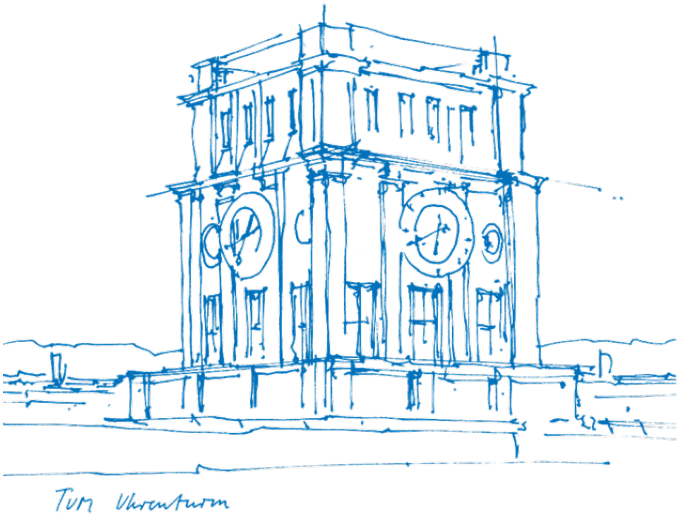
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor thesis

# Implementation of Merge Sort on SoC FPGAs

Nour Hadj Fredj



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor thesis

# Implementation of Merge Sort on SoC FPGAs

| | |
|---|---|
| Author: | Nour Hadj Fredj |
| Supervisor: | Dirk Stober |
| Examiner: | Prof. Martin Schulz |
| Submission Date: | 21 Aug 2025 |

I confirm that this bachelor thesis is my own work and I have documented all sources and material used.

Heilbronn, 21 Aug 2025 Nour Hadj Fredj

# Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dirk Stober, for his invaluable guidance, support, and patience throughout this research project. His expertise and constructive feedback were instrumental in shaping this work and helping me navigate the challenges I encountered.

# Abstract

This thesis presents an efficient implementation of a merge sort algorithm on a system on chip (SoC) Field Programmable Gate Array (FPGA), specifically targeting the AMD ZCU102 evaluation kit. The design utilizes High-Level Synthesis (HLS) tools to develop a novel non-recursive 16-leaf merge tree architecture that attempts to maximize FPGA resource utilization while maintaining high throughput through dynamic stream population and iterative processing techniques, enabling the sorting of large datasets. The implementation features a four-level parallel merge tree with 16 input streams, utilizing custom load and store units for dynamic memory management. Performance evaluation demonstrates significant improvements over CPU-based implementations, achieving speedup factors ranging from 1.39× for small datasets to 9.08× for large datasets when compared to CPU quicksort implementations. The results validate the potential of FPGA-based acceleration for sorting algorithms while highlighting both opportunities and challenges in hardware acceleration.

# Contents

# Nomenclature

## Symbols

| Symbol | Name | Description |
|---|---|---|
| II | Initiation Interval | Number of clock cycles before the function can accept new input data |

## Abbreviations

| Abbreviation | Description |
|---|---|
| GPU | Graphical Processing Unit |
| CPU | Central Processing Unit |
| SoC | System on a Chip |
| FPGA | Field-Programmable Gate Array |
| HLS | High Level Synthesis |
| DDR | Double Data Rate |
| DRAM | Dynamic Random Access Memory |
| ASIC | Application-Specific Integrated Circuits |
| RTL | Register Transfer Level |
| CCI | Coherent Central Interconnect |
| SMMU | System Memory Management Unit |

# 1 Introduction

## 1.1 Overview

In the age of big data and computationally intensive applications, the demand for high-performance computing solutions has reached unprecedented levels[1]. Traditional CPU architectures, while versatile and general-purpose, often fall short in delivering the specialized performance required for specific computational kernels[2]. This limitation has driven the computing industry toward heterogeneous computing architectures that leverage specialized hardware accelerators to achieve optimal performance for domain-specific applications.

Field-Programmable Gate Arrays (FPGAs) have emerged as a compelling solution for accelerating computationally intensive algorithms due to their unique combination of hardware-level performance and post-manufacturing reconfigurability[3]. Unlike Application-Specific Integrated Circuits (ASICs), which offer high performance but lack flexibility, or Graphics Processing Units (GPUs), which excel at parallel workloads but are constrained by their fixed architecture[4], FPGAs provide a middle ground that allows developers to create custom hardware implementations tailored to specific algorithmic requirements.

The importance of designing kernels closer to hardware lies in the fundamental trade-offs between generality and efficiency in computing systems. CPUs are designed to handle a wide variety of computational tasks efficiently, but this generality comes at the cost of specialized performance. When algorithms are implemented closer to hardware, several key advantages emerge, like spatial parallelism, custom memory hierarchies, and deterministic performance[5], which we will investigate in more detail through a concrete example by designing a merge sort implementation.

## 1.2 Contributions

In this thesis, we will go through the process of designing algorithms on an FPGA with a comprehensive investigation of the design process, and a specific focus on merge sort optimization for the ZCU102 evaluation platform.

We develop a 16-leaf merge tree architecture that maximizes the utilization of FPGA resources while maintaining high throughput. The design employs a non-recursive, tree-based approach that circumvents the limitations of High-Level Synthesis (HLS) tools regarding recursive implementations.

We present a comprehensive approach to scaling merge sort implementations beyond the constraints imposed by limited on-chip memory resources. Our methodology employs dynamic stream population and iterative processing techniques that enable the sorting of large datasets, which results in a design that exceeds some of the best sorting implementations on CPU.

## 1.3 Outline

The second chapter will present the necessary background information, definitions, and setup for both hardware and software, and provide an overview of the necessary information on the board as well as the tools used for the implementation. We also provide an explanation of the choice of algorithm in detail.

The main implementation is described and explained in the third section, where difficulties related to designing algorithms for an FPGA, as well as solutions, are presented, which is followed by the fourth section, where we go into detail about the steps taken to integrate our design on the board.

The fifth section is where we present the result of our implementation, and we compare it to a quicksort implementation on the CPU. We also explain any discrepancies in the result, postulate hypotheses, and finally, in the last section, we discuss possible improvements and additional implementations before concluding our work.

# 2 Background

## 2.1 Hardware Architecture and Platform

In this section, we will discuss the setup related to the hardware, which is a fundamental part of our work, since the whole idea revolves around implementing an algorithm on an FPGA board. Thus, it is necessary to understand the tools we will be working with.

### 2.1.1 FPGAs

It is best to start by discussing what FPGAs are. FPGAs, or Field-Programmable Gate Arrays, are programmable and configurable integrated circuits that, after manufacturing, can be reprogrammed multiple times, making them very versatile[6], allowing us to optimize them for specific kernels.

#### Differences between FPGAs and CPUs

Since one of our objectives is to note the differences between the implementation of our algorithm on both FPGA and CPU, it is necessary to understand the differences between them. FPGAs, as we discussed in previous sections, are programmable and configurable, and they excel at spatial parallelism, where multiple operations can be performed simultaneously across different regions of the chip[7]. CPUs rely primarily on temporal parallelism through techniques like pipelining and superscalar execution. Also, FPGAs can implement custom memory hierarchies and access patterns optimized for specific algorithms, while CPUs are constrained by cache-based memory systems designed for general-purpose applications.

#### Timing closure

Timing closure is the process of ensuring that signals in a digital circuit meet all of its timing requirements, such as timing constraints, and the design successfully operates at the target clock frequency without timing violations, making it extremely relevant to FPGAs. At its core it means that all signal paths must satisfy two key timing constraints which are setup time requirements, where data must arrive at a flip-flop and be stable for a minimum time before the clock edge, and hold time requirements where data must remain stable for a minimum time after the clock edge to ensure reliable capture, thus if signals arrive too late the design may fail timing. Not meeting the time constraints can cause setup violations and data Corruption since the flip-flop may capture incorrect or metastable data as well as race conditions if hold Violations occur[6]. Timing closure also involves the concept of critical path, which is the longest combinational logic path between two sequential elements, which determines your maximum achievable clock frequency, and if this path takes too long, it will cause performance bottlenecks and limit our clock frequency[8], regardless of how fast the rest of your design could theoretically run.

