A High Throughput Sorting Accelerator using an Intel PAC FPGA.

Bas Nijkamp

May 6, 2022

Abstract

Numerous services and products use large databases to store all kinds of data. One commonly used operation is sorting. The amount of data to be sorted is increasing rapidly [15]. Current CPU architectures are not able to scale/advance with the same rate. A solution is needed to optimise the sorting of these large databases. Previous research has shown that specialised hardware such as GPUs and configurable logic (FPGA, ASIC) can accelerate certain database operations by leveraging their natural parallelism. FPGAs and ASICs show the greatest performance gains over a commonly used server CPU (i.e. Intel Xeon). However, manufacturing an ASIC is very expensive and only profitable if it is produced in large quantities. ASICs cannot be reconfigured and therefore can only optimise non-application specific tasks. FPGAs are reconfigurable and not only have higher performance, but also more efficient energy utilisation compared to their GPP/CPU counterparts by using significantly lower clock speeds while having better performance [15].

In this thesis we describe the development of an accelerator on the *Intel Hardware Accelerator Research Program (HARP)* platform. This platform provides tools and services to build FPGA based accelerators that work closely together with a GPP/CPU to optimize certain workloads. We focus on the design and development of a sorting accelerator.

The concept of hardware based sorting is explored using sorting networks. Sorting networks are a straightforward, efficient and predictable way of implementing sorting in hardware. Using sorting networks we can maximize the parallel capabilities a FPGA has to offer since a lot of the sorting stages can be executed simultaneously. Sorting networks can be implemented in hardware with combinational, synchronous and pipelined circuits. Pipelined sorting networks provides the best throughput. Research [13] shows that the Batcher's even-odd merge sort network is most efficient in hardware resource usage and throughput.

Different concepts are being analysed and found that the best solution is to move the sorting operation completely to the FPGA. When fully executing this operation on the FPGA we reduce the maximum amount of load of the CPU and make the most use of the available FPGA resources compared to the other given concepts. By using faster local memory of the FPGA for merging, the overall throughput is increased.

Sorting networks are predictable when it comes to hardware cost. When examining the synthesised design we discover that the hardware usage of our design matches the hardware usage estimation. Comparing the sorting accelerator to similar solutions shows that although the throughput is higher, the performance of the sort accelerator is limited by the bandwidth of the communication bus between FPGA and the host CPU memory. Existing solutions face the same limitation and try to reduce this by applying compression to the data that is being transferred between the FPGA and CPU.

Finally, we conclude about the feasibility of using the *Intel Hardware Accelerator Research Program (HARP)* platform, OPAE and the remaining available tools.

List of abbreviations

AFU	Accelerator Functional Unit
ALM	Adaptive Logic Module
ASE	Accelerator Functional Unit (AFU) Simulation Environment
ASIC	Application-Specific Integrated Circuit
BBB	Basic Building Block
CCI	Core Cache Interface
CL	Cache Line
CPU	Central Processing Unit
CU	Comparator Unit
ES	Execution Steps
FIM	FPGA Interface Manager
FIU	FPGA Interface Unit
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
GPU	Graphical Processing Unit
HARP	Hardware Accelerator Research Program
HDL	Hardware Description Language
LUT	Look Up Table
MMIO	Memory Mapped I/O
MPF	Memory Properties Factory
OEM	Odd Even Merge
OEMS	Odd Even Merge Sort
OPAE	Open Programmable Acceleration Engine

PAC	Programmable Acceleration Card
PCI	Peripheral Component Interconnect
VC	Virtual Channel
VHDL	VHSIC Hardware Description Language

Contents

Chapter 1

Introduction

Many services and products use large databases to store all kinds of data. One commonly used operation is sorting. The amount of data to be sorted is increasing rapidly [15]. Current CPU architectures are not able to scale/advance with the same rate. A solution is needed to optimise the sorting of these large databases. Previous research has shown that specialised hardware such as GPUs and configurable logic (FPGA, ASIC) can accelerate certain database operations by leveraging their natural parallelism. FPGAs and ASICs show the greatest performance gains over a commonly used server CPU (i.e. Intel Xeon). However, manufacturing an ASIC is very expensive and only profitable if it is produced in large quantities. ASICs cannot be reconfigured and therefore can only optimise non-application specific tasks. FPGAs are reconfigurable and not only have higher performance, but also more efficient energy utilisation compared to their GPP/CPU counterparts by using significantly lower clock speeds while having better performance [15].

