input code.

Nevertheless, the complex nature of the algorithm that is both compute-intensive and data-intensive calls for the HLS-based development process. And therefore the aforementioned specifics of the HLS should be addressed and solved during the work on the project. The project is developed using Vivado HLS by Xilinx, which is described in the next subsection.

### 2.1.5. Vivado HLS overview

Vivado HLS [23] is the commercial tool provided by Xilinx company. Originally named AutoPilot it was developed by AutoESL. Xilinx bought AutoPilot in 2011 and released the first version of Vivado HLS in 2013 instead of their previous tool AccelDSP. Li et al. [1] calls it "the world-leading HLS tool".

Vivado HLS accepts code written in C, C++ and SystemC as an input and synthesize hardware description in VHDL, Verilog and SystemC. The tool includes a variety of pragma directives for design optimization and libraries for hardware specific features, such as arbitrary precision data types, Xilinx IP (Intellectual Property) functions for streams, shift-registers, etc. Additionally, it provides automatic test-bench creation, C and RTL co-simulation, and support for floating-point and fixed-point arithmetics.

After synthesis Vivado HLS creates a report describing the performance metrics of the generated design, including the maximum frequency of the design based on the longest combinational delay, latency of the design, initiation interval (number of clock cycles before the application can accept new input), amount of utilized resources based on the number of resources available on the target platform, types of interfaces used for input and output signals. The same information is presented for every function and loop instantiated in the design.

In order to guarantee synthesizability of the design, Vivado HLS does not accept some C/C++ language constructs, including recursion, dynamic memory allocation, function pointers and operating system calls.

Vivado HLS uses LLVM compiler infrastructure that first transforms the input code into LLVM-IR (intermediate representation based on LLVM instruction set),

which then undergoes standard compiler transformations and optimizations, such as dead and redundant code elimination, constant propagation, logic and arithmetic expressions refactoring to replace computationally expensive operations with simple ones. On top of that, hardware-specific optimizations are performed: bitwidth analysis and propagation through the design in order to optimize the usage of the resources, memory dependency analysis to uncover parallelism, memory blocks partition to increase memory bandwidth, loop transformations [10, 13].

Optimized LLVM-IR is used for further optimizations during scheduling and binding.

Since Vivado HLS is a commercial tool, its internal workflow is not disclosed, however, the workflow of AutoPilot is known as it is based on the academic project xPilot. It is then logical to assume that the workflow of Vivado HLS is based on it. AutoPilot workflow is presented in Figure 4 [24].

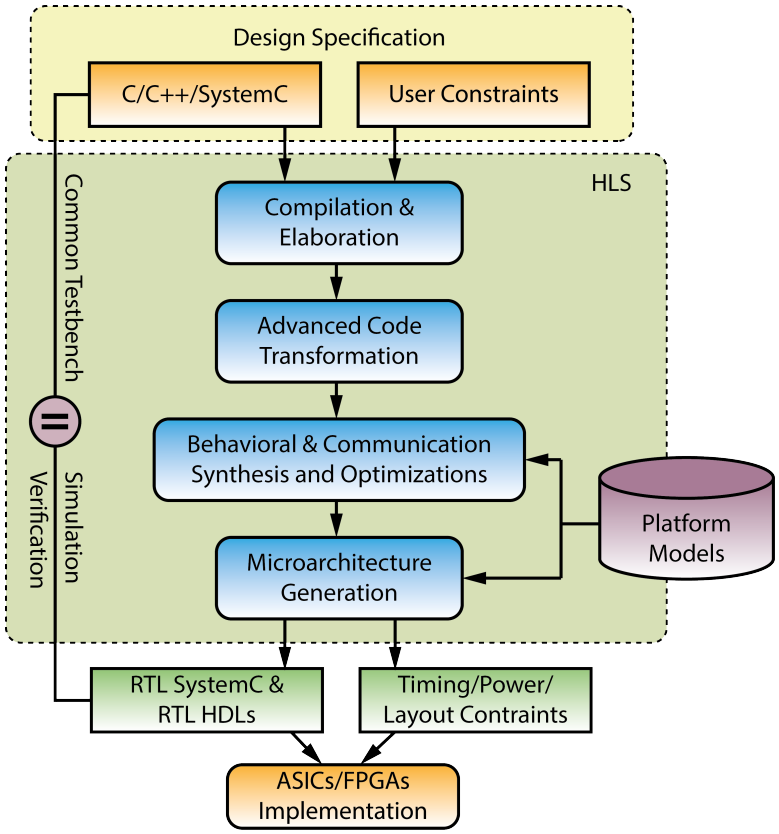


Figure 4. AutoPilot workflow [24]

Design of the system written in C, C++ or SystemC and user defined constraints are accepted as an input. Input code is compiled and passed through a series of compiler transformations. Transformed design representation is then used for HLS optimizations. When synthesis is done, an RTL implementation in HDL or SystemC is generated.

## 2.2. Project overview

The work on the thesis was held within the joint project of the National Institute of Chemical Physics and Biophysics (NICPB) group and Tallinn University of Technology (TalTech) dedicated to the development of the tau lepton decay triggering algorithm for the LHC. The algorithm should be implemented on the Xilinx Virtex UltraScale+ FPGA VCU118 Evaluation Kit, and the hardware implementation of the algorithm according to the time and area constraints is the task of the TalTech group.

Tau lepton is an important particle for analysis of the physical processes happening in LHC, it "plays an important role in both precise measurement of Standard Model physics and search for physics beyond the Standard Model" [25]. However, it has a short life time and short decay length, and can be found and reconstructed only by its decay products. The tau triggering algorithm is designed to identify the events that have hadronically decaying tau leptons [25].

The algorithm consists of 3 major steps. The principal data flow of the algorithm is presented in Figure 5.

At the first step, the event data is buffered and 16 best seeds are selected based on their pT (transverse momentum) value. Every event consists of 36 regions that come one region at a time. Every region contains 22 charged tracks, 13 photon tracks and 10 tracks from neutral particles. For brevity, they will be referred to as tracks, photons and neutrals, the same way as it is done in the code. Data from each region comes packed as one long sequence of bits. Step 1 divides it into separate tracks and buffers them into three 2D arrays. 4 best charged tracks from every region form an array of seed candidates ($4 \times 36 = 144$). 16 tracks with high pT value, which can be an indication of tau lepton decay, are selected as seeds from those candidates.

Selected seeds and buffered data from the whole event are then passed to the select candidates block. The task of this intermediate step is to select up to 30 tau candidates for each seed.

Select candidates block consists of the following stages:

1. Selection of 4 regions for every seed from its neighborhood performed by the select_seed_regions function.

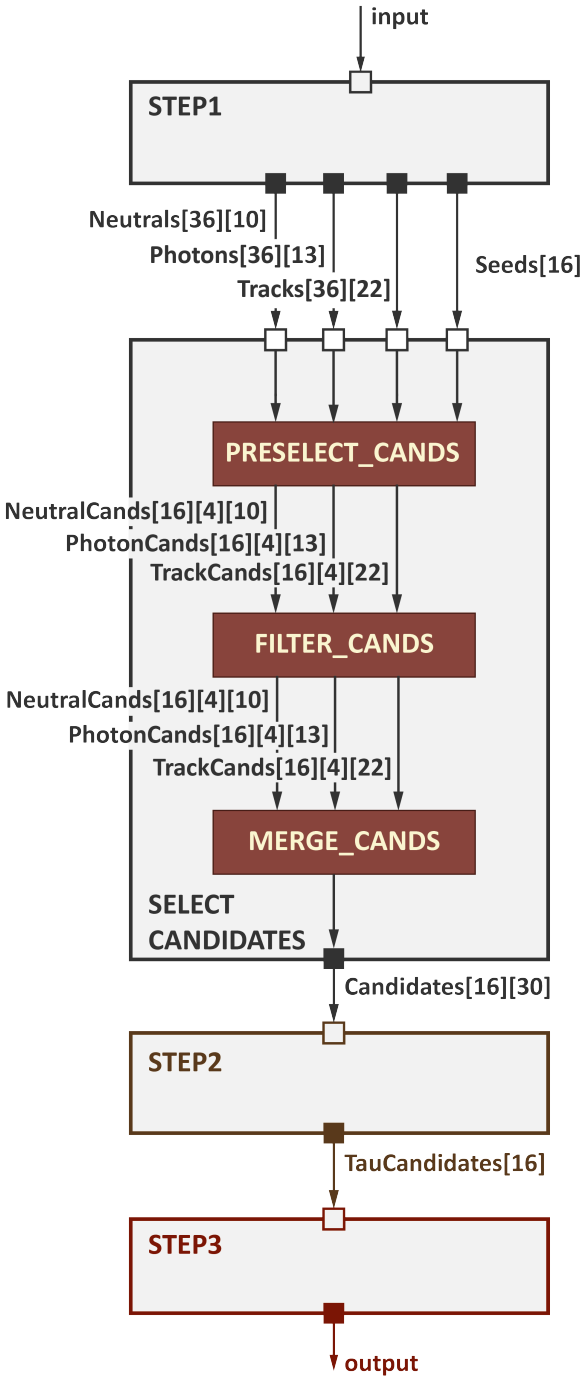


Figure 5. Data flow of the algorithm

2. Saving tracks, photons and neutrals from those regions into separate arrays performed by the preselect_cands function. Originally, it was also done by the select_seed_regions function.

3. Filtering those elements in order to find tau candidates that potentially contain relevant information about tau lepton decay done by the filter_cands function.

4. Merging found candidates ($30 \times 16 = 480$) into one array performed by the merge_cands function.

At the second step, selected tau candidates are analyzed to find signal objects that can be used to reconstruct tau objects. Using obtained data, tau leptons are reconstructed and verified whether they are valid or not.

At the third step, possible duplicates are eliminated from the reconstructed tau leptons and cleaned tau objects are written to the output.

The algorithm is data and computationally intensive and has very strict time and area constraints. The output should be ready within 250 clock cycles. Therefore, it is important that the algorithm is well optimized and area efficient.

Moreover, originally, the algorithm had problems with routing due to the high resource utilization of the step1 function and huge multiplexers in the preselect_cands function that were solved within the work on the thesis.

# 3. Hardware optimizations

The chapter is dedicated to the hardware optimization methods and their availability through HLS tools. The first section of the chapter gives an overview of the optimization methods, the second section describes optimizations introduced in the project.

## 3.1. Overview of the optimization methods in hardware

HLS tools suggest numerous hardware optimization techniques that can be used to create an efficient RTL implementation of the designed application, including several levels of parallelism and memory usage optimizations to improve latency and throughput, arbitrary precision and function inlining to optimize area usage, logic optimization, etc.

*Function level parallelism.* In the RTL description, functions are represented as separate entities that can be executed concurrently if they do not have data dependencies. By default, Vivado HLS generates highly sequential design [26], however, it will try to schedule independent functions to work in parallel if possible. Additionally, there is a way to parallelize functions that have data dependency by using streaming data and dataflow directive. Dataflow region should comply with several requirements: it should follow one producer–one consumer model, meaning data passed between the functions should be written and read exactly once; there should be no feedback between the functions and no conditional execution [3]. Depending on the way the streamed data is accessed, the channels between the functions can be implemented either as FIFOs (First In First Out) or as ping-pong buffers (PIPO). Sequentially accessed data channels are implemented as FIFOs, out-of-order accessed data as PIPOs. The example of the dataflow optimization is presented in Figure 6. Dataflow is a powerful hardware optimization that can significantly improve the latency of the application.