### 2.1.2 ZCU102

The ZCU102 Evaluation Kit is the board we will be using for our work and on which we will be implementing the design. The kit features an AMD Zynq UltraScale+ Multi-core Processor System-on-Chip (MPSoC) with the Processing System (PS) block containing a quad-core Arm Cortex-A53 application processing unit (APU), dual-core Cortex-R5F real-time processing unit (RPU), and a Mali-400 MP2 graphics processing unit (GPU) based on 16nm FinFET+ programmable logic fabric by AMD. In this section, we will go into more detail about the main components of the board as seen in Figure 2.1[9].

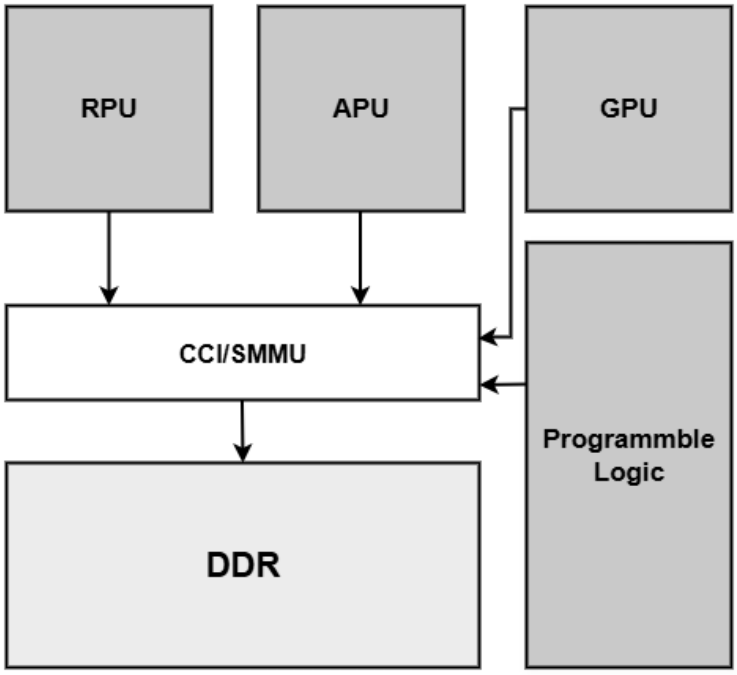



Figure 2.1: Zynq MPSOC Simplified Top-Level Block Diagram

**Programmable Logic(PL):**

This is the FPGA fabric of the board, and it is composed of Configurable Logic Blocks (CLBs), which are the fundamental building blocks that implement the actual computational logic of our design. They contain lookup tables (LUTs) that can implement any 6-input Boolean function, flip-flops for storing state and creating sequential logic, and carry chains for efficient arithmetic operations like addition and multiplication. It is also composed of Block RAM (BRAM) with 36Kb blocks and an UltraRAM (URAM) 288Kb block, both of them being on-chip memory[9].

**Interconnect Infrastructure:**

Data moves between PS and PL via Multiple AXI interfaces and even Network-on-Chip (NoC) in newer devices. These connections can be seen in figure 2.2, and we will dive into more detail about them in the next section.

### 2.1.3 Interconnect[10]

It is of great importance to go a bit more into detail about how we connect to DDR memory, or Double Data Rate memory, which is the main memory of our board, since one of our objectives is to optimize our design. These connections are done through AXI interfaces as mentioned in the previous section. However, the choice of the port can make a big difference, hence we will discuss a few of them in this section.

- **High-performance master (HPM):** They are AXI interface ports either from the Full-power domain (FPD) into the PL (HPM0,1_FPD) or from the Low-power domain (LPD) into the PL (HPM0_LPD), allowing 32/64/128 bit data width.
- **High-performance coherent (HPC):** They are 128-bit interface ports passing through the CCI and SMMU, providing one-way (I/O) coherency with no L2 cache allocation. They allow for high throughput and multiple interfaces; however, this impacts the CCI and degrades APU and other masters accessing memory via the CCI.
- **High-performance non-coherent (LPD):** this AXI port path from PL to LPD, which allows access between the PL and RPU even when the FPD is powered down. They offer between 32 and 128-bit data width, and it has the benefit of being the fastest, low-latency path to the OCM and TCM.
- **Accelerator Coherency Port (ACP):** They are 128-bit AXI interface ports which are I/O coherent with CCI with L2 cache allocation. They have the lowest latency to L2 cache, but they are Limited to 16B and 64B transactions, impacting PL DMA design.
- **AXI coherency extensions (ACE):** This port is a two-way coherent path between memory in PL and CCI. It allows for 128-bit data width, making it optional cache coherency. However, burst length is limited to 64B when CCI snoops the PL master.

#### High Performance ports(HP)[10]

We mentioned a few of the ports that we can use in our board. however, we dedicated an entire section to HP ports because these will be the ones we will use, of which there are 4: HP0 until HP3, with HP1 and HP2 having a switch before memory control. First of all, they have a direct connection to the DDR memory controller, bypassing the PS CPU and caches. This can be clearly seen in the figure 2.2. Secondly, the ports allow us up to 128-bit data width and have support for burst transfers with a peak clock frequency of 333MHz and a peak bandwidth of 5.328 GB/s per. Finally, by having 4 of them, we can have multiple ports operating simultaneously.

## 2.2 High-Level Synthesis

In this section, we will discuss the software environment and the tools used to design our algorithm, and understand the advantages they offer and the functionalities they provide to help us achieve our goal.

High-level synthesis (HLS) represents a major paradigm shift in FPGA design by allowing developers to describe hardware functionality using more human-readable and high-level languages like C or C++ instead of regular hardware descriptive languages like Verilog or