This research focuses on accelerating the sorting operation using hardware available from the *Intel Hardware Accelerator Research Program (HARP)*. The Intel HARP environment is a cluster of servers with an Intel FPGA attached. The goal of this programme is make the research opportunity into hardware-based accelerators available to a broader audience.

The objective of this thesis is to develop a sorting accelerator using hardware and tools provided by the *Intel Hardware Accelerator Research Program (HARP)*. Sub-objectives include a performance evaluation of this solution compared to existing solutions and a discussion of the convenience of using this platform.

This gives us the following research questions:

- Is it possible to develop a high performant sorting accelerator on the *Intel Hardware Accelerator Research Program (HARP)* platform?
  - How does the performance compare to existing sorting solutions?
  - What is the benefit of using an accelerator card for sorting instead of a more traditional solution?
- How feasible is it to develop a sorting accelerator using the *Intel Hardware Accelerator Research Program (HARP)*?

- Are the hardware and tools made available through the platform ready to be used in a production environment?

Chapter 2

Background Information

Before talking about the design and implementation of the sorting accelerator, there are important concepts the reader should be slightly familiar with. In this chapter a brief explanation of these concepts is given.

1 ASIC

An ASIC (*Application-Specific Integrated Circuit*) is an *Integrated Circuit* (IC) customised for a particular use, rather than intended for general-purpose use. Designers of digital ASICs often use a *Hardware Description Language* (HDL), such as Verilog or VHDL, to describe the functionality of ASICs. [2]

2 GPP/CPU

GPP sometimes referred as CPU in this document is an abbreviation for *General Purpose Processors*. These processors can be traditional consumer processors which can do a large set of tasks. However, because it supports a large set of tasks it is often not efficient because it uses a lot of energy for the same task compared to a hardware implementation. A hardware implementation used to be considered inflexible until the introduction of FPGAs.

3 FPGA

An FPGA (*Field Programmable Gate Array*) is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence the term "field-programmable". The FPGA configuration is generally specified using a *Hardware Description Language* (HDL), similar to that used for an Application-Specific Integrated Circuit (ASIC). Circuit diagrams were previously used to specify the configuration, but this is increasingly rare due to the advent of electronic design automation tools. [3]

FPGAs contain an array of programmable logic blocks, and a hierarchy of "reconfigurable interconnects" that allow the blocks to be "wired together", like many logic gates that can be inter-wired in different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory. Many FPGAs can be reprogrammed to implement different logic functions by using *Look Up Tables* (LUTs), allowing flexible reconfigurable computing as performed in computer software. [3]

Partial Reconfiguration

Partial Reconfiguration is the ability to dynamically modify blocks of logic by downloading partial bit files while the remaining logic continues to operate without interruption. Partial Reconfiguration technology allows designers to change functionality on the fly, eliminating the need to fully reconfigure and re-establish links, dramatically enhancing the flexibility that FPGAs offer. The use of Partial Reconfiguration can allow designers to move to fewer or smaller devices, reduce power, and improve system upgrade-ability. Make more efficient use of the silicon by only loading in functionality that is needed at any point in time. [8]

4 IL Academic Compute Environment provided by Intel

Intel is building a family of FPGA accelerators aimed at data centers. The family shares a common software layer, the Open Programmable Acceleration Engine (OPAE), as well as a common hardware-side Core Cache Interface (CCI-P). Intel is making a collection of systems available to researchers through IL Academic Compute Environment and the Intel Hardware Accelerator Research Program (HARP). [4]

Chapter 3

System Overview

Different platforms exist for creating FPGA-based accelerators. For this research a platform provided by Intel was used. Most concepts of the sorting accelerator can be applied to any FPGA accelerator platform. In this chapter, we discuss some specific concepts related to the provided platform. This platform is a key element for the design choices made in order to develop the sorting accelerator. We will discuss the most relevant features of this platform and some important details which led into the design of the sorting accelerator.