*Loop level parallelism and loop manipulations.* Parallelism in loops can be achieved through unrolling, executing every iteration of the loop at the same time in a SIMD-like (Single Instruction Multiple Data) fashion. The level of parallelism and even the possibility to unroll the loop is determined by the used memory interface and loop-carried dependencies, data dependencies between iterations. In order to resolve

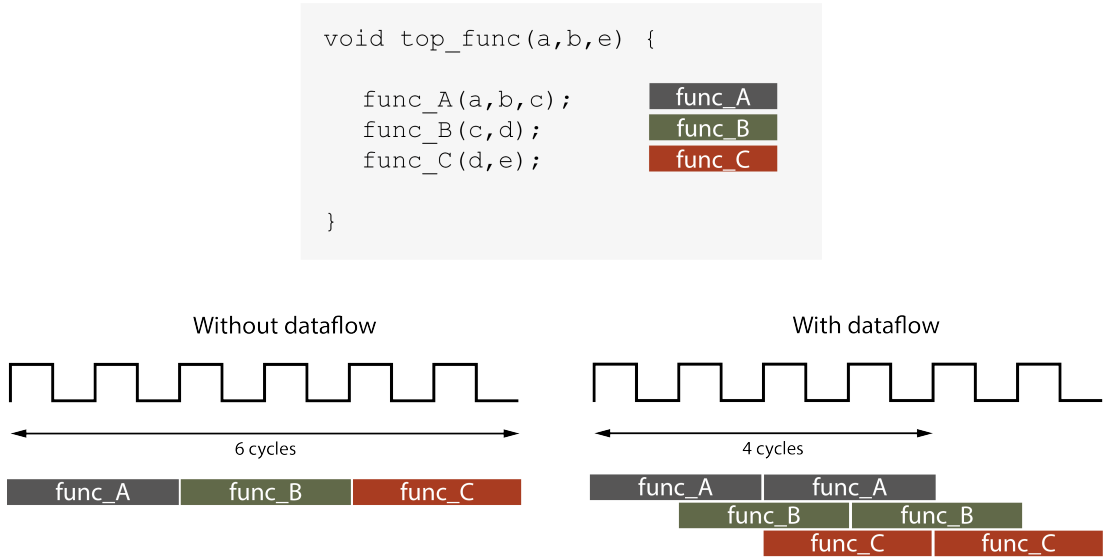


Figure 6. Example of the dataflow optimization

loop-carried dependencies, [16] suggest several solutions including iteration space transposition, buffered accumulation, tiled and batched accumulation, since even partial loop unrolling can increase the throughput and efficiency of the application.

Another way to increase the parallelism of the loops is to merge consecutive loops or flatten nested loops. HLS assigns every loop to a different state during the control and datapath extraction process, therefore, even independent loops will be executed sequentially [1]. Moreover, moving between those states will take one clock cycle. For the nested loops, it is even worse, because there are two transitions between states every outer iteration of the outer loop: from outer to inner and then from inner back to outer [3]. By merging or flattening, loops are assigned to the same state and execute concurrently. It can be done manually by reorganizing the code or by using appropriate directives in Vivado HLS.

*Instruction level parallelism.* Instruction level parallelism can be achieved by pipelining the design. Pipelining can be applied to both functions and loops. Result of the function pipelining is presented in Figure 7. Without pipelining, function can read new input only after fully processing the previous one, thus the total latency of the application equals the latency of the function multiplied by the number of inputs it needs to process. With pipelining, the function can read new input every cycle, while still processing the previous one, thus reducing the latency compared to the original design using the same hardware resources. Morvan et al. [27] calls it "a key transformation in high level synthesis as it helps maximizing both computational throughput and hardware utilization". An important parameter of the pipelined

design is the initiation interval (II), the number of cycles after which a function can accept the new input or a loop can start its next iteration.

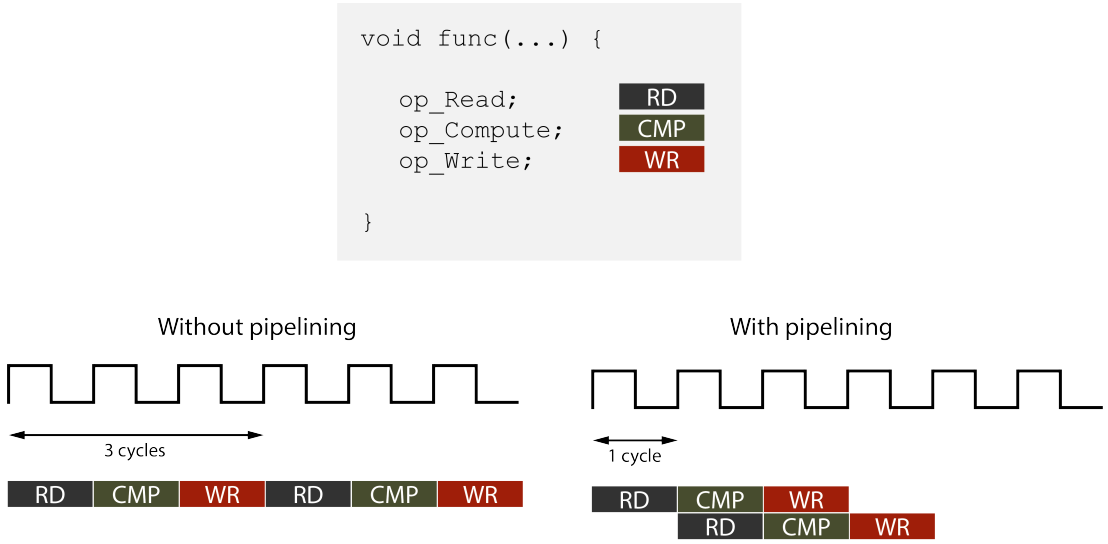


Figure 7. Function execution with and without pipelining

Another way to achieve instruction level parallelism on hardware platforms is an if-conversion [2, 12, 28]. If-conversion refers to the implementation of if-else construction as two independent but mutually exclusive instruction blocks, if block and else block, guarded by the predicate and negated predicate consecutively. It is called a predicated execution and allows for two disjoint branches to run in parallel, thus, reducing the latency of the application. However, it does increase the usage of the resources, as they cannot be shared in that case. Vivado HLS does if-conversion automatically.

*Memory usage optimizations.* As it was already mentioned, the memory interface can prevent the loop from being unrolled or pipelined. For example, BRAM (Block RAM) allows only one read or write access per clock cycle. Therefore, an array mapped to BRAM cannot be vectorized. However, an array can be mapped to individual registers that can be accessed in parallel or to a memory core with more read/write ports if available. For these, Vivado HLS has array_partition and resource directives accordingly. Array partition can be used to divide multidimensional arrays into several arrays along the chosen dimension. It should be noted that complete partitioning of the array into individual elements will increase the resource usage. Therefore, a balance should be found between the latency and the area.

Additionally, arrays can be reshaped or several smaller arrays can be mapped together into a larger one to improve the area using array_reshape and array_map directives.

*Bitwidth optimization.* Unlike traditional processors that support a limited number of data types, hardware platforms allow to create data types of arbitrary precision. Creating operands with the minimum number of bits required to store the data leads to decrease in the area of functional and storage units, shorter critical paths and smaller power consumption [12]. Vivado HLS can automatically determine the exact number of bits needed to store the variables that have their range explicitly specified in the code, e.g. for iterators, however, for most variables the knowledge of the input data is required.

*Function inline.* It was already mentioned that each function is synthesized into a separate entity in the RTL implementation. Inlining simplifies the hierarchy of the design by assembling several functions into one entity, thus eliminating interface connections and providing a possibility to increase resource sharing. This can improve the overall latency and area of the design. At the same time, inlining increases the complexity of the synthesized functional units and can cause routing congestions [29]. Vivado HLS automatically inlines small functions, designers can enforce inlining using pragma directive.

*Logic and arithmetic expressions optimization.* Expressions optimization can increase instruction level parallelism, improve the latency of the design and decrease the area. One of the possible optimizations is expression balancing or height reduction of long expression chains [2], rearrangement of operands for a balanced implementation. For example, an expression like $sum = a + b + c + d$ will result in sequential computation that will take 3 clock cycles. In comparison, balanced expression $sum = (a+b)+(c+d)$ will be computed in 2 clock cycles, since terms $(a + b)$ and $(c + d)$ can be calculated in parallel. The architectures for both examples are presented in Figure 8. Vivado HLS does expression balancing for integers automatically.

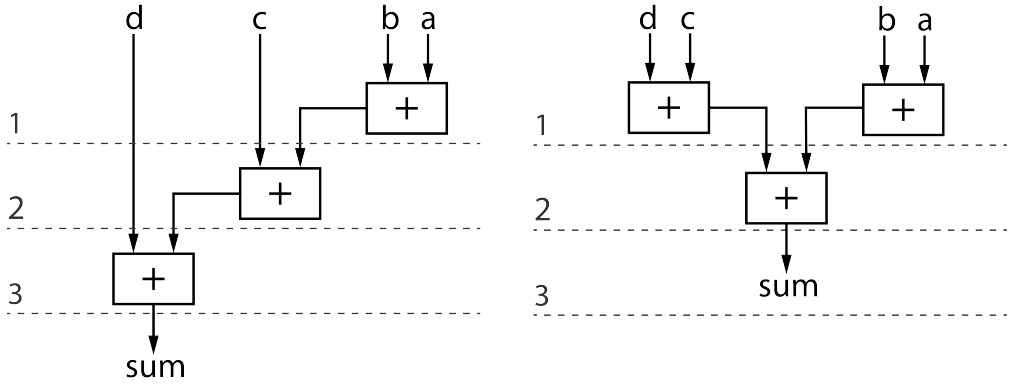


Figure 8. Adder tree before balancing (left) and after balancing (right)

Another possible expression optimization is a transformation of computationally expensive arithmetic operations like multiplication and division into shifting and addition/subtraction. Such transformations are performed even by software compilers, however, they become more crucial for hardware platforms. In Vivado HLS, arithmetic expression optimizations are handled by the LLVM compiler that can handle simple cases but will not perform transformations that are not beneficial for a CPU, like replacing one multiplication with 2 additions and 2 shifts [30]. Therefore, more intricate arithmetic expression optimizations should still be done by the designer.

## 3.2. Optimizations introduced to the project

### 3.2.1. Design space exploration of seed regions selection

The first task was to optimize select_seed_regions function that accepted buffered tracks, photons and neutrals and one of the selected seeds as an input and output tracks, photons and neutrals from the region, where the seed was found, plus from the three adjacent regions for further analysis. Originally, the selection of the regions was static: for each region the other three regions were predefined. Hence, the original solution consisted of 37 if-else statements (one statement for each region number) and three loops to write data from selected regions to the output arrays (see Figure 9). Additionally, the original design did not have any pragma directives, so it was a great opportunity to explore the possibilities of Vivado HLS.

Several designs were tried in order to optimize the function, including:

- generalization of the if-else statements that decreased their amount to 5 and then to 3,
- mathematical solution that replaced if-else statements with simple formulas,
- matrix solution that stored the predefined regions into a 2D matrix.

The results for different solutions are given in Table 1. Results include the latency of the function in clock cycles and the amount of used flip-flops (FF) and Look-up tables (LUT).

```
region_num_t Region1, Region2, Region3, Region4;

if ( Seed.hwRegion == 0 ) {
      Region1 = 0;  Region2 = 1;  Region3 = 2;  Region4 = 3;
} else if ( Seed.hwRegion == 1 ) {
      Region1 = 0;  Region2 = 1;  Region3 = 2;  Region4 = 3;
<...>
} else if ( Seed.hwRegion == 35 ) {
      Region1 = 32; Region2 = 33; Region3 = 34; Region4 = 35;
} else {
      Region1 = 0; Region2 = 1; Region3 = 2; Region4 = 3;
}

for (int j = 0; j < NTRACK; ++j) {
      Seed_Tracks[0][j] = All_Tracks[Region1][j];
      Seed_Tracks[1][j] = All_Tracks[Region2][j];
      Seed_Tracks[2][j] = All_Tracks[Region3][j];
      Seed_Tracks[3][j] = All_Tracks[Region4][j];
}
for (int j = 0; j < NPHOTON; ++j) {
      <...>
}
for (int j = 0; j < NSELCALO; ++j) {
      <...>
}
```

Figure 9. Original design of the select_seed_regions function

The pattern observed from the first series of experiments showed that the decrease in amount of if-else statements and, therefore, decrease in the amount of predicated instructions executed in parallel, decrease the amount of FF/LUTs used. Those experiments also proved the point that by default Vivado HLS will generate a sequential design, executing rolled loops one after another.