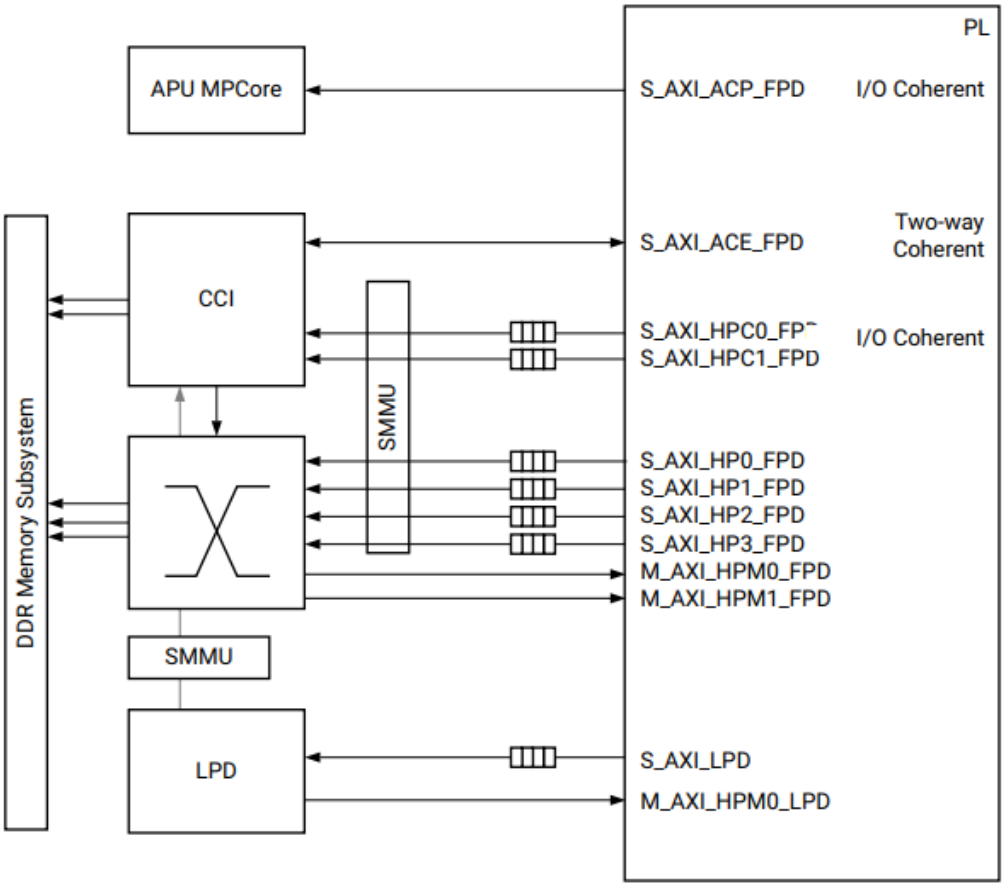



Figure 2.2: PS-PL AXI Interface Datapaths[10]

VHDL[11]. This approach reduces the design complexity for FPGA implementations significantly while making FPGA development accessible to software engineers without extensive hardware design experience, even though fundamental hardware knowledge is still necessary. Another advantage of HLS is that it gives the developer more control over optimization of the design through the use of directives, which we will explain in more detail in section 2.2.2. The high-level synthesis process consists of multiple stages, which include lexical processing, algorithm optimization, scheduling, and more[12]. These transformations are guided by user-specified constraints and optimization directives, allowing fine-grained control over the resulting hardware implementation.

### 2.2.1 Vitis HLS[13]

Vitis HLS is a tool designed by AMD that allows users to easily create complex FPGA algorithms by synthesizing a C/C++ function into Register Transfer Level (RTL), and it offers many features that support hardware design, which include:

**C simulation:**

After writing out C/C++, the C simulation functionality allows us to test our implementation by compiling and executing it using a standard software compiler through a self-checking test bench, which validates that the results from the function to be synthesized are correct. This

requires configuring the HLS component to support simulation, and it also only works with a C++ compiler.

**C synthesis:**

This is the main HLS functionality, which translates our C/C++ code into RTL as we discussed previously in section 2.2. This necessitates defining a top-level function representing the main function to be synthesized. After running the synthesis functionality, the result would be in both Verilog and VHDL, and we can see how the code was translated into multiple RTL files. This step also provides a summary that contains time estimation, performance, and resource estimates, as well as additional information on interfaces, programs, etc. This will be very useful when trying to analyze our results later.

**C/RTL cosimulation:**

Similar to C simulation, this functionality allows us to test our synthesized RTL code using the same test bench we utilized previously to test the C++ code. This allows us to make sure that our RTL code still performs as expected while also accounting for clock cycles and deadlocks that can affect functionality, which C simulation does not account for.

**Package:**

This step creates a reusable Intellectual Property (IP)/Kernel core from the synthesized RTL design. This process generates all necessary interface files, documentation, and metadata required for integration into larger FPGA designs or the Vivado IP catalog. The packaged IP/Kernel includes AXI interface specifications, timing constraints, and resource utilization reports, and the final output is, in our case, a Xilinx object (XO) file.

**Implementation:**

This phase integrates the HLS-generated IP/Kernel with the target platform through Vivado Design Suite, performing place-and-route operations to generate the final bitstream. This step validates timing closure, optimizes resource placement, and produces the executable FPGA configuration file (.bit) along with detailed implementation reports, including actual resource utilization and achieved clock frequencies.

To address critical path delays concerning timing closure, Vitis HLS tools also employ several optimization strategies, including automatic pipelining, where long combinational paths are broken into smaller stages separated by registers, and resource selection, where faster but potentially larger hardware operators are chosen for timing-critical operations.

### 2.2.2 Pragma Directives[13]:

Without Pragmas, our code would run on a CPU, but it cannot be synthesized into Register Transfer Level languages. It is these pragmas that help the tool produce a design that not only meets our requirements but also is optimized. Here we will present a list of a few pragmas that we will use multiple times in our implementation, as well as their effect:

- **HLS PIPELINE II=1:** One of the most important pragma which we use multiple times in our code which reduces the initiation interval (II) for a function or loop to a specified

value and thus for an II of 1 it processes a new input every clock cycle allowing it to be implemented in a concurrent manner. Vitis HLS then generates a design that meets this constraint if possible.

- **HLS INLINE OFF:** Function inlining is an optimization technique where the body of a function is copied directly into the code of its callers, eliminating the overhead of function calls. The following directive prevents automatic function inlining, maintaining each function as a distinct hardware resource, stopping it from being copied wherever it's used, and thus improving performance by reducing resources.
- **HLS UNROLL:** Allows us to unroll loop iterations into parallel operations by creating multiple copies of the loop body in the RTL design. Fully unrolling the loop creates a copy of the loop body in the RTL for each loop iteration, so the entire loop can be run concurrently.
- **HLS DATAFLOW:** This pragma enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation and increasing the overall throughput of the design.
- **Memory Interface Pragmas:** These directive allows us to control how the HLS module interacts with external or on-chip memory and define memory interfaces with more detail, including Streams, AXI interfaces, etc. It specifies how RTL ports are created from the function arguments during interface synthesis, and we use it to assign specific depth to our streams or define memory access via AXI. As an example, we have this directive **"#pragma HLS STREAM variable=A_streams depth=2"** which defines an A stream with a depth of 2.

### 2.2.3 Recursion:

Even though HLS is a very powerful tool, it comes with certain limitations that we need to keep in mind to be able to successfully synthesize our C++ code into RTL. HLS does not allow recursion for the simple reason that high-level code gets synthesized into hardware circuits, and recursion relies on dynamic memory allocation, which is not easily translated into a fixed hardware structure. This brings a challenge since many kernels and algorithms are implemented recursively, and thus, we have to resort to an iterative design or similar workarounds.

## 2.3 Algorithm

We discussed in the introduction that we would implement a sorting algorithm, but we never discussed our choice of algorithm in detail. We have a wide array of kernels to choose from, like matrix multiplication, graph algorithms, etc. However, we decided to choose a sorting algorithm since sorting is one of the most famous kernels ever and has many different implementations to choose from.

### 2.3.1 Comparing algorithms

Out of the multitude of sorting algorithms, we decided to compare the most prominent choices, which came down to four possibilities:

- **Merge Sort:** It has highly parallelizable merge operations with predictable memory access patterns, stable sorting behavior, and O(n log n) guaranteed complexity. Making it a very good choice to implement on an FPGA, thanks to its regular structure, perfect for pipelining and parallel implementation[14].
- **Quick Sort:** Even though it is highly optimized, especially for CPU, it suffers from irregular memory access patterns with a worst-case $O(n^2)$ complexity and difficulty in parallel implementation due to data-dependent partitioning. This becomes a major disadvantage when trying to integrate it on an FPGA due to the unpredictable control flow, which complicates hardware implementation and timing closure[15].
- **Heap Sort:** This is an efficient and very simple implementation and a more favorable worst-case $O(n \log n)$ whoever it has the same disadvantages as quick sort with irregular memory access patterns, limited parallelization opportunities as well as tree traversal operations being difficult to pipeline efficiently and difficult to implement on FPGA[16].
- **Bitonic Sort:** This sorting algorithm has the advantage of having a highly regular structure, excellent parallelization potential, and deterministic execution time, with the only disadvantages being its higher complexity than merge sort for large datasets and being limited to only power-of-2 input sizes[17].