1 IL Academic Compute Environment

The IL Academic Compute Environment provides tools and compute nodes for academic research. One of the research clusters focuses on the development of FPGA Accelerators using a variety of Intel FPGAs. These FPGAs are mainly focused on accelerating tasks in a data center environment.

1.1 FPGA System classes

There are several system classes that can be used on the IL Academic Compute Environment [4]:

Class	Description
fpga-pac-s10	PCIe D5005 Programmable Acceleration Cards (PACs) with a Stratix 10 SX FPGA (1SX280HN2F43E2VG). OpenCL and RTL are both supported. These cards offer PCIe Gen 3 x16 and 4 channels of 8GB DDR4-2400 with ECC.
fpga-pac-s10-2	Two PCIe D5005 Programmable Acceleration Cards (PACs) with a Stratix 10 SX FPGA (1SX280HN2F43E2VG). OpenCL and RTL are both supported. The cards are not yet networked with QSFP+ ports, but could be used for projects requiring a pair of cards, communicating through system memory.
fpga-pac-a10	PCIe Programmable Acceleration Cards (PACs) with an Arria 10 GX FPGA (10AX115N2F40E2LG). Two cards are installed in each system and the 40GbE QSFP+ ports are connected to each other in order to facilitate research on networked FPGAs. OpenCL and RTL are both supported.
fpga-bdx-opae	Broadwell Xeon CPUs (E5-2600v4) with an integrated in-package Arria 10 GX1150 FPGA (10AX115U3F45E2SGE3). These systems are for workloads written in RTL.
fpga-bdx-opencl	The same Broadwell Xeon+FPGA systems as fpga-bdx-opae, but configured for use with logic written in OpenCL. Unlike later OPAE-managed systems, Broadwell uses different FPGA-side base logic for OpenCL, forcing a separation of RTL and OpenCL servers.
fpga-bdx-aal	The same Broadwell Xeon+FPGA systems as fpga-bdx-opae, but the loaded Linux kernel driver is Intel's legacy Accelerator Abstraction Layer (AAL).

1.2 Decision system class

The system class chosen for this research project is *fpga-pac-s10.* This Intel PAC FPGA is a PCI Express add-in card that contains a Stratix 10 SX FPGA. The features of this card meet the goal of this research project as this accelerator card can be added to existing systems rather than replacing compute nodes in the data center. This has the advantage that system administrators can add these cards to their existing compute nodes without changing the entire data center infrastructure. This makes the acceleration compatible with any system that has PCIe 3.0 x16 lanes. A tightly packed configuration such as *fpga-bdx-opae* could be used for special use cases where the accelerator requires more communication channels to the *General Purpose Processor* (GPP). However, this comes at the cost of portability of the acceleration solution.