The next set of experiments explored the aforementioned designs with the function pipelining and additionally the effect of using different memory cores to store the matrix in the fifth design. The results are presented in Table. 2.

Pipelined design truly decreased the latency of the function but noticeably increased the number of LUTs used. It should be noted that the initiation interval in every case was equal to the latency of the function. It is not always possible to pipeline the design in a way that new input can be accepted while the previous one is processed, nevertheless, pipeline directive forces Vivado HLS to generate more parallelized architecture.

Table 1. Static region selection designs

| **Design** | **Latency, cycles** | **FF** | **LUT** |
|---|---|---|---|
| Original design | 139 | 231 (<1%)* | 2504 (<1%) |
| Design #2 (5 if-else statements) | 139 | 218 (<1%) | 1601 (<1%) |
| Design #3 (3 if-else statements) | 139 | 220 (<1%) | 1564 (<1%) |
| Design #4 (mathematical solution) | 139 | 214 (<1%) | 1528 (<1%) |
| Design #5 (matrix solution) | 139 | 243 (<1%) | 1447 (<1%) |

* Number in the brackets is the percentage from the total number of resources available on the board.

Table 2. Static regions selection designs with function pipeline

| **Design** | **Latency, cycles** | **BRAM** | **FF** | **LUT** |
|---|---|---|---|---|
| Original design | 45 | 0 | 207 (<1%) | 11,622 (1%) |
| Design #2 (5 if-else statements) | 45 | 0 | 203 (<1%) | 10,844 (<1%) |
| Design #3 (3 if-else statements) | 45 | 0 | 196 (<1%) | 10,844 (<1%) |
| Design #4 (mathematical solution) | 45 | 0 | 195 (<1%) | 10,644 (<1%) |
| Design #5 (matrix solution, matrix is implemented as distributed ROMs) | 46 | 0 | 212 (<1%) | 10,691 (<1%) |
| Design #5 (matrix solution, matrix is mapped to BRAM) | 46 | 1 | 182 (<1%) | 10,726 (<1%) |

An alternative way to decrease the latency of the design is by pipelining the loops instead of the whole function. The results of the next set of experiments are presented in Table 3. They show that latency can be decreased without significant increase in the used resources as well, though the latency of the function pipeline version is indeed much better.

Later, the algorithm got updated and selection of the regions became dynamic. The regions are organized in a $4 \times 9$ grid and folded like a torus, so the first and last

Table 3. Static regions selection designs with pipelined loops

| **Design** | **Latency, cycles** | **FF** | **LUT** |
|---|---|---|---|
| Original design | 97 | 210 (<1%) | 2564 (<1%) |
| Design #2 (5 if-else statements) | 96 | 193 (<1%) | 1661 (<1%) |
| Design #3 (3 if-else statements) | 97 | 195 (<1%) | 1624 (<1%) |
| Design #4 (mathematical solution) | 97 | 193 (<1%) | 1588 (<1%) |
| Design #5 (matrix solution) | 97 | 218 (<1%) | 1507 (<1%) |

rows are connected. The regions are selected based on the position of the seed on the region. The position of the seed is defined by its Eta and Phi parameters: Eta specifies horizontal coordinate and Phi specifies vertical coordinate. Center of the region is denoted as $(0, 0)$, the coordinates are local to the region. The developed dynamic solution is presented in Figure 10. The results for the dynamic regions selection is presented in Table 4.

Table 4. Dynamic region selection results

| **Design** | **Latency, cycles** | **FF** | **LUT** |
|---|---|---|---|
| Function pipeline | 45 | 206 (<1%) | 10,624 (<1%) |
| Function pipeline + array partition | 11 | 45 (<1%) | 71,960 (6%) |

By partitioning all input arrays and increasing the number of read/write ports, it became possible to unroll the loops and decrease the latency of the function further, however, 4× increase in speed came along with the 7× increase in the area, which should be taken into account during the optimization process.

It can be seen that even this small function provided a great opportunity for design space exploration, including both manual changes of the design and automatic changes caused by pragma directives applied.

```
ap_uint<4> row;
ap_uint<2> col;
(row,col) = Seed.hwRegion;

/* Helping variables denote whether seed is in the region
 * on the border of the 4x9 grid */
ap_uint<1> x_edge_0 = (col == 0);
ap_uint<1> x_edge_1 = (col == 3);
/* If the region is on the 0th or 8th row,
 * we can add or subtract 8
 * to get 0 as the next row for the  8th row
 * and 8 as the previous row for the 0th row */
ap_uint<4> y_edge_0 = (row == 0) ? 7 : 0;
ap_uint<4> y_edge_1 = (row == 8) ? 7 : 0;

ap_uint<2> x0, x1;
ap_uint<4> y0, y1;

/* Eta gives x coordinate in the region,
 * Phi gives y coordinate.
 * Center of the region is denoted as (0,0) */
if (Seed.hwEta < 0) {
      x0 = col - 1 + x_edge_0;      x1 = col + x_edge_0;
 }
else {
      x0 = col - x_edge_1;          x1 = col + 1 - x_edge_1;
}

if (Seed.hwPhi < 0) {
      y0 = row - 1 - y_edge_0;      y1 = row;
}
else {
      y0 = row;                  y1 = row + 1 + y_edge_1;
}

Region1 = (y0,x0);     Region2 = (y0,x1);
Region3 = (y1,x0);     Region4 = (y1,x1);
```

Figure 10. Region selection based on the seed coordinates

### 3.2.2. Control logic optimization of the candidates preselection

There was a problem with the way tracks, photons and neutrals from the selected regions were read from the input and written to the output. Access through non-sequential indices created huge multiplexers and caused serious routing problems. A new representation of selected regions and a new way to write the data from the buffered arrays were required. At the time, the selection of regions and retrieving

the data from those regions were divided between two functions: select_seed_regions and preselect_cands.

A new way to represent the selected regions by the row and column according to Figure 11 was suggested by the supervisors.

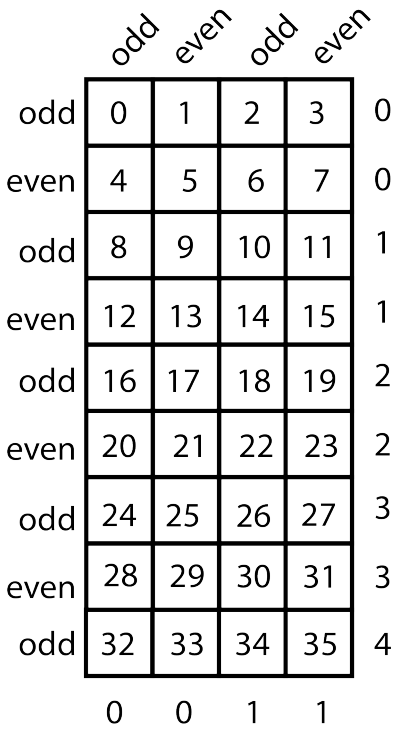


Figure 11. New regions representation

Every row and column was marked as either odd or even, and every odd-even pair was numbered. This way, selected regions can be represented by their position in the grid in the following fashion: region 1 is represented by the odd row and odd column, region 2 is represented by the odd row and even column, region 3 is represented by the even row and odd column, and region 4 is represented by the even row and even column. Now, instead of 4 regions function select_seed_regions returns a structure with rows and columns. Since the first row and the last row are both odd, two additional parameters are added to the structure that indicates whether the last row is used or not and if it is used then whether it is considered as even or as odd. The structure is presented in Figure 12. New design for the select_seed_regions function is presented in Appendix 1.

However, more important changes should have been introduced to the preselect_cands function. It was thought that changes in the selected regions representation should result in 8-to-1 MUXs (multiplexers) for row selection, 2-to-1 MUXs for last row

```
typedef struct seed_regions_t {
    ap_uint<2> rowEven;
    ap_uint<2> rowOdd;
    ap_uint<1> lastRowEven;
    ap_uint<1> lastRowOdd;
    ap_uint<1> colEven;
    ap_uint<1> colOdd;
} seed_regions_t;
```

Figure 12. Seed regions structure

selection, and 2-to-1 MUXs for column selection instead of previous 36-to-1 MUXs. The problem was to make Vivado HLS infer it from the design.

Preselect candidates function was redesigned in the following way. First, two rows should be selected: one even and one odd. Since the number of rows is uneven, the last row is considered as a special case. This way, if the candidate row is not the last one, then the selection is done from only four rows: 0, 2, 4, 6 for the even row and 1, 3, 5, 7 for the odd. Then, from those two rows, two columns should be selected. Again, one even and one odd, therefore, in each case the selection is done from two columns: 0 and 2 for the even column and 1 and 3 for the odd.

In order to make Vivado HLS synthesize appropriate MUXs, the following idea was implemented (presented on the example of tracks.) Tracks were saved in two arrays: 2D $4 \times 8$ track array for the first 8 rows and 1D trackLastRow array for the last row. Each row in the track array contains two rows from the original grid. This way, all even rows are on the left and all odd rows are on the right. Selected rows are copied to two 2D $2 \times 2$ arrays: tRowEven and tRowOdd. This way, even columns end up in the first column and odd columns in the second column. The function code is presented in Appendix 2.

Described architecture is presented in Figure 13. Those shapes make it clear for Vivado HLS which elements are valid for each selection. This design gives 2-to-1 MUXs for last row selection, 4-to-1 MUXs for row selection, and 2-to-1 for column selection, which is even better than predicted.

This task made it clear that creating a well optimized algorithm using HLS tools requires to solve two problems: first, to design an efficient architecture and, second, to write an implementation that Vivado HLS will be able to synthesize according to the idea.

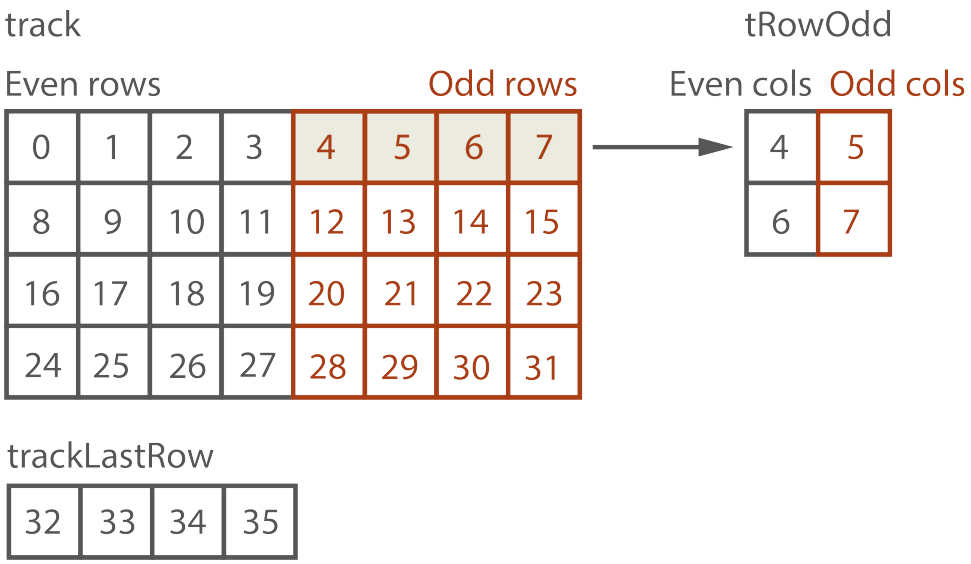


Figure 13. Preselect candidates selection process

### 3.2.3. Latency optimization of the second step

Another task was to explore the possibility to decrease the latency of the step 2 functions. Step 2 functions process 30 selected candidates for every seed in order to find those that carry tau lepton decay data. Functions step2_3 and step2_4a process one candidate for every seed at a time and work in parallel. step2_3 writes 16 signal candidates at a time, and passes them to the step2_4a function that accumulates the tau parameters data for every seed. step2_4b waits for the step2_4a to finish, because it needs the accumulated data from step2_4a. The idea was to change the way the data is processed in order to make all three functions execute in parallel and pass the data in dataflow fashion.