In the end we chose the merge sort algorithm as the kernel to design and implement on our FPGA for the aforementioned reasons.

### 2.3.2 Merge Sort

Merge sort is a very popular general-purpose comparison-based sorting algorithm that uses a divide-and-conquer approach, invented by John von Neumann in 1945[16]. In its most basic form, merge sort consists of 2 main steps: dividing an unsorted list into n sub-lists, each containing one element, then repeatedly merging these sub-lists to produce new sorted sub-lists until there is only one sorted sub-list remaining, as we can see in diagram 2.3. It has a time complexity of $O(n \log n)$ in all cases and a space complexity of $O(n)$[14]. It is also considered a stable sorting algorithm because it maintains the relative order of equal elements.

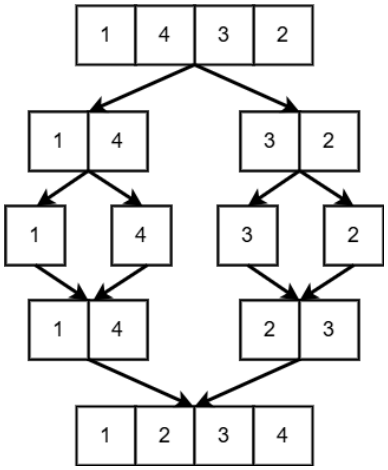



Figure 2.3: merge sort example

This approach is usually done recursively, where we call the function each time with a smaller sub-list until we end up with a list of 1 element. From then on, we start merging back the sub-lists while sorting.

**Merge Sort as tree:**

As we already mentioned, merge sort is usually implemented recursively but since HLS does not allow for a recursive implementation, we decided to implement a merge sort tree which fulfills the same goal using a non-recursive approach by designing a N-way merge tree which takes N sorted of sized S and produce 1 sorted array of size $N \cdot S$. This is done through a perfect binary tree where each node is composed of a 2-leaf sorter to form a 16-leaf sorter with 16 leaf nodes, thus we read 16 sorted arrays of size $S$ and output a sorted array of size $S \cdot 16$, with each node of our tree working in parallel to compute the full sort. Choosing a 16-leaf sorter rather than a 2-leaf sorter reduces the number of passes needed to sort the array considerably, while also being not too complex to implement, and if we were to increase the number of leaves beyond that, the resource consumption could become too great to the point where it won't benefit us. We also need to consider that our sorting approach is done through a windowed design, where for a given window, the sorter performs 16 times that window, and for arrays larger than 16, we would have run multiple iterations to sort each sub-array of size Window$\cdot$ 16. Thus, after one pass, our unsorted array, starting with a window of 2, i.e., 2 sub-arrays of size one, which are the smallest sorted arrays, the output would contain sorted sub-arrays of 16, which is the benefit of using a 16-leaf sorter. Thus for an array of size $S$, we would need a total of $\log_{16} S$ passes where for each pass:

$$window_1 = 2$$

$$window_n = window_{n-1} \cdot 16$$

$$Iteration_1 = \frac{Size}{16}$$

$$Iteration_n = \frac{Iteration_{n-1}}{16}$$

We will discuss this in more detail while discussing our code implementation in section ref scaling.

## 2.4 Limitations

In this section, we will briefly mention limitations to our work, mainly due to time constraints or ideas discarded in order to make the thesis feasible within the work period.

### 2.4.1 Board

To ensure a focused and thorough investigation within the scope of this thesis, we made the strategic decision to concentrate our efforts on a single FPGA platform rather than attempting to evaluate multiple boards simultaneously. The ZCU102 Evaluation Kit was selected as our target platform due to its robust feature set, comprehensive documentation, and suitability for high-performance sorting implementations.

### 2.4.2 Input Size and configuration

We decided to limit our input size for our design to $16^6$ elements, which is exactly 16777216 elements or 67.11 MB, since it is high enough to test the performance for a large input size and also not push the board to its limits, which isn't our goal. We also populate our input array by adding from the beginning the size of the array and then decreasing until the last element. This does not have any major effect on the performance since merge sort has a time complexity of $O(n \log n)$ in all cases, as we mentioned before, but it was an easy way for us to populate the array to then make sure the code is working.

# 3 Code Implementation

After discussing our choice of implementing a merge sort on the FPGA, we delve deeper into the code implementation to discuss our road map to designing our algorithm, while also mentioning the issues and difficulties encountered and their solutions.

## 3.1 Merge module

It is necessary to start our design with a basic module upon which we will build our full implementation, and we decided to start with a 2-leaf merge sorter which, as we mentioned previously, takes in 2 input of sized S with S being half our window size, and outputs 1 sorted stream of size $2 \cdot S$. It starts by initializing 2 variables with the first elements in our FIFO streams with a maximum depth of 64, then in a loop, it compares them and writes to our output stream the smallest and reads the next element to be compared. We utilize the **#pragma HLS INLINE OFF** directive, which prevents automatic function inlining, maintaining each merge unit as a distinct hardware resource. This approach allows for better resource utilization and enables parallel instantiation of multiple merge units across different tree levels, which will be used multiple times in future functions.
The core merging logic is done through this merge loop, which executes an exact number of passes, with each pass guaranteed to produce one output element. The HLS pragma **#pragma HLS PIPELINE II=1** ensures that each pass can begin every clock cycle, maximizing throughput, which will also usually be multiple times throughout our implementation. The comparison logic follows standard merge sort principles, which include:

- **Stream Exhaustion Handling:** If one stream is exhausted (!left_valid or !right_valid), the function outputs elements from the remaining active stream.
- **Comparison-Based Selection:** When both streams are active, the function compares current values and outputs the smaller element.
- **Stream Management:** After outputting an element, the function attempts to read the next element from the corresponding stream, updating counters and validity flags.

This 2-leaf sorter is then implemented as a parallel merge tree architecture, which was specifically designed to sort 16 input streams. The merge tree follows a bottom-up approach with four distinct levels, where each level performs parallel merge operations on increasingly larger sorted subsequences. In the first stage we have 8 2-leaf sorters working in parallel through the **#pragma HLS unroll** directive, and outputting the sorted stream which is fed into the next stage made up of 4 2-leaf sorter until the last stage where one 2-leaf sorter output the full sorted stream, thus we need in total 16 input streams and 14 intermediary streams which link each stage to the next. This is done through a second function, which orchestrates the complete 16-leaf sorting process by instantiating and connecting the four-level merge tree

hierarchy. This function demonstrates how the individual merge units are composed into a parallel sorting network.

## 3.2 Scaling up the input size

With the previously discussed implementation, scaling up the design to sort a larger dataset becomes more challenging since we cannot add more levels to the merge tree simply because the resource consumption is so important that running a C simulation might be achievable, however, running C synthesis would not be possible, and testing this approach makes the process run indefinitely without finishing.
It was thus necessary to find another approach to scale up our code and which will be our topic of discussion in this section. The main limitation we have with our current implementation is the small stream depth, where at most it reaches 64 elements. For this reason, we have to find a way to use a sorter with a limited input and output stream, and figure out how to populate the input stream dynamically while the sorting units are running.