2 System characteristics

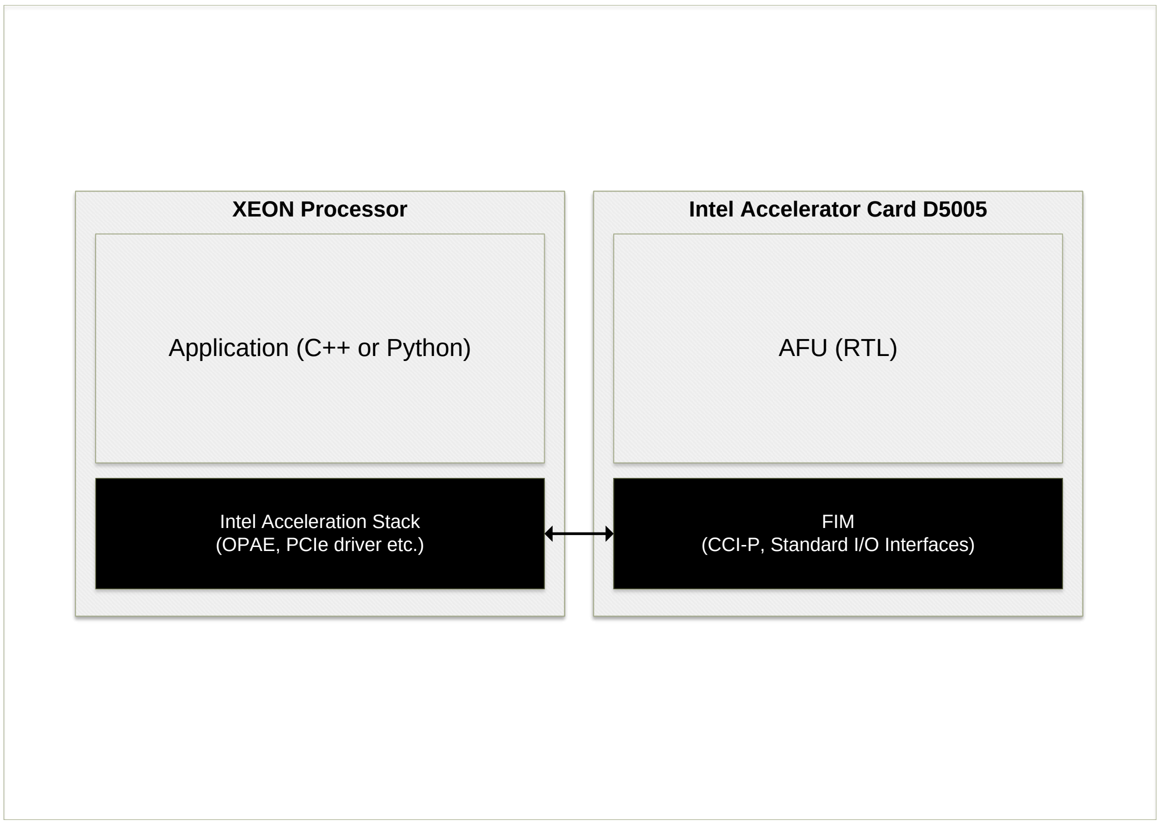
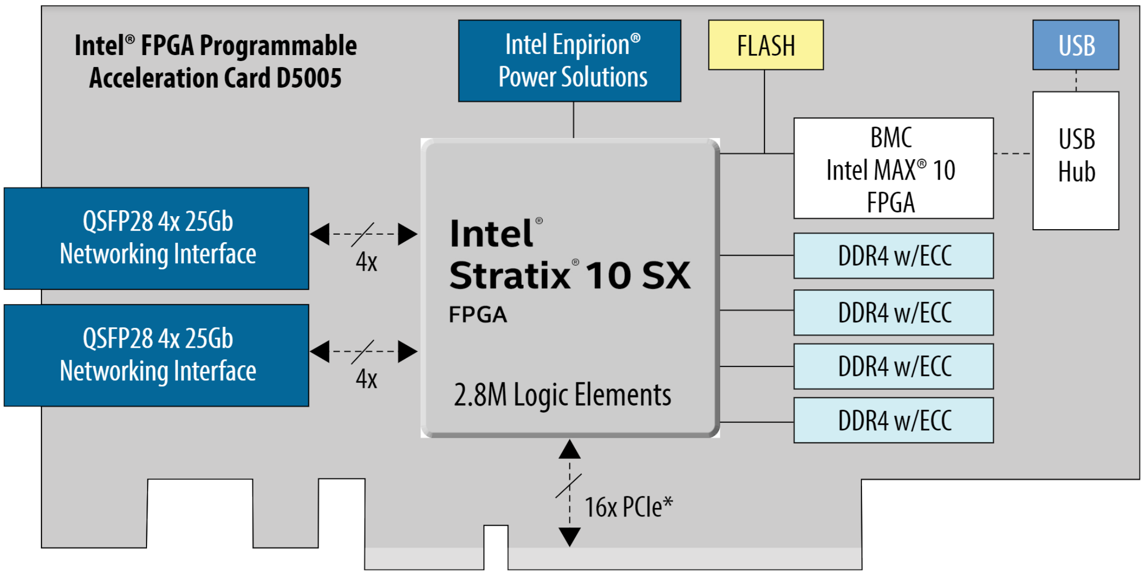

XEON Processor
Application (C++ or Python)
Intel Acceleration Stack
(OPAE, PCIe driver etc.)
Intel Accelerator Card D5005
AFU (RTL)
FIM
(CCI-P, Standard I/O Interfaces)

Figure 3.1: System Overview.