Functions step2_3 and step2_4a were rewritten in order to process all candidates for one seed at a time, this way, function step2_4b gets the data seed by seed and can start working on it before step2_4a finishes. The new design required to change the dimensionality of the input and output data for functions step2_3 and step2_4a, change of the memory interface for data passed between function step2_4a and step2_4b from individual elements to FIFO, and introduction of the buffered accumulation in step2_3 and step2_4a.

The total latency of three functions has decreased from 64 cycles to 51 cycles. The resource utilization results are presented in Table 5.

Table 5. Step 2 functions results

| **Function** | | **DSP48*** | **FF** | **LUT** |
|---|---|---|---|---|
| step2_3 | original | 16 | 15,120 (<1%) | 14,049 (1%) |
| | modified | 30 | 39,077 (1%) | 38,447 (3%) |
| step2_4a | original | 32 | 4336 (<1%) | 6922 (<1%) |
| | modified | 30 | 2068 (<1%) | 5893 (<1%) |
| step2_4b | original | 0 | 9850 (<1%) | 5860 (<1%) |
| | modified | 0 | 2577 (<1%) | 2027 (<1%) |

* Digital Signal Processing (DSP) block.

It can be seen that the resource usage for function step2_3 increased more than twice as expected, since now it needs 30 units working in parallel compared to 16 before, however, the change in data processing scheme decreased the amount of resources for the other two functions. This example shows that the change in scheduling and data processing can affect different functions in a different way that should be considered during the design analysis.

### 3.2.4. Discussions

Different optimizations were introduced during the work on the algorithm, proving that HLS tools provide a convenient way for design space exploration of the algorithm without the need to rewrite the source code and give an opportunity to analyse possible RTL implementations in terms of latency–area trade-off and select the most suitable one in a manageable time. It also proved that a restructured code or a better design, optimized for both hardware and high-level synthesis workflow, will always show better results compared to the naive implementation even with the pragma directives applied. Additionally, a routing problem was solved by introducing a more efficient design of the preselect_cands function and by implementing this design in a way that will create a correct CDFG of the function. Instead of huge 36-to-1 multiplexers, the suggested version of the preselect_cands function uses 4-to-1 and 2-to-1 multiplexers.

# 4. Sorting

The first step of the algorithm is dedicated to buffering the input data and selecting 16 seeds out of 144 candidates with the highest pT (transverse momentum) value. The candidates are sorted, and 16 elements from the top are saved into the Seeds array for further analysis.

The first section of the chapter presents the overview of the sorting algorithms from the hardware point of view. The second section describes the development process of the new sorting algorithm for the project.

## 4.1. Sorting algorithms overview from the hardware point of view

Effectiveness of the sorting algorithm is usually determined based on the amount of time or steps and the amount of memory the algorithm needs to sort an input sequence denoted by the big $O$ notation. Big $O$ notation defines the asymptotic upper bound. Even though the actual time and amount of memory used depend on many factors, including the size of the sorted data, the order of the elements, etc, big $O$ notation is a good approximation of the algorithm efficiency for comparison purposes.

However, for hardware implementation of sorting algorithms there are specific factors that predetermine their effectiveness, such as possible frequency, complexity of the sorting logic and consequently the amount of area used, possibility to decrease the latency by exploiting computational parallelism. The overview given below is based on the factors important for hardware implementation.

*Selection sort.* Selection sort divides the input array into two parts: sorted and unsorted. It iteratively finds either minimum or maximum in an unsorted part of the array and swaps it with the first element in an unsorted part, thus expanding the sorted part. The algorithm is simple and area efficient, however, it has latency of $O(n^2)$, where $n$ is the length of the input array, even for hardware implementation. Parallel search for minimum and maximum can decrease the latency of the algorithm only to $O(n^2/2)$ [17].

*Rank sort.* Rank sort is composed of two steps. At the first step, the rank of each element is calculated. Rank is defined by the number of elements greater or smaller than the one to be sorted. At the second step, elements are rearranged based on their rank indices. The algorithm has a latency of $O(n^2)$ as well, however, the rank calculation step can be easily parallelized with $n$ functional units working at the same time, giving the latency of $O(n)$. Unfortunately, the parallel implementation has a high storage requirement of $O(2n^2)$ [17] and might produce massive multiplexers for the second step, where elements should be written to the new array based on their ranks.

*Insertion sort.* Insertion sort, similar to selection sort, divides the input array into sorted and unsorted parts, and iteratively finds a place for each element from the unsorted part in the sorted one, shifting bigger or smaller elements if necessary. The latency of the insertion sort is $O(n^2)$, however, it can be implemented in hardware in the form of a linear or spatial sorter with latency $O(n)$ or $O(2n)$. Spatial sorter is a linear sequence or an array of $n$ nodes or cells that can sort $n$ elements in $2n$ steps [17, 31]. Each node is an insertion cell primitive consisting of a comparator, a multiplexer, a register to keep the current value and a control logic. The node compares the input value with the current value, the smallest or biggest value depending on the sorting order is saved in the register, the other value is passed to the next cell. The architecture of the spatial sorter is shown in Figure 14. The architecture does work well with the streaming data.

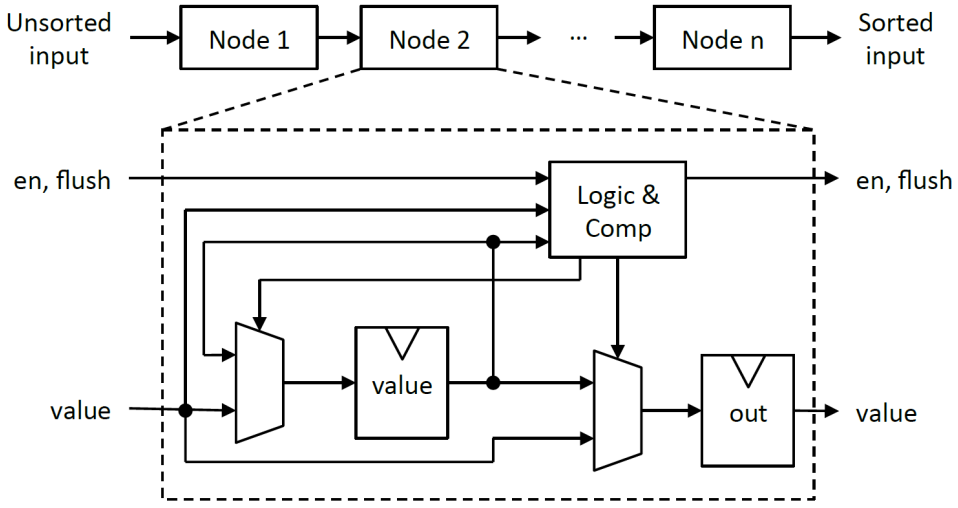


Figure 14. Architecture of the spatial sorter [31]

Another linear architecture is possible that can sort $n$ elements in $n$ steps. Instead of traversing through the whole sequence of nodes one by one, the new input is sent to all nodes at the same time. Each node compares the input value with the current value and its neighbors values, when the new value is inserted in the correct place,

the bigger or smaller values are shifted [32, 33, 34]. The described architecture is presented in Figure 15.

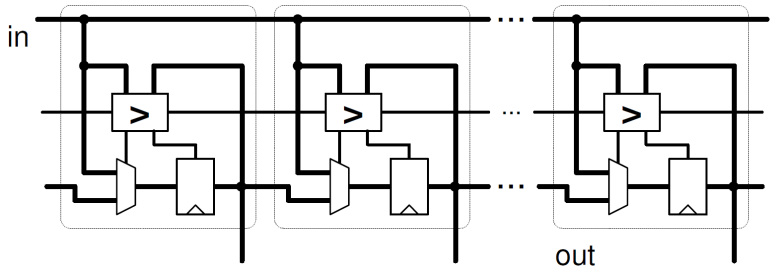


Figure 15. Architecture of the linear sorter [34]

Hardware implementation of insertion sort requires n sorting units working in parallel, and is considered as a fairly good choice for smaller arrays [34, 35].

*Merge sort.* Merge sort is a recursive algorithm that successively merges the smaller subsequences into a big sorted sequence. The latency of merge sort is $O(n \log n)$. In hardware, merge sort is usually implemented without recursion. Instead a streaming architecture is used. Basic merge sorter consists of two input FIFOs, a select-value primitive (comparator and multiplexer) and an output FIFO [34, 36]. Input FIFOs can be sorted using other sorting algorithms or basic merge sorters can be organized in a tree based architecture for efficient sorting of big sequences with linear time complexity, since the latency of the streaming merge sort is $O(n)$. However, unlike linear insertion sort for example, it requires more storage units.

*Quick sort.* Quick sort is a very popular software algorithm using the divide-and-conquer approach. It randomly selects a pivot point and moves all elements smaller than the pivot on one side and all elements greater than the pivot on the other side, then recursively repeats the procedure for each part. The time complexity of the algorithm is $O(n^2)$ in the worst case and $O(n \log n)$ on average. However, because of its recursive nature and the fact that the efficiency of the algorithm highly depends on the selected pivot point, the algorithm is not used much in hardware [17].

*Counting sort.* Counting sort is an example of non-comparison sorting algorithms. Counting sort is composed of three stages. On the first stage, for each unique value the number of times it appears in an input array is calculated and saved in a temporary array, creating a histogram. The size of the temporary array depends on the maximum value in the input array. On the second stage, the index for each element in the output array is calculated by incrementing each value in the temporary array by the sum of all previous values. On the third stage, the elements are moved from the input array to the output using values from the temporary array as indices.

Parallel counting sort has a latency of $O(n)$ and requires $O(nk)$ storage area, where $k$ is the maximum value in the input array [17]. Additionally, the last stage, similar to rank sort, can create huge multiplexers.

*Sorting networks.* Sorting network is a very popular sorting algorithm for hardware implementation due to their parallel nature. Sorting network is a set of compare-swap primitives connected by wires. Two common examples of sorting network architectures are presented in Figure 16. Bitonic sorting network has the time complexity of $O(\log^2 n)$ and requires $O(n \log^2 n)$ sorting logic and $O(n \log^2 n)$ storage. The advantage of the sorting network architecture is a very high throughput, however, it is achieved at the expense of a very high resource usage [17, 31, 32, 33, 34, 37].

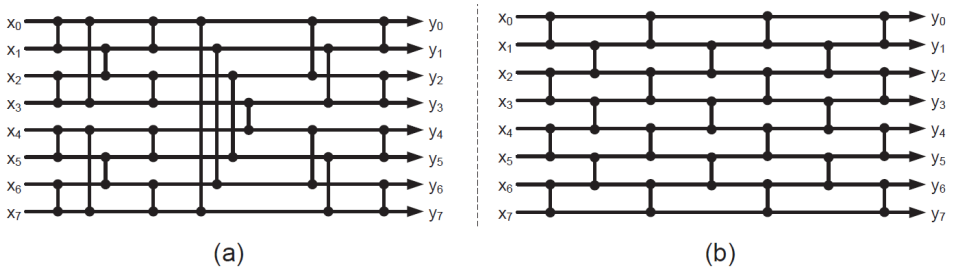


Figure 16. Bitonic sorting network (a) and odd-even transposition sorting network (b) [17]

For a similar application, Farmahini-Farahani et al. [38] suggest an $n$-to-$m$ sorting units that select $m$ largest (or smallest) elements from $n$ inputs ($m < n$) based on the bitonic sorting algorithm. Proposed solution has a time complexity of $O(\log n \times \log m)$ and area complexity $O(n \log^2 m)$. Even though it is an improvement over the classic bitonic sorting network, it still requires a lot of resources. It also assumes that data comes all at once.

From reviewed sorting algorithms two algorithms can be emphasized: insertion sort and merge sort. Insertion sort is recommended for smaller arrays due to its simplicity and fairly good time complexity. In addition, it is a good candidate for the tau trigger project because of its natural ability to handle streaming data. Merge sort gives a simple parallel architecture for a small amount of data and suits for streaming data as well. The way they can be implemented in the project is discussed in the next section.