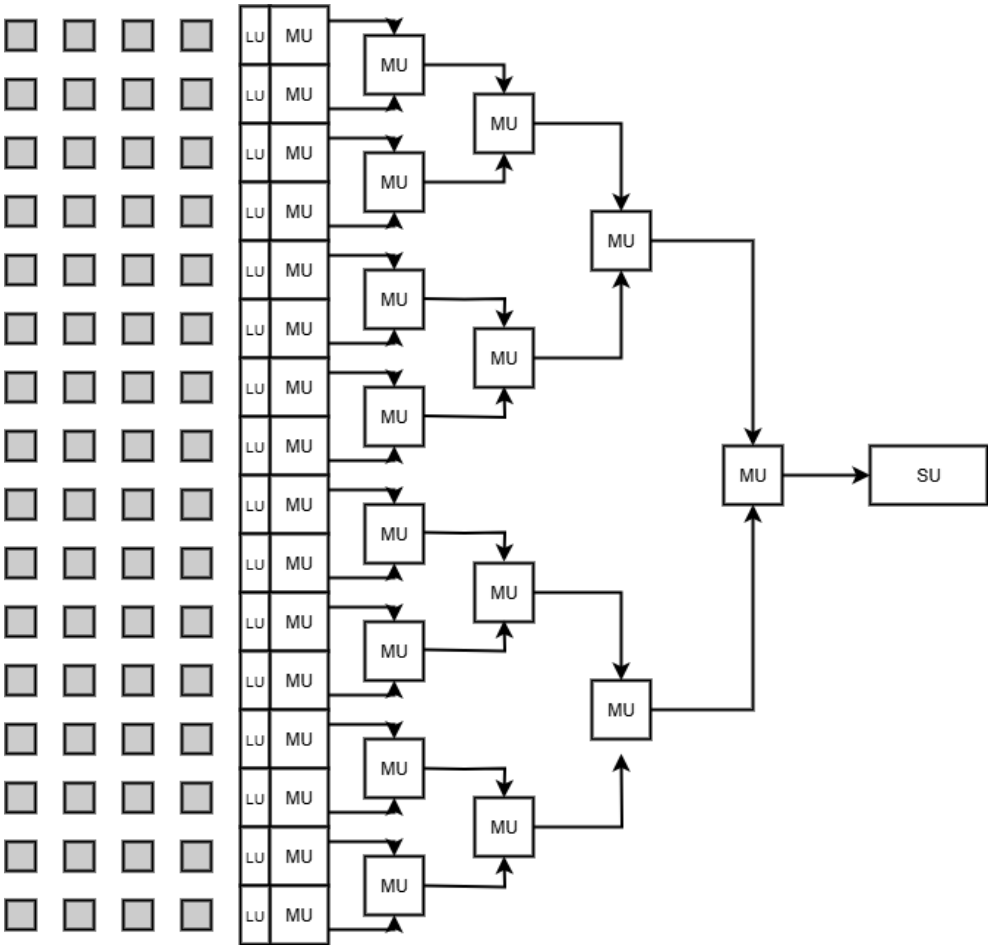



Figure 3.1: Merge Sorter Diagram

We decided to use an intuitive method, which we will explain with the use of the figure 3.1. We already mentioned the idea briefly while explaining the merge sort tree in subsection 2.3.2. For arrays of a small enough size, we can feed the full array into our streams without having to deal with our limited input. However, for larger sizes, we need to utilize iterations where we split our array into sizes of 16 and run our 16-leaf sorter for each iteration.

As an example for an array larger than 16, the first pass will sort 1 element per iteration since that is the smallest sorted array with the total iterations being one, then for the next runs we reduce the number of iterations by a factor of 16. This also requires dynamically populating our input streams, which can be achieved with a load and store units that run simultaneously as the merge unit to populate or flush out the streams. In figure 3.1, we can see the full structure of the design, with each of the gray boxes representing the elements. Thus, for the first run, each column of the elements represents the batches to be sorted in each iteration, and for the next run, we increase the iterations accordingly.

## 3.3 Possible code solutions

After explaining the idea behind how we can utilize a sorter of a limited capacity by splitting our array into small batches and feeding them dynamically to our input stream, we will thus see a possible code implementation of this method.

We start by adding an iteration variable, which will account for those repeated runs, and an additional iteration loop for our main merge unit. To be able to achieve our objective of dynamically feeding inputs and flushing them out, we designed a Load and Store unit which works as follows:

Listing 3.1: Load unit

```
void load_unit(
    const int offset,
    const int iter,
    const int window,
    const int * dram,
    hls::stream<int>& out_stream
){
    for(int j = 0; j < iter ; j++){
        int off = j*NUM_LEAVES*window/2+ offset;
                for(int i = 0; i < window/2; i++){
                  #pragma HLS pipeline II=1
                  const auto value = dram[off + i];
                  out_stream.write(value);
                }
        }
}
```

The load unit would read from DRAM with the exact values while ensuring each iteration reads from the correct memory location without overlap. The outer loop allows us to handle the multiple iterations, while the inner loop reads $window \div 2$ elements from DRAM because each input stream only needs half the window size, since the two streams will be merged together to form the full window. Then it writes them to the output stream, which is connected to the input streams of the merge units.

Listing 3.2: Store unit

```
void store_unit(
    int size,
        hls::stream<int>& out_stream,
```

```
        int * dram_out
){

    for(int i = 0; i < size/4; i++){
        for (int j = 0; j < 4; j++) {
            const auto v = out_stream.read();
            dram_out[j + (i*4)] = v;
        }
        }
}
```

The store unit reads the output stream and writes it back to memory in chunks of four. With this structure, we reach our objective with the only limitation of having an input with a power of 16. Using these functions that we have designed, we define the main sort function composed of 3 main sections:

- **Load Stage:** We read our inputs from DRAM using our load unit with a stream connected to the merge units.
- **Merge Stage:** We call the merge tree with each merge unit sorting simultaneously and feeding into the next levels of the tree, and sorting each batch at the corresponding iteration.
- **Store Stage:** We store back the sorted elements read from the output streams of the final merge unit.

After running the code for the first time, we obtain an array with the same size yet with sub-arrays of size 16 sorted, accounting that we use a window of 2 and an iteration of $size \div 16$. For an array of size 16, the full array would be sorted, yet for any larger array, we would need to run our code more than once and change the window and iteration accordingly. The full code implementation is available under `https://github.com/Nour-HF/FPGA-MergeSort`.

# 4 Integration

In this brief section, we will discuss the steps we went through after finishing our implementation of the algorithm to integrate the design on the board.

## 4.1 Linking

After the C/C++ kernels are compiled and any RTL kernels are packaged as Xilinx object (XO) files, which contain all necessary metadata, interface specifications, and RTL code required for system integration, we run the vitis link command, which links them with the target platform to build the device binary (.xclbin) file. The linking step consists of combining multiple kernel objects, integrating kernels with the target platform, resolving interconnections, and finally generating the (.xclbin) file[18]. To run the link command, we need to specify a Config file in which we specify additional information like the platform, clock frequency, and connectivity by assigning to each interface a port on our board. We specifically target HP0 and HP1 ports to maximize concurrent memory access capabilities.

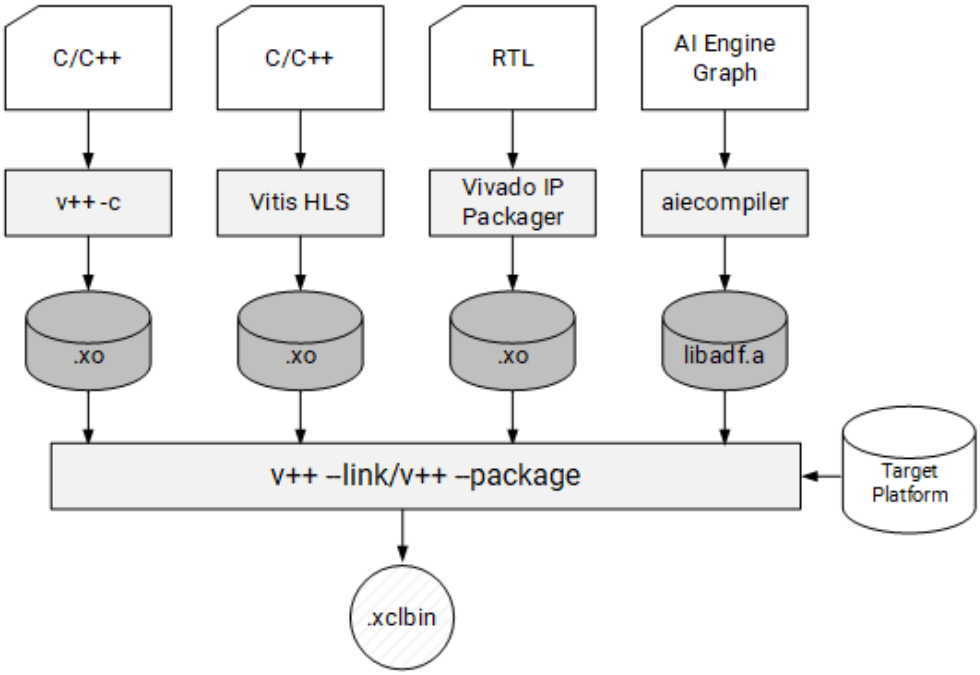



Figure 4.1: Device Build Process[18]

## 4.2 Integrating the design

### 4.2.1 Connecting to the board

In order to integrate our design on the board, we decided to choose a more remote way by connecting to the board through SSH instead of importing the design on the board directly. This way, we were able to experiment on the board without having it be physically with us.

### 4.2.2 Communicating with the FPGA

Since we cannot directly communicate with the FPGA, we had to configure it. Hence, we wrote a host code that would run our design on the FPGA. To do this, we use the Xilinx Runtime library (XRT), which is an open-source, easy-to-use software stack that facilitates management and usage of FPGA devices and mediates communication between the host processor and the FPGA accelerator[19].

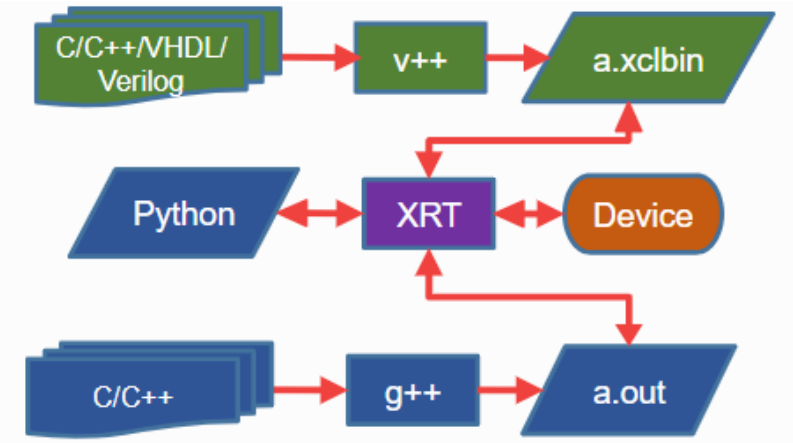



Figure 4.2: User application compilation and execution with XRT[19]

Using the library, we can write the host code where we define the device and the kernel, then choose the parameters to run. We also included the time measurement used to analyze the performance of the design.

The host code is also available under `https://github.com/Nour-HF/FPGA-MergeSort`.

# 5 Results

After going through the code implementation and how to integrate it on the board, we will go through the results we obtained from running our design on the FPGA and try to take note of the performance.

## 5.1 Time Estimation

Before executing our code and collecting results, it is useful to calculate a time estimation for running our design with the specific number of elements we intend to sort, in order to have a reference to compare to after obtaining our results. As mentioned in a previous section, Vitis HLS offers us a summary of the C synthesis report with a lot of information about the performance of our design, which we will be using extensively. To calculate the total number of cycles of our design when sorting $16^6$, we use the following formula, which works because we assume that the design acts as a global pipeline with a global II:

$$Cycles = (Total\ elements - 1) \cdot II + Latency$$

It is also worthy of notice that from this formula, the higher the number of elements, the less the latency affects the total number of cycles, thus for high values, the latency has barely any effect on the cycles, while the II matters most.
Finally, to get the total time estimation, we use the following formula:

$$Time\ Estimation\ per\ Pass = Cycles \cdot Time\ period$$

The C synthesis report provided by Vitis HLS, after synthesizing our C++ code, provides us with all the values needed. We get the following results:

Table 5.1: C synthesis report of design performance

| **Parameter** | **Value** |
|---|---|
| Latency (cycles) | 1216 |
| Total Elements | 16,777,216 |
| Initiation Interval (II) | 1 |
| Clock Period | 3.33 ns |

We can thus compute the total cycles to be:

$$Cycles = (16777216 - 1) \cdot 1 + 1216$$

$$= 16778431$$

And we get a time estimation of one pass:

$$Time\ Estimation\ per\ Pass = 16778431 \cdot 3.33ns$$

$$= 55.87ms$$

Finally, to get the total time estimation to sort the complete array of $16^6$ elements, we need to multiply by 6, since for $16^6$ elements, we need a total of 6 passes using the formula provided previously. The total time estimation is:

$$Total\ Time\ Estimation = Time\ Estimation\ per\ Pass \cdot 6$$

$$= 55.87ms \cdot 6$$

$$= 335.233ms$$

So, ideally, our algorithm, when run on the FPGA, should finish after 335.233 ms. We also remark from the C synthesis report that our design utilizes 5% of the total BRAM and 18% of the total LTUs of our board.

## 5.2 Execution time

After integrating the algorithm on the board and setting up the host code, allowing us to run the design on the FPGA, we run it multiple times with powers of 16 while keeping track of the time of execution. Since our objective is to test the performance of our algorithm and compare it to CPU, it would be unfair to compare the merge sort on both FPGA and CPU for the same reasons that also made us choose merge sort to implement, which is the advantage FPGAs have with merge sort in terms of strengths in parallel processing and pipelined operations. Thus, we decided to compare our merge sort implementation to quicksort on CPU, which is one of the fastest and most optimized sorting algorithms, and specifically designed to leverage CPU architectural features, which were included in the host code discussed in section4.2.2. After having multiple runs, we obtained the following results presented in Table 5.2.

Table 5.2: CPU Time vs Total FPGA Time and Speedup Factor

| **Elements ($16^n$)** | **CPU Time (ms)** | **FPGA Total Time (ms)** | **Speedup Factor** |
|---|---|---|---|
| $16^1$ | $6.681 \times 10^{-3}$ | $4.791 \times 10^{-3}$ | 1.39 |
| $16^2$ | $6.124 \times 10^{-2}$ | $9.751 \times 10^{-3}$ | 6.28 |
| $16^3$ | $8.259 \times 10^{-1}$ | $1.030 \times 10^{-1}$ | 8.02 |
| $16^4$ | $1.615 \times 10^{1}$ | $2.099 \times 10^{0}$ | 7.69 |
| $16^5$ | $3.125 \times 10^{2}$ | $3.599 \times 10^{1}$ | 8.68 |
| $16^6$ | $5.877 \times 10^{3}$ | $6.472 \times 10^{2}$ | 9.08 |

We also complemented Table 5.2 with the speedup factor to get a better grasp of our performance. We can also plot the execution times using a logarithmic scale for our y-axis to better show the duration for all runs, as in Figure 5.1.

Finally, we compared the execution for each pass of our algorithm, which can be found in Table 5.3, and summed up to get the full execution time for the FPGA.