## 4.2. Development of the sorting algorithm

As it was said before, the first step of the algorithm has two important tasks: buffering of the input data and selecting 16 seeds out of 144 seed candidates. Input data is a continuous stream of particle events. Each event consists of 36 regions, new region data comes every clock cycle. Every region consists of 22 tracks, 13 photons and 10 neutrals. First 4 tracks from every region form an array of seed candidates ($36 \times 4 = 144$). Later an array of seed candidates is sorted to find 16 best seeds based on their pT value.

### 4.2.1. Original algorithm

Original sorting algorithm was a hardware optimized version of bubble sort that was taking too much resources causing problems with the routing of the whole algorithm. The task was to develop a new sorting algorithm that will significantly decrease the amount of utilized resources and potentially decrease the latency of the first step, since the total latency of the algorithm is very strict. The latency of the first step can be defined as $max(B, S)$, where $B$ is the latency of the buffering process and $S$ is the latency of the sorting. Buffering takes 56 clock cycles, original sorting solution takes 57 clock cycles. Therefore, an optimized solution should take the same amount of clock cycles or less than the buffering.

At the beginning it was decided to try some modifications of the original algorithm to explore possible optimizations without introducing a completely new sorting algorithm. Each seed candidate object is represented by a structure with 6 members. During the sorting, seed candidates are read and written numerous times. The idea was to use a smaller structure containing pT value and index of the candidate for the sorting and then write 16 best seeds using obtained indices. Another idea was instead of a smaller structure use a temporary array of $8 + 16$ bit integers that will hold both the index of the candidate (8 bits) and its pT value (16 bits). Both approaches only slightly reduced the resource usage of the function, did not reduce the latency and introduced a problem of using many multiplexers at the stage when the selected seeds should be written to the output array based on their indices in the seed candidates array. The results are presented in Table 6.

Eventually, it was decided to try a different sorting algorithm. The algorithm should take into account the streaming nature of incoming data, the fact that it comes

Table 6. Results of the original algorithm modifications

| **Algorithm** | **Latency, cycles** | **FF** | **LUT** |
|---|---|---|---|
| Original | 57 | 151,287 (6%) | 262,690 (22%) |
| Smaller struct | 58 | 132,448 (5%) | 208,487 (17%) |
| $8 + 16$ bit integer | 72 | 146,627 (6%) | 238,428 (20%) |

in chunks of 4, and that only 16 elements out of 144 should be saved for further processing. The straightforward approach to use some hardware optimized sorting architecture to sort all 144 elements would not be applicable, because even sorting algorithms with time complexity $O(n)$ would take much more clock cycles than desired.

### 4.2.2. Streaming merge sort

At first it was decided to implement a merge sorter tree that at every level will leave only 16 best elements, discarding the other. The architecture of the sorter tree is presented in Figure 17.

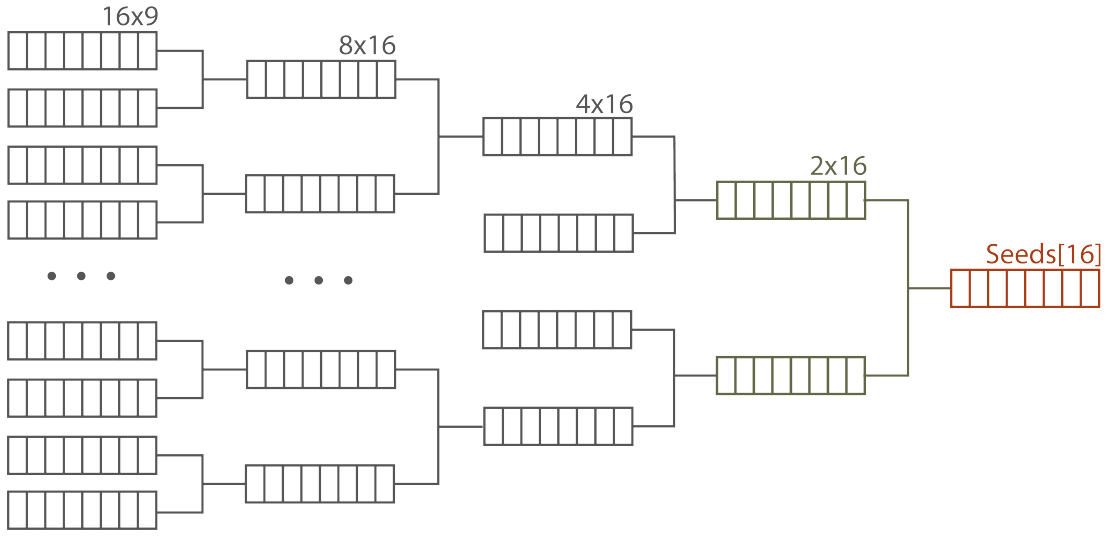


Figure 17. Merge sorter tree architecture

On the first level, there are 16 arrays, 9 elements each ($16 \times 9 = 144$). On the second level, they are merged into 8 arrays of size 16, and so on until there is only one array left with the best seeds. 16 arrays on the first level were sorted in an insertion sort manner: a new coming element would traverse through the array to find its place, then if needed elements greater than the new one will be shifted and a new element inserted. To parallelize the process, each seed candidate from one region would be

inserted into a different array. This way, new element is inserted in the array on the first level every third cycle, giving the previous element 2 clock cycles to find its place, which was proved sufficient during the runs of the algorithm.

Merge unit is based on the example from [17] and is presented in Figure 18.

```
void merge_unit16(PFSeedObj IN1[SIZE],
                  PFSeedObj IN2[SIZE],
                  PFSeedObj OUT[NSEED])
                  {
#pragma HLS ARRAY_PARTITION variable=IN1 complete
#pragma HLS ARRAY_PARTITION variable=IN2 complete
#pragma HLS ARRAY_PARTITION variable=OUT complete
#pragma HLS inline
    PFSeedObj a, b;
    int subIdx1 = 1, subIdx2 = 1;
    a = IN1[0]; b = IN2[0];

    for (int i = 0; i < NSEED; i++){
    #pragma HLS PIPELINE
        if (subIdx1 == SIZE){
            OUT[i] = b;
            b = IN2[subIdx2];
            subIdx2++;
        }
        else if (subIdx2 == SIZE){
            OUT[i] = a;
            a = IN1[subIdx1];
            subIdx1++;
        }
        else if (a.hwPt > b.hwPt) {
            OUT[i] = a;
            a = IN1[subIdx1];
            subIdx1++;
        }
        else {
            OUT[i] = b;
            b = IN2[subIdx2];
            subIdx2++;
        }
    }
}
```

Figure 18. Merge unit implementation

Three different variants of the merge sorter were tried: OUT arrays implemented as BRAMs, OUT arrays partitioned and implemented as separate registers, and finally OUT arrays implemented as FIFO streams. The results prove that streams are the best solution for the merge sorter as they decrease the latency of the merge stage.

However, they have a certain drawback: all elements from the previous level that are not passed to the next level should anyway be read from the stream. The results are presented in Table 7.

Table 7. Results of the original algorithm modifications

| **Algorithm** | **Latency, c** | **BRAM** | **FF** | **LUT** |
|---|---|---|---|---|
| Merge sort: OUT arrays as BRAM | 104 | 48 | 35,333 (1%) | 146,998 (12%) |
| Merge sort: OUT arrays partitioned | 84 | 0 | 63,552 (2%) | 439,358 (37%) |
| Merge sort: OUT arrays as streams | 72 | 0 | 41,117 (1%) | 391,176 (33%) |

It can be seen that the implemented merge sort does not fit into defined latency and does not decrease the amount of resources used and therefore does not satisfy the task description. The increased latency is attributed to the fact that it was required to divide the buffering process and merge sorter between different functions. Buffering requires a pipeline directive in order to read new region data every clock cycle and start processing new event data every 36 cycles. Merge sorter though requires a dataflow directive to make all merge levels work in parallel; and Vivado HLS cannot instantiate dataflow region from pipelined function. Therefore, merge sorter waits until the buffering stage is over and starts to work only after.

### 4.2.3. Spatial insertion sort

First algorithm developed that improved the timing and the resource usage was a modification of insertion sort. A spatial sorter was built that included 16 insertion cells. The sorting architecture is presented in Figure 19.

The sorting algorithm takes into account that 4 inputs from each region are already sorted and, therefore, 4 elements are used as an input for each insertion cell, unlike traditional implementation that has 1 new coming element as an input.

Each insertion cell works on one element of the Seeds array. First, the insertion cell gets 4 elements from the new region and compares it to the first element of the Seeds array. The biggest element is saved to the CURR_REG variable. Since it is known that elements in the IN array are sorted, the biggest element is either CURR_REG or IN[0]. If it is not CURR_REG, IN[0] is saved into the CURR_REG, the previous

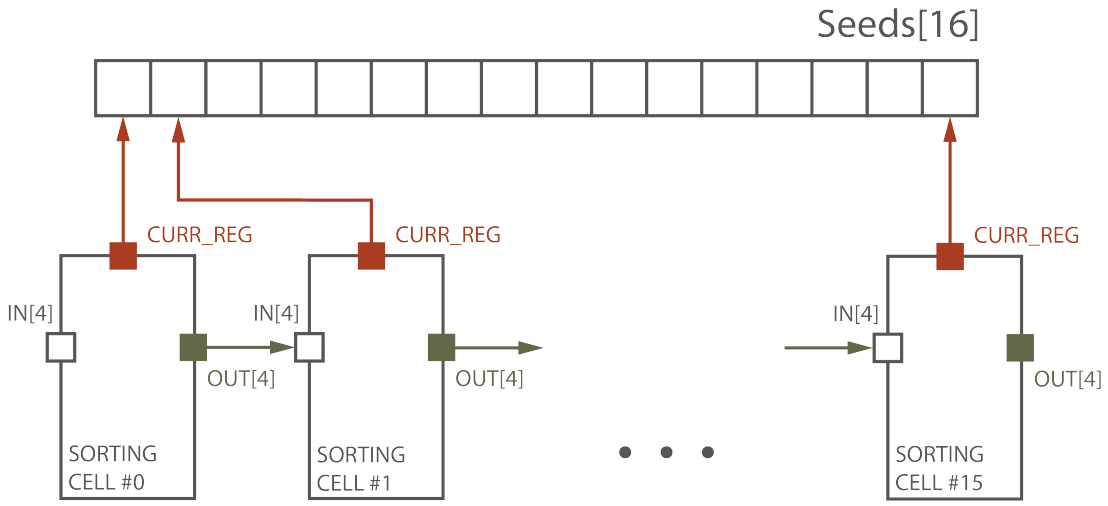


Figure 19. Spatial sorter architecture

value of CURR_REG is inserted in the appropriate place in the OUT array. The OUT array is then passed to the next sorting cell. The work of the insertion cell is presented in Figure 20. The code for the insertion cell primitive is presented in Figure 21. For the whole step1 function, see Appendix 3.

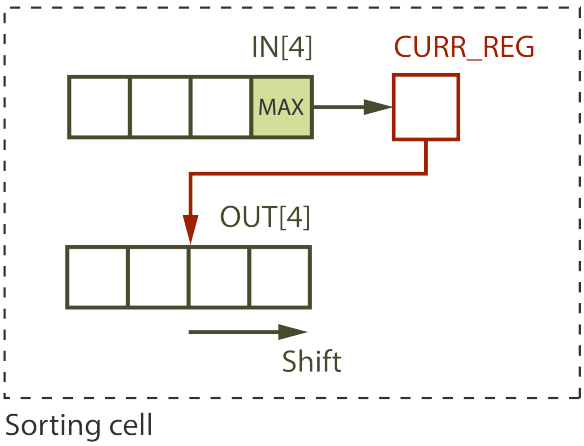


Figure 20. Sorting cell architecture

In order to make Vivado HLS implement 16 different instantiations of the insertion cell function, so each of them will use their own static variable CURR_REG, the function is defined as a template with integer parameter number that acts as an ID number of sort. Otherwise Vivado HLS will generate the design, in which all insertion cells will share the same CURR_REG variable.