## 5.3 Observations

From these results, we can make three observations concerning the design:

The First and most obvious observation is that we can notice from table 5.2 the major improvement in performance, where the design outperformed the CPU by a factor varying

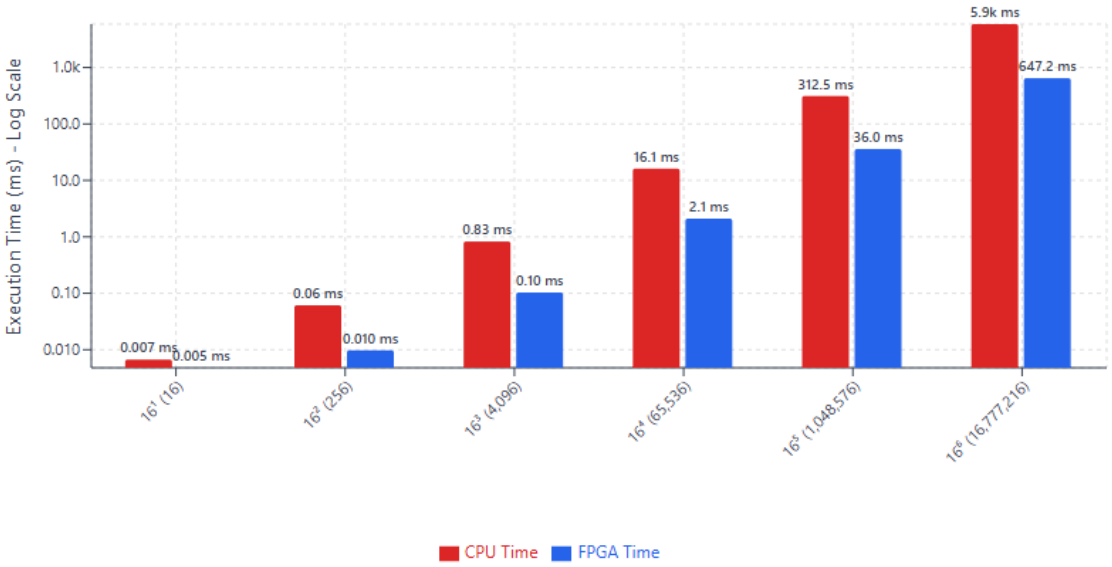



Figure 5.1: CPU Time vs Total FPGA Time

Table 5.3: FPGA Pass Times for Different Input Sizes

| **Elements** ($16^n$) | **Pass 1 (ms)** | **Pass 2 (ms)** | **Pass 3 (ms)** | **Pass 4 (ms)** | **Pass 5 (ms)** | **Pass 6 (ms)** |
|---|---|---|---|---|---|---|
| $16^1$ | $4.791 \times 10^{-3}$ | | | | | |
| $16^2$ | $6.391 \times 10^{-3}$ | $3.360 \times 10^{-3}$ | | | | |
| $16^3$ | $6.171 \times 10^{-2}$ | $2.001 \times 10^{-2}$ | $2.127 \times 10^{-2}$ | | | |
| $16^4$ | $1.224 \times 10^{0}$ | $2.890 \times 10^{-1}$ | $2.927 \times 10^{-1}$ | $2.939 \times 10^{-1}$ | | |
| $16^5$ | $1.761 \times 10^{1}$ | $4.590 \times 10^{0}$ | $4.593 \times 10^{0}$ | $4.595 \times 10^{0}$ | $4.605 \times 10^{0}$ | |
| $16^6$ | $2.801 \times 10^{2}$ | $7.341 \times 10^{1}$ | $7.341 \times 10^{1}$ | $7.342 \times 10^{1}$ | $7.342 \times 10^{1}$ | $7.343 \times 10^{1}$ |

from 1.39 for $16^1$ and 9.08 for $16^6$, meaning our main objective of designing an implementation which outperforms CPU was achieved and we are able to show the potential of implementing algorithms and kernels closer to hardware.

However, we also see from our calculation of the expected time for our last run with $16^6$ elements, our algorithm is more than twice as slow as we would expect, with exactly 311.955 ms difference. Even though the algorithm was efficient, there is still a noticeable difference that needs to be investigated.

We additionally notice from table 5.3 how the first pass is always considerably slower than the rest of the passes, and this is the case for each run we had, no matter the total number of elements, while the rest of the passes have approximately the same time.

## 5.4 Discussion of Results

Even though the performance of our design was remarkable and achieved the goal of surpassing one of the most efficient algorithms on CPU, we still need to understand why the execution time was slower than the expected time and investigate possible reasons for this discrepancy in order to improve the performance.

We also observed that the first pass was always considerably slower than the rest, and even

though this can explain why the execution was slower, we still need to understand why the first run always lagged behind the rest.

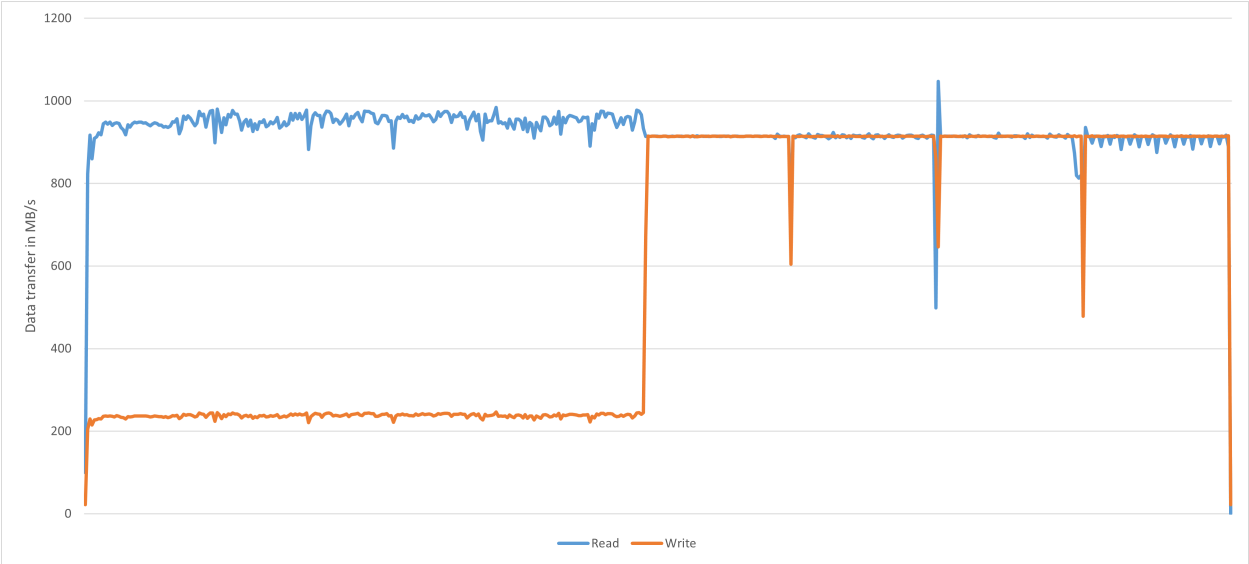



Figure 5.2: Read and write data transfer

Going back to the C synthesis reports, we notice that there is negligible slack in terms of execution and an II of 1, meaning that the problem must be in the data transfer between the FPGA and memory. Figure 5.2 can provide us with a few answers. It has been generated by monitoring the DDR controller for both read and write data transfers. In this case, we ran an example with $16^4$ elements. The first thing to notice is that the write data transfer starts off a lot slower than the read before catching up. This might explain why the first run is always considerably slower than the rest. We also see that both read and write lines sink abruptly at certain time points. This can be explained by the repeated runs of our algorithm, which matches up with our case, since for $16^4$ we need four runs, and we have three dips, including the first run, which matches up correctly. Now, even though this explains why the first run is slower than the rest, we still need to explain why the write was slower, which might explain to us why the total execution time was slower than expected.
The issue cannot be with the bandwidth limitation since our board can handle more than 19 GB/s, while our entire structure reaches at most 2 GB/s, so stalling cannot be the problem. Another hypothesis to investigate is the burst size. Going back to the C synthesis report, we can see in the M_AXI Variable Accesses section that the tool fails to widen the burst size, meaning that we are not using the full 128 bits that the HP ports offer. We already mentioned how Vitis HLS tools tries to optimize the design, and one way is to widen the burst size if this is possible. In our case, we can see that this optimization attempt has failed, and this can be explained by our design choice, where for the first iteration, we only read one value sequentially, while for the rest of the runs, we read more than one. This explained why it was unable to widen the burst size, which would limit our data transfer rate.
Finally, we should also consider that the results provided by the Vitis HLS tool are only an estimation and cannot be perfectly exact since they do not account for interface protocol and memory control overhead, and analyze the kernel in isolation, which may affect the estimation.

# 6 Future Work

## 6.1 Future Works

### 6.1.1 Possible Improvements

We have two possible improvements that can increase the efficiency of our implementation considerably. Our analysis identified memory access patterns as a primary performance bottleneck, particularly the inability of the HLS tools to optimize burst sizes for our iterative access patterns. Additionally, leveraging additional HP ports can increase aggregate memory bandwidth and reduce memory access contention. We can also try to change the structure in order to account for input of varying sizes by implementing padding and partitioning strategies to support arbitrary input sizes.

### 6.1.2 Different Implementations

There also exist alternate designs for merge sort, which have the potential of being a lot more optimized and efficient than our current implementation, which we will present.

**Bitonic merge sort[20]:**

Bitonic sort represents an alternative approach with distinct advantages for FPGA implementation, by first constructing a bitonic sequence, which is a sequence that first increases then decreases, or vice versa. This is done by splitting the input into two halves, recursively sorting one ascending and one descending, then concatenating them to form a bitonic sequence, after which we use a bitonic merger to sort it by applying a fixed pattern of compare-and-swap operations across log n stages to convert the bitonic sequence into a sorted sequence. This sorting algorithm is particularly well-suited for FPGA implementation due to its regular, parallel structure, and can achieve very high throughput once the pipeline fills.

**Adaptive merge trees (ATM)[21]:**

Based on the Bonsai framework, this design allows us to optimize the merge unit by sorting vectors instead of single elements. It is basically a binary tree of hardware mergers where each merger can process k records per cycle (k-merger), thus, instead of merging separate elements, the higher up in our merge tree, we sort vectors of elements. AMTs can be configured for any problem size without fundamental algorithmic changes, and previous implementations have shown very promising results, surpassing CPU implementations.

# 7 Conclusion

This thesis demonstrates that FPGAs provide a compelling platform for accelerating sorting algorithms. Through our research and implementation of merge sort on the ZCU102 platform, we have demonstrated that carefully designed FPGA implementations can achieve significant performance improvements over traditional CPU-based approaches, with our design reaching speedup factors ranging from 1.39× for small datasets to 9.08× for large datasets containing $16^6$ elements when compared to CPU quick sort implementations.

The success of our merge sort implementation stems from the natural alignment between the algorithm's characteristics and FPGA architectural strengths[22]. Merge sort's predictable memory access patterns, inherent parallelism, and hierarchical structure map effectively to FPGA spatial parallelism capabilities and custom memory hierarchy implementations. Our 16-leaf merge tree architecture demonstrates how algorithmic requirements can be matched to hardware constraints to achieve decent resource utilization. However, our analysis also revealed important limitations and optimization opportunities. The discrepancy between theoretical performance predictions and measured results highlights the complexity of achieving optimal memory subsystem utilization and the importance of considering implementation-specific factors that may not be captured in high-level performance models.

Furthermore, the growing importance of energy efficiency in computing systems positions FPGAs as an attractive alternative to traditional CPU and GPU implementations[23]. The ability to create custom hardware implementations optimized for specific algorithms can potentially deliver superior energy efficiency compared to general-purpose processors, making FPGAs increasingly relevant for data center and edge computing applications.

In conclusion, this thesis contributes to the understanding of FPGA-based sorting acceleration while providing a foundation for future research in hardware-accelerated algorithms. The demonstrated performance improvements, comprehensive integration framework, and detailed analysis of optimization opportunities provide valuable insights for both researchers and practitioners interested in leveraging FPGA acceleration for computational kernels. As FPGA technology continues to evolve and development tools become more sophisticated, the approaches demonstrated in this work will become increasingly accessible and practical for a broader range of applications and developers.

# List of Figures



# List of Tables

# Bibliography


[1] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters." In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. "Dark silicon and the end of multicore scaling." In: *ACM SIGARCH Computer Architecture News* 39.3 (2011), pp. 365–376.

[3] K. Compton and S. Hauck. "Reconfigurable computing: a survey of systems and software." In: *ACM Computing Surveys* 34.2 (2002), pp. 171–210.

[4] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli. "Graphics processing unit (GPU) programming strategies and trends in GPU computing." In: *Journal of Parallel and Distributed Computing* 73.1 (2013), pp. 4–13.

[5] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, et al. "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" In: *ACM SIGARCH Computer Architecture News* 45.2 (2017), pp. 5–17.

[6] S. Brown and Z. Vranesic. *Fundamentals of digital logic with VHDL design*. 3rd. McGraw-Hill, 2012.

[7] R. Tessier and W. Burleson. "Reconfigurable computing for digital signal processing: A survey." In: *Journal of VLSI signal processing systems for signal, image and video technology* 28.1 (2001), pp. 7–27.

[8] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu. "VLSI physical design: from graph partitioning to timing closure." In: *Design Automation Conference*. Springer. 2009, pp. 454–459.

[9] AMD. *Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit User Guide*. UG1182. Version v1.8. Version 1.8. Advanced Micro Devices, Inc. Apr. 2023.

[10] AMD. *Zynq UltraScale+ Device Technical Reference Manual*. UG1085. Version v2.4. Version 2.4. Advanced Micro Devices, Inc. July 2023.

[11] G. Martin and G. Smith. "High-level synthesis: Past, present, and future." In: *IEEE Design & Test of Computers* 26.4 (2009), pp. 18–25.

[12] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin. "High-level synthesis: introduction to chip and system design." In: *Springer Science & Business Media*. Springer, 1992.

[13] AMD. *Vitis High-Level Synthesis User Guide*. UG1399. Version v2023.2. Version 2023.2. Advanced Micro Devices, Inc. Oct. 2023.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. "Introduction to algorithms." In: (2009).

[15] R. Sedgewick and K. Wayne. "Algorithms." In: (2011).

[16] D. E. Knuth. *The art of computer programming, volume 3: Sorting and searching*. 2nd. Addison Wesley Longman Publishing Co., Inc., 1998.

[17] K. E. Batcher. "Sorting networks and their applications." In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. 1968, pp. 307–314.

[18] AMD. *Vitis Unified Software Platform Documentation: Application Acceleration Development*. UG1393. Version v2023.2. Version 2023.2. Advanced Micro Devices, Inc. Dec. 2023.

[19] AMD. *XRT and Vitis Platform Overview*. Version 17. Online documentation. Advanced Micro Devices, Inc. Dec. 2024. URL: https://xilinx.github.io/XRT/master/html/platforms.html.

[20] R. Chen, S. Siriyal, and V. K. Prasanna. "Energy and Memory Efficient Mapping of Bitonic Sorting on FPGA." In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 240–249.

[21] N. Samardzic, W. Qiao, V. Aggarwal, M.-C. F. Chang, and J. Cong. "Bonsai: High-Performance Adaptive Merge Tree Sorting." In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2021, pp. 282–294.

[22] A. H. Jalilvand, F. S. Banitaba, S. N. Estiri, S. Aygun, and M. H. Najafi. "Sorting it out in Hardware: A State-of-the-Art Survey." In: *arXiv preprint* (2023). also accepted / available through ACM (see publisher record). eprint: 2310.07903.

[23] J. Fowers, G. Brown, P. Cooke, and G. Stitt. "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications." In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, 2012, pp. 47–56. DOI: 10.1145/2145694.2145704.