With this approach, sorting takes $36 + 1 + 16$ (53 cycles), because the first region elements are passed to the first sorting cell on the second cycle and the elements from the last region take 16 cycles to pass through all sorting cells. Since the latency of the first step function is determined by either buffering or sorting, depending on

```
template <int number>
PFSeedObj insertion_cell(PFSeedObj IN[4], PFSeedObj OUT[4],
   int i) {
#pragma HLS ARRAY_PARTITION variable=IN complete
#pragma HLS ARRAY_PARTITION variable=OUT complete
    static PFSeedObj CURR_REG = {0};

    int idx = 3;
    PFSeedObj max = IN[0];
    PFSeedObj temp = CURR_REG;

    for (int j = 0; j < 4; j++) {
#pragma HLS UNROLL
        OUT[j] = IN[j];
        if (j == 0 && CURR_REG.hwPt >= IN[0].hwPt) {
            idx = 0;
            max = CURR_REG; temp = IN[0];
        }
        if (j != 3 && CURR_REG.hwPt < IN[j].hwPt
            && CURR_REG.hwPt >= IN[j + 1].hwPt) {
            idx = j;
            max = IN[0]; temp = CURR_REG;
        }
    }

    if (i != 35) CURR_REG = max;
    else clear(CURR_REG);
    shift_array(OUT,idx);
    OUT[idx] = temp;
    return max;
}
```

Figure 21. Insertion cell implementation

which operation takes more clock cycles, the total latency of the function with the described sorting algorithm is 56 clock cycles.

It was possible to modify the created algorithm to decrease the timing of sorting part even more. Instead of working on 1 element from the Seeds array, new sorting cells work on 2 elements from the Seeds array. Graphically, a modified sorter is presented in Figure 22.

The architecture of the insertion cell has changed as well. The solution for the modified insertion cell was inspired by [39] that presents a single-stage $N$ sorter based on a comparison counting matrix. Suggested $N$-sorter does not require to calculate the rank of each element, it is built according to the equations derived from the comparison counting result.

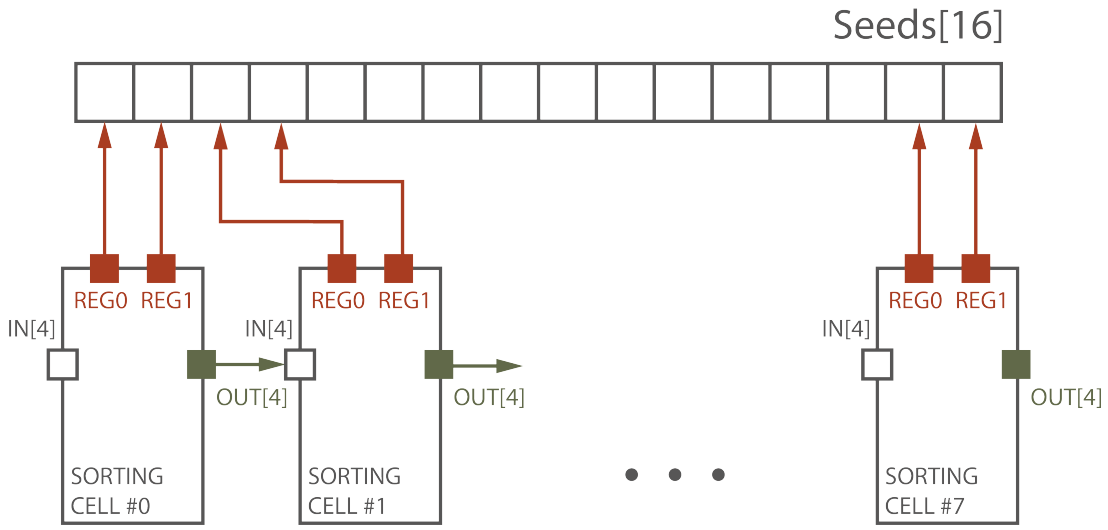


Figure 22. Modified spatial sorter

Since it is known that the IN array is sorted and REG0 and REG1 are sorted, several simple rules were deducted to sort those elements. 6 elements should be sorted into 6 output positions: out1, out2, out3, out4, out5, and out6. out1 and out2 are saved into REG0 and REG1, out3–out6 are saved into the OUT array.

The examples of the rules are presented below:

1. Only two elements can go to the out1: REG0 or IN[0]. The biggest one goes to the out1.
2. Four elements can go to the out2: REG0, REG1, IN[0] or IN[1]. If REG1 is bigger than IN[0], then it goes to the out2. If IN[0] is smaller than REG0 but greater than REG1, then it goes to the out2. If IN[1] is bigger than REG0, then it goes to the out2. Else, REG0 goes to the out2.
3. The rules for the other output positions are depicted in the same way. For the code, see Appendix 4.

The latency of the modified sorting algorithm is $36 + 1 + 8$ (45) cycles. The resource usage for the spatial sorter modifications is presented in Table 8.

It can be seen that developed spatial sorter significantly decrease the usage of the resources and provide an opportunity to decrease the latency of the first step function if it will be possible to decrease the latency of the buffering process.

Table 8. Results for spatial insertion sorter versus original algorithm

| **Algorithm** | **Latency, cycles** | **FF** | **LUT** |
|---|---|---|---|
| Original | 57 (57)* | 151,287 (6%) | 262,690 (22%) |
| Spatial sorter | 56 (53) | 104,749 (4%) | 33,822 (2%) |
| Modified spatial sorter | 56 (45) | 103,783 (4%) | 22,987 (1%) |

* The latency of the sorting process is given in the brackets.

### 4.2.4. Discussions

Two different sorting algorithms were implemented and compared with the original solution: streaming merge sort and spatial insertion sort. Merge sort solution has shown that not every design can be implemented using HLS due to the limitations of the tools. It was not possible to write the code in a way that Vivado HLS can schedule merge sorter with a streaming data in parallel with the buffer process.

Insertion sort solution, on the other hand, was able to reduce the resource usage of the function drastically (7× less LUTs for the first version and 11× less LUTs for the modified version compared to the original algorithm) as well as decrease the latency of the sorting part (21 % for the modified version) and solve the problem with the routing. Suggested insertion sort algorithm takes into account the properties of the input data (new data comes presorted in blocks of 4) and the task specification (only 16 best elements out of 144 should be selected), proving that a solution tailored to the task provides a better design.

# 5. Summary

Large Hadron Collider (LHC) experiments produce a huge amount of data that should be processed in a timely manner. Before this data can be used for scientific analysis, it should be filtered in order to extract events that contain relevant information. This is the task of the tau lepton decay triggering algorithm developed by National Institute of Chemical Physics and Biophysics (NICPB) and Tallinn University of Technology (TalTech). The algorithm should be implemented on an FPGA platform according to very strict time and area constraints.

Due to its complexity, the algorithm is developed using high-level synthesis approach (HLS). However, the capabilities of the HLS tools are limited, and an intimate knowledge of the hardware specifics is required in order to design and implement an efficient algorithm.

The goal of the thesis was to optimize the algorithm in terms of run-time and resource usage using Vivado HLS. Two problems should have been solved during the work on the thesis: development of the well optimized design for the task and proper implementation of it so that the HLS tool will synthesize it accordingly.

The main contribution of the thesis is the optimization of the sorting task in the first step and the candidates selection task.

For the first step of the algorithm, an application-specific sorting algorithm based on the spatial insertion sort was developed that decreased the amount of resources used 7 times compared to the original solution. Moreover, this improvement of the area resolved the problem with routing, making it possible to implement the algorithm on the FPGA fabric. The modified version of the suggested sorting architecture decreased the number of used resources even more and uses 11 times less resources than the original solution. In addition, the modified version decreased the latency of the sorting part by 21 %, thus allowing to decrease the total latency of the step1 function in the future if the buffering part will be optimized in terms of time.

For the candidates selection task, two functions, select_seed_regions and preselect_cands, were rewritten to introduce a new way to select 4 regions for each seed and access the data from those regions in order to avoid huge multiplexers in the RTL design that originally caused problems with routing as well. New design instead

of 36-to-1 MUXs uses 4-to-1 and 2-to-1 MUXs, which take less area and do not cause problems with implementation on FPGA.

For the future work, the presented changes in data processing pattern in the second step can be considered further in order to find whether it will be possible to introduce it in the previous steps and decrease the latency of the whole algorithm.

# References


[1] C. Li, Y. Bi, Y. Benezeth, D. Ginhac, and F. Yang, "High-level synthesis for FPGAs: code optimization strategies for real-time image processing," *Journal of Real-Time Image Processing*, vol. 14, no. 3, pp. 701–712, oct 2017.

[2] N. D. Dutt, D. D. Gajski, S. Y.-L. Lin, and A. C.-H. Wu, *High-Level Synthesis: Introduction to Chip and System design.* Springer US, Feb. 1992.

[3] Xilinx, "Vivado design suite user guide: High-level synthesis," Tech. Rep., 2018.

[4] D. Bruni, A. Bogliolo, and L. Benini, "Statistical design space exploration for application-specific unit synthesis," in *Proceedings of the 38th conference on Design automation - DAC '01.* ACM Press, 2001.

[5] G. Micheli, *Synthesis and optimization of digital circuits.* New York: McGraw-Hill, 1994.

[6] L. Huang, D.-L. Li, K.-P. Wang, T. Gao, and A. Tavares, "A survey on performance optimization of high-level synthesis tools," *Journal of Computer Science and Technology*, vol. 35, no. 3, pp. 697–720, may 2020.

[7] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18–25, Aug. 2009.

[8] (1994) Synopsys sings the praises of its behavioral compiler for chip designers. [Last access: 24.04.2021]. [Online]. Available: https://techmonitor.ai/techonology/synopsys_sings_the_praises_of_its_behavioral_compiler_for_chip_designers

[9] E. M. G Savaton, E Casseau, "Behavioral VHDL styles and high-level synthesis for IPs," *FDL 2000, Forum on Design Languages*, pp. 107–115, Sep. 2000.

[10] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, apr 2011.

[11] J. Cong, B. Liu, G. Luo, and R. Prabhakar, "Towards layout-friendly high-level synthesis," in *Proceedings of the 2012 ACM international symposium on International Symposium on Physical Design - ISPD '12.* ACM Press, 2012.

[12] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, oct 2016.

[13] M. W. Numan, B. J. Phillips, G. S. Puddy, and K. Falkner, "Towards automatic high-level code deployment on reconfigurable platforms: A survey of high-level synthesis tools and toolchains," *IEEE Access*, vol. 8, pp. 174 692–174 722, 2020.

[14] J. Goeders and S. J. E. Wilton, "Allowing software developers to debug HLS hardware," Aug. 2015.

[15] K. Wakabayashi, "C-based behavioral synthesis and verification analysis on industrial design examples," in *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '04. IEEE Press, 2004, pp. 344–348.

[16] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1014–1029, may 2021.

[17] J. Matai, D. Richmond, D. Lee, Z. Blair, Q. Wu, A. Abazari, and R. Kastner, "Resolve: Generation of high-performance sorting architectures from high-level synthesis," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, feb 2016.

[18] N. K. Pham, A. K. Singh, A. Kumar, and M. M. A. Khin, "Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*. IEEE Conference Publications, 2015.

[19] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen, "High-level synthesis: Productivity, performance, and software constraints," *Journal of Electrical and Computer Engineering*, vol. 2012, pp. 1–14, 2012.

[20] K. Rupnow, Y. Liang, Y. Li, D. Min, M. Do, and D. Chen, "High level synthesis of stereo matching: Productivity, performance, and software constraints," in *2011 International Conference on Field-Programmable Technology*. IEEE, dec 2011.

[21] S. Lahti, P. Sjovall, J. Vanne, and T. D. Hamalainen, "Are we there yet? A study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, may 2019.

[22] Z. Zhao and J. C. Hoe, "Using Vivado HLS for structural design," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, feb 2017.

[23] Vivado design suite—Vivado HLS. Xilinx Inc. [Last access: 04.05.2021]. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[24] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "AutoPilot: A platform-based ESL synthesis system," in *High-Level Synthesis*. Springer Netherlands, 2008, pp. 99–112.

[25] Y. Sakurai, "The ATLAS tau trigger performance during LHC Run 1 and prospects for Run 2," *arXiv: High Energy Physics - Experiment*, Sep. 2014.

[26] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel programming for FPGAs," *ArXiv e-prints*, May 2018.

[27] A. Morvan, S. Derrien, and P. Quinton, "Efficient nested loop pipelining in high level synthesis using polyhedral bubble insertion," in *2011 International Conference on Field-Programmable Technology*. IEEE, dec 2011.

[28] M. Hohenauer, F. Engel, R. Leupers, G. Ascheid, H. Meyr, G. Bette, and B. Singh, "Retargetable code optimization for predicated execution," in *2008 Design, Automation and Test in Europe*. IEEE, mar 2008.

[29] J. Zhao, T. Liang, S. Sinha, and W. Zhang, "Machine learning based routing congestion prediction in FPGA high-level synthesis," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, mar 2019.

[30] Y. Uguen, F. D. Dinechin, V. Lezaud, and S. Derrien, "Application-specific arithmetic in high-level synthesis tools," *ACM Transactions on Architecture and Code Optimization*, vol. 17, no. 1, pp. 1–23, mar 2020.

[31] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Lujan, "An empirical evaluation of high-level synthesis languages and tools for database acceleration," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, sep 2014.

[32] J. Ortiz and D. Andrews, "A streaming high-throughput linear sorter system with contention buffering," *International Journal of Reconfigurable Computing*, vol. 2011, pp. 1–12, 2011.

[33] M. Zuluaga, P. Milder, and M. Püschel, "Streaming sorting networks," *ACM Transactions on Design Automation of Electronic Systems*, vol. 21, no. 4, pp. 1–30, sep 2016.

[34] D. Koch and J. Torresen, "FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '11*. ACM Press, 2011.

[35] Y. B. Jmaa, R. B. Atitallah, D. Duvivier, and M. B. Jemaa, "A comparative study of sorting algorithms with FPGA acceleration by high level synthesis," *Computación y Sistemas*, vol. 23, no. 1, mar 2019.

[36] R. Marcelino, H. Neto, and J. M. Cardoso, "Sorting units for FPGA-based embedded systems," in *Distributed Embedded Systems: Design, Middleware and Resources*. Springer US, 2008, pp. 11–22.

[37] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on FPGAs," *The VLDB Journal*, vol. 21, no. 1, pp. 1–23, jun 2012.

[38] A. Farmahini-Farahani, A. Gregerson, M. Schulte, and K. Compton, "Modular high-throughput and low-latency sorting units for FPGAs in the Large Hadron Collider," in *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*. IEEE, jun 2011.

[39] R. B. Kent and M. S. Pattichis, "Design, implementation, and analysis of high-speed single-stage N-sorters and N-filters," *IEEE Access*, vol. 9, pp. 2576–2591, 2021.

# Appendix 1 Select seed regions

```
void select_seed_regions(const PFSeedObj& Seed,
            seed_regions_t& regions) {

    ap_uint<1> row1_coeff, row2_coeff;
    ap_uint<1> col1, col2, col1_coeff;
    ap_uint<2> t2;
    ap_uint<3> row1, row2, t1;

    /* Find row and column of the seed region */
    (row1, t1) = Seed.hwRegion;
    (row1_coeff, t2) = t1;
    (col1, col1_coeff) = t2;

    /* Find rows based on the coordinates.
    * Eta gives x coordinate in the region,
    * Phi gives y coordinate.
    * Center of the region is denoted as (0,0) */
    if (Seed.hwEta < 0) {
        if (col1_coeff == 1)
            col2 = col1;
        else col2 = 0;
    } else {
        if (col1_coeff == 0)
            col2 = col1;
        else
            col2 = 1;
    }

    if (Seed.hwPhi < 0) {
        row2 = row1 - (row1_coeff ^ 1);
    }
    else {
        row2 = row1 + row1_coeff;
    }

    row2_coeff = row1_coeff ^ 1;
    if (row2 == 7) {
        row2 = 4; row2_coeff = 0;
    }
```

```
    if (row2 == 4 && row2_coeff == 1) {
        row2 = 0; row2_coeff = 0;
    }


    if (col1_coeff == 0) {
        regions.colEven = col1; regions.colOdd = col2;
    }
    else {
        regions.colEven = col2; regions.colOdd = col1;
    }

    if (row1_coeff == 0 && row2_coeff == 1) {
        (regions.lastRowEven,regions.rowEven) = row1;
        (regions.lastRowOdd,regions.rowOdd) = row2;
    }
    else if (row1_coeff == 1 && row2_coeff == 0) {
        (regions.lastRowEven,regions.rowEven) = row2;
        (regions.lastRowOdd,regions.rowOdd) = row1;
    }
    else {
        (regions.lastRowEven,regions.rowEven)=(ap_uint<3>)0;
        (regions.lastRowOdd,regions.rowOdd) = (ap_uint<3>)4;
    }
}
```

# Appendix 2 Pre-select candidates

```
void select_seed_regions(
    const seed_regions_t seedRegions[NSEED],
    hls::stream<PFChargedObj> allTracks[NREGIONS],
    hls::stream<PFNeutralObj> allPhotonsNeutrals[NREGIONS],
    hls::stream<PFChargedObj> trackCands[NSEED][NSEEDREGIONS
       ],
    hls::stream<PFNeutralObj> neutralCands[NSEED][
       NSEEDREGIONS])
{

// Three loops will be merged into one
#pragma HLS LOOP_MERGE

// Pre-select tracks
for (int i = 0; i < NTRACK; i++) {
#pragma HLS PIPELINE II=1

  PFChargedObj track[4][8];
  PFChargedObj trackLastRow[4];
#pragma HLS ARRAY_PARTITION variable=track complete dim=0
#pragma HLS ARRAY_PARTITION variable=trackLastRow complete

  for (int j = 0; j < 4; j++) {
      for (int c = 0; c < 8; c++){
          allTracks[j*8 + c].read(track[j][c]);
      }
  }
  for (int j = 0; j < 4; j++){
      allTracks[32 + j].read(trackLastRow[j]);
  }

  for (int j = 0; j < NSEED; j++) {
      ap_uint<2> rowEven = seedRegions[j].rowEven;
      ap_uint<2> rowOdd = seedRegions[j].rowOdd;
      ap_uint<1> lastRowEven = seedRegions[j].lastRowEven;
      ap_uint<1> lastRowOdd = seedRegions[j].lastRowOdd;
      ap_uint<1> colEven = seedRegions[j].colEven;
      ap_uint<1> colOdd = seedRegions[j].colOdd;
```

```
      PFChargedObj tRowEven[2][2], tRowOdd[2][2];
#pragma HLS ARRAY_PARTITION variable=pRowEven complete dim=0
#pragma HLS ARRAY_PARTITION variable=pRowOdd complete dim=0

      for (int r = 0; r < 2; r++) {
          for (int c = 0; c < 2; c++) {
              if (lastRowEven == 1)
                  tRowEven[r][c] = trackLastRow[r*2+c];
              else tRowEven[r][c] = track[rowEven][r*2+c];
              if (lastRowOdd == 1)
                  tRowOdd[r][c] = trackLastRow[r*2+c];
              else tRowOdd[r][c] = track[rowOdd][(r*2)+(c+4)
                 ];
          }
      }

      trackCands[j][0].write(tRowEven[colEven][0]);
      trackCands[j][1].write(tRowEven[colOdd][1]);
      trackCands[j][2].write(tRowOdd[colEven][0]);
      trackCands[j][3].write(tRowOdd[colOdd][1]);
      }
}

// Pre-select photons/neutrals
for (int i = 0; i < NPHOTON+NSELCALO; i++) {
#pragma HLS PIPELINE II=1

  PFNeutralObj neutral[4][8];
  PFNeutralObj neutralLastRow[4];
#pragma HLS ARRAY_PARTITION variable=photonNeutral complete
   dim=0
#pragma HLS ARRAY_PARTITION variable=photonNeutralLastRow
   complete

  for (int j = 0; j < 4; j++) {
      for (int c = 0; c < 8; c++)
          allPhotonsNeutrals[j*8 + c].read(neutral[j][c]);
  }
  for (int j = 0; j < 4; j++){
      allPhotonsNeutrals[32 + j].read(neutralLastRow[j]);
  }
```

```
  for (int j = 0; j < NSEED; j++) {
      ap_uint<2> rowEven = seedRegions[j].rowEven;
      ap_uint<2> rowOdd = seedRegions[j].rowOdd;
      ap_uint<1> lastRowEven = seedRegions[j].lastRowEven;
      ap_uint<1> lastRowOdd = seedRegions[j].lastRowOdd;
      ap_uint<1> colEven = seedRegions[j].colEven;
      ap_uint<1> colOdd = seedRegions[j].colOdd;

      PFNeutralObj pRowEven[2][2], pRowOdd[2][2];
#pragma HLS ARRAY_PARTITION variable=pRowEven complete dim=0
#pragma HLS ARRAY_PARTITION variable=pRowOdd complete dim=0

      for (int r = 0; r < 2; r++) {
          for (int c = 0; c < 2; c++) {
              if (lastRowEven == 1)
                  pRowEven[r][c] = neutralLastRow[r*2+c];
              else pRowEven[r][c] = neutral[rowEven][r*2+c];
              if (lastRowOdd == 1)
                  pRowOdd[r][c] = neutralLastRow[r*2+c];
              else pRowOdd[r][c] = neutral[rowOdd][(r*2)+(c
                 +4)];
          }
      }

      neutralCands[j][0].write(pRowEven[colEven][0]);
      neutralCands[j][1].write(pRowEven[colOdd][1]);
      neutralCands[j][2].write(pRowOdd[colEven][0]);
      neutralCands[j][3].write(pRowOdd[colOdd][1]);
  }
}
}
```

# Appendix 3 Step 1 v.1

```
void algo_kbfi_step1(
            hls::stream<ap_uint<STREAM_SIZE>> &stream_in,
            hls::stream<PFChargedObj> Tracks[NREGIONS],
            hls::stream<PFNeutralObj> Photons[NREGIONS],
            hls::stream<PFNeutralObj> Neutrals[NREGIONS],
            PFSeedObj Seeds[NSEED]) {
#pragma HLS ARRAY_PARTITION variable=Seeds complete
#pragma HLS data_pack variable=Tracks
#pragma HLS data_pack variable=Photons
#pragma HLS data_pack variable=Neutrals

#pragma HLS PIPELINE II=36

for (int i = 0; i < NREGIONS; i++){
#pragma HLS PIPELINE
    // Read next region
    ap_uint<STREAM_SIZE> buf;
    stream_in.read(buf);

    // Extract relevant data
    ap_uint<PACKING_DATA_SIZE> data[NCHANN_IN];
#pragma HLS ARRAY_PARTITION variable=data complete
    for (int j = 0; j < NCHANN_IN; ++j) {
#pragma HLS UNROLL
        data[j] = buf(PACKING_DATA_SIZE*(j+1)-1,
                                PACKING_DATA_SIZE*j);
    }

    PFSeedObj Buffer[4], OUT1[4], OUT2[4], OUT3[4], OUT4[4];
    PFSeedObj OUT5[4], OUT6[4], OUT7[4], OUT8[4], OUT9[4];
    PFSeedObj OUT10[4], OUT11[4], OUT12[4], OUT13[4]
    PFSeedObj OUT14[4], OUT15[4], OUT16[4];

    // Extract tracks/seeds
    for (int j = 0; j < NTRACK; j++) {
        PFChargedObj TrackCand;
#pragma HLS data_pack variable=TrackCand

        l1pf_pattern_unpack<1,0>(&data[j], &TrackCand);
```

```
        Tracks[i].write(TrackCand);

        if (j < NSEEDNEW) {
            // Fetch first seeds and add as seed candidates
            Buffer[j].hwRegion = i;
            Buffer[j].hwPt = TrackCand.hwPt;
            Buffer[j].hwEta = TrackCand.hwEta;
            Buffer[j].hwPhi = TrackCand.hwPhi;
            Buffer[j].hwId = TrackCand.hwId;
            Buffer[j].hwZ0 = TrackCand.hwZ0;
        }
    }
    if (i != 35) {
        insertion_cell<0>(Buffer,OUT1,i);
        insertion_cell<1>(OUT1,OUT2,i);
        insertion_cell<2>(OUT2,OUT3,i);
        insertion_cell<3>(OUT3,OUT4,i);
        insertion_cell<4>(OUT4,OUT5,i);
        insertion_cell<5>(OUT5,OUT6,i);
        insertion_cell<6>(OUT6,OUT7,i);
        insertion_cell<7>(OUT7,OUT8,i);
        insertion_cell<8>(OUT8,OUT9,i);
        insertion_cell<9>(OUT9,OUT10,i);
        insertion_cell<10>(OUT10,OUT11,i);
        insertion_cell<11>(OUT11,OUT12,i);
        insertion_cell<12>(OUT12,OUT13,i);
        insertion_cell<13>(OUT13,OUT14,i);
        insertion_cell<14>(OUT14,OUT15,i);
        insertion_cell<15>(OUT15,OUT16,i);
    }
    else {
        Seeds[0] = insertion_cell<0>(Buffer,OUT1,i);
        Seeds[1] = insertion_cell<1>(OUT1,OUT2,i);
        Seeds[2] = insertion_cell<2>(OUT2,OUT3,i);
        Seeds[3] = insertion_cell<3>(OUT3,OUT4,i);
        Seeds[4] = insertion_cell<4>(OUT4,OUT5,i);
        Seeds[5] = insertion_cell<5>(OUT5,OUT6,i);
        Seeds[6] = insertion_cell<6>(OUT6,OUT7,i);
        Seeds[7] = insertion_cell<7>(OUT7,OUT8,i);
        Seeds[8] = insertion_cell<8>(OUT8,OUT9,i);
        Seeds[9] = insertion_cell<9>(OUT9,OUT10,i);
```

```
        Seeds[10] = insertion_cell<10>(OUT10,OUT11,i);
        Seeds[11] = insertion_cell<11>(OUT11,OUT12,i);
        Seeds[12] = insertion_cell<12>(OUT12,OUT13,i);
        Seeds[13] = insertion_cell<13>(OUT13,OUT14,i);
        Seeds[14] = insertion_cell<14>(OUT14,OUT15,i);
        Seeds[15] = insertion_cell<15>(OUT15,OUT16,i);
    }

    for (int j = 0; j < NPHOTON; j++) {
        PFNeutralObj PhotonCand;
#pragma HLS data_pack variable=PhotonCand

        l1pf_pattern_unpack<1,0>(&data[NTRACK+j],
                                 &PhotonCand);
        Photons[i].write(PhotonCand);
    }

    for (int j = 0; j < NSELCALO; j++) {
        PFNeutralObj NeutralCand;
#pragma HLS data_pack variable=NeutralCand

        l1pf_pattern_unpack<1,0>(&data[NTRACK+NPHOTON+j],
                                           &NeutralCand);
        Neutrals[i].write(NeutralCand);
    }
}

}

template <int number>
PFSeedObj insertion_cell(PFSeedObj IN[4], PFSeedObj OUT[4],
   int i){
#pragma HLS ARRAY_PARTITION variable=IN complete
#pragma HLS ARRAY_PARTITION variable=OUT complete
    static PFSeedObj CURR_REG = {0};

    int idx = 3;
    PFSeedObj max = IN[0];
    PFSeedObj temp = CURR_REG;

    for (int j = 0; j < 4; j++){
```

```
#pragma HLS UNROLL
        OUT[j] = IN[j];
        if (j == 0 && CURR_REG.hwPt >= IN[0].hwPt){
            idx = 0;
            max = CURR_REG; temp = IN[0];
        }
        if (j != 3 && CURR_REG.hwPt < IN[j].hwPt &&
                    CURR_REG.hwPt >= IN[j + 1].hwPt){
            idx = j;
            max = IN[0]; temp = CURR_REG;
        }
    }

    if (i != 35) CURR_REG = max;
    else clear(CURR_REG);
    shift_array(OUT,idx);
    OUT[idx] = temp;
    return max;
}


void shift_array(PFSeedObj Buffer[4], int last){
#pragma HLS inline
#pragma HLS ARRAY_PARTITION variable=Buffer complete
    for (int i = 0; i < 4; i++){
#pragma HLS unroll
        if (i < last){
            Buffer[i] = Buffer[i + 1];
        }
    }
}
```

# Appendix 4 Step 1 v.2

```
void algo_kbfi_step1(
            hls::stream<ap_uint<STREAM_SIZE>> &stream_in,
            hls::stream<PFChargedObj> Tracks[NREGIONS],
            hls::stream<PFNeutralObj> Photons[NREGIONS],
            hls::stream<PFNeutralObj> Neutrals[NREGIONS],
            PFSeedObj Seeds[NSEED]) {
#pragma HLS ARRAY_PARTITION variable=Seeds complete
#pragma HLS data_pack variable=Tracks
#pragma HLS data_pack variable=Photons
#pragma HLS data_pack variable=Neutrals

#pragma HLS PIPELINE II=36

for (int i = 0; i < NREGIONS; i++){
#pragma HLS PIPELINE
    // Read next region
    ap_uint<STREAM_SIZE> buf;
    stream_in.read(buf);

    // Extract relevant data
    ap_uint<PACKING_DATA_SIZE> data[NCHANN_IN];
#pragma HLS ARRAY_PARTITION variable=data complete
    for (int j = 0; j < NCHANN_IN; ++j) {
#pragma HLS UNROLL
        data[j] = buf(PACKING_DATA_SIZE*(j+1)-1,
                                PACKING_DATA_SIZE*j);
    }

    PFSeedObj Buffer[4], OUT1[4], OUT2[4], OUT3[4], OUT4[4];
    PFSeedObj OUT5[4], OUT6[4], OUT7[4], OUT8[4];
    PFSeedObj SOUT1[2], SOUT2[2], SOUT3[2], SOUT4[2];
    PFSeedObj SOUT5[2], SOUT6[2], SOUT7[2], SOUT8[2];

    // Extract tracks/seeds
    for (int j = 0; j < NTRACK; j++) {
        PFChargedObj TrackCand;
#pragma HLS data_pack variable=TrackCand

        l1pf_pattern_unpack<1,0>(&data[j], &TrackCand);
```

```
        Tracks[i].write(TrackCand);

        if (j < NSEEDNEW) {
            // Fetch first seeds and add as seed candidates
            Buffer[j].hwRegion = i;
            Buffer[j].hwPt = TrackCand.hwPt;
            Buffer[j].hwEta = TrackCand.hwEta;
            Buffer[j].hwPhi = TrackCand.hwPhi;
            Buffer[j].hwId = TrackCand.hwId;
            Buffer[j].hwZ0 = TrackCand.hwZ0;
        }
    }
    insertion_cell<0>(Buffer,OUT1,SOUT1,i);
    insertion_cell<2>(OUT1,OUT2,SOUT2,i);
    insertion_cell<4>(OUT2,OUT3,SOUT3,i);
    insertion_cell<6>(OUT3,OUT4,SOUT4,i);
    insertion_cell<8>(OUT4,OUT5,SOUT5,i);
    insertion_cell<10>(OUT5,OUT6,SOUT6,i);
    insertion_cell<12>(OUT6,OUT7,SOUT7,i);
    insertion_cell<14>(OUT7,OUT8,SOUT8,i);

    if (i == 35){
        Seeds[0] = SOUT1[0]; Seeds[1] = SOUT1[1];
        Seeds[2] = SOUT2[0]; Seeds[3] = SOUT2[1];
        Seeds[4] = SOUT3[0]; Seeds[5] = SOUT3[1];
        Seeds[6] = SOUT4[0]; Seeds[7] = SOUT4[1];
        Seeds[8] = SOUT5[0]; Seeds[9] = SOUT5[1];
        Seeds[10] = SOUT6[0]; Seeds[11] = SOUT6[1];
        Seeds[12] = SOUT7[0]; Seeds[13] = SOUT7[1];
        Seeds[14] = SOUT8[0]; Seeds[15] = SOUT8[1];
    }

    for (int j = 0; j < NPHOTON; j++) {
        PFNeutralObj PhotonCand;
#pragma HLS data_pack variable=PhotonCand

        l1pf_pattern_unpack<1,0>(&data[NTRACK+j],
                                 &PhotonCand);
        Photons[i].write(PhotonCand);
    }
```

```
    for (int j = 0; j < NSELCALO; j++) {
        PFNeutralObj NeutralCand;
#pragma HLS data_pack variable=NeutralCand

        l1pf_pattern_unpack<1,0>(&data[NTRACK+NPHOTON+j],
                                                &NeutralCand);
        Neutrals[i].write(NeutralCand);
    }
}

}

template <int number>
void insertion_cell(PFSeedObj IN[4], PFSeedObj OUT[4],
                    PFSeedObj SEEDS[2], int i) {
#pragma HLS ARRAY_PARTITION variable=SEEDS complete
#pragma HLS ARRAY_PARTITION variable=IN complete
#pragma HLS ARRAY_PARTITION variable=OUT complete
    static PFSeedObj CURR_REG_0 = {0};
    static PFSeedObj CURR_REG_1 = {0};
#pragma HLS data_pack variable=CURR_REG_0
#pragma HLS data_pack variable=CURR_REG_1

    PFSeedObj out1, out2, out3, out4, out5, out6;
#pragma HLS data_pack variable=out1
#pragma HLS data_pack variable=out2
#pragma HLS data_pack variable=out3
#pragma HLS data_pack variable=out4
#pragma HLS data_pack variable=out5
#pragma HLS data_pack variable=out6

    if (IN[0].hwPt >= CURR_REG_0.hwPt)
        out1 = IN[0];
    else out1 = CURR_REG_0;

    if (IN[1].hwPt >= CURR_REG_0.hwPt)
        out2 = IN[1];
    else if (IN[0].hwPt < CURR_REG_0.hwPt && IN[0].hwPt >=
       CURR_REG_1.hwPt)
        out2 = IN[0];
    else if (IN[0].hwPt < CURR_REG_1.hwPt)
```

```
    out2 = CURR_REG_1;
else out2 = CURR_REG_0;

if (IN[2].hwPt >= CURR_REG_0.hwPt)
    out3 = IN[2];
else if (IN[1].hwPt >= CURR_REG_1.hwPt && IN[1].hwPt <
   CURR_REG_0.hwPt)
    out3 = IN[1];
else if (IN[0].hwPt < CURR_REG_1.hwPt)
    out3 = IN[0];
else if (IN[1].hwPt >= CURR_REG_0.hwPt && IN[2].hwPt <
   CURR_REG_0.hwPt)
    out3 = CURR_REG_0;
else out3 = CURR_REG_1;

if (IN[3].hwPt >= CURR_REG_0.hwPt)
    out4 = IN[3];
else if (IN[2].hwPt >= CURR_REG_1.hwPt && IN[2].hwPt <
   CURR_REG_0.hwPt)
    out4 = IN[2];
else if (IN[1].hwPt < CURR_REG_1.hwPt)
    out4 = IN[1];
else if (IN[2].hwPt >= CURR_REG_0.hwPt && IN[3].hwPt <
   CURR_REG_0.hwPt)
    out4 = CURR_REG_0;
else out4 = CURR_REG_1;

if (IN[3].hwPt >= CURR_REG_0.hwPt)
    out5 = CURR_REG_0;
else if (IN[3].hwPt >= CURR_REG_1.hwPt)
    out5 = IN[3];
else if (IN[2].hwPt < CURR_REG_1.hwPt)
    out5 = IN[2];
else out5 = CURR_REG_1;

if (IN[3].hwPt >= CURR_REG_1.hwPt)
    out6 = CURR_REG_1;
else out6 = IN[3];

// Reset the registers
if (i == 35) {
```

```
        SEEDS[0] = out1; SEEDS[1] = out2;
        clear(CURR_REG_0); clear(CURR_REG_1);
    }
    else {
        CURR_REG_0 = out1; CURR_REG_1 = out2;
    }
    OUT[0] = out3; OUT[1] = out4;
    OUT[2] = out5; OUT[3] = out6;
}
```