The system class *fpga-pac-s10* provides an Intel CPU coupled with an Intel FPGA *Programmable Acceleration Card* (PAC) D5005. This accelerator card contains a Stratix 10 SX FPGA (*1SX280HN2F43E2VG*). This FPGA has 2.8 million configurable logic elements. The card is a PCI Express card that supports PCIe 3.0 with 16 channels (PCIe x16). The accelerator card contains four channels of 8 GB DDR4-2400 with Error-Correcting Code (ECC). The card has two QSFP28 network interfaces that can be used for research requiring more than one accelerator card.

Figure 3.2: D5005 Acceleration card [11]



Intel® FPGA Programmable Acceleration Card D5005
Intel Enpirion® Power Solutions
FLASH
USB
BMC Intel MAX® 10 FPGA
USB Hub
QSFP28 4x 25Gb Networking Interface
4x
QSFP28 4x 25Gb Networking Interface
4x
Intel® Stratix® 10 SX FPGA
2.8M Logic Elements
DDR4 w/ECC
DDR4 w/ECC
DDR4 w/ECC
DDR4 w/ECC
16x PCIe*

Figure 3.3: D5005 Acceleration card with internal interfaces [10]

The Programmable Acceleration Card is coupled with an Intel XEON CPU. The processor and memory provided by the Hardware Acceleration Research Program have the following properties as described in Table 3.1. The hardware properties of the FPGA are given in Table 3.2.

Processor	**Intel® Xeon® Platinum 8280 Processor**
Base clock speed	2,70 GHz
Max Clock speed	4,00 GHz
No. cores	28
No. threads	56
Total cache	38,5 MB
Memory	**Available physical memory**
Clock speed	2933 MHz
Size	791 GB

Table 3.1: Hardware properties of the Intel CPU provided by the Hardware Acceleration Research Program

FPGA	**Intel 1SX280HN2F43E2VG**
Max. clock speed	1 GHz
No. logic elements	2800
Memory (M20K)	11721 Blocks with 229 Mbit
Memory (MLAB)	23796 Blocks with 15 Mbit

Table 3.2: Hardware properties of the Intel FPGA provided by the Hardware Acceleration Research Program

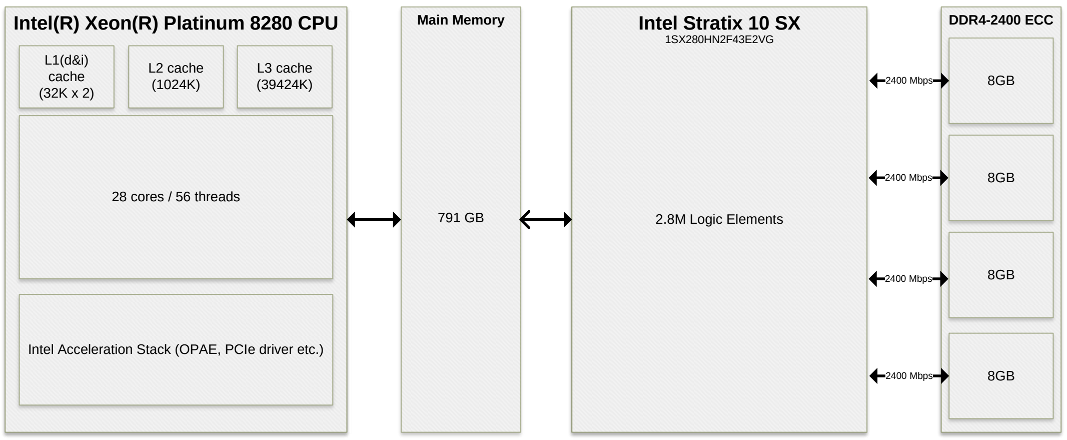


Figure 3.4: System Overview.

The CPU and the FPGA are interconnected using a PCIe 3.0 x16 connection. Initiating communication doesn't require specific knowledge of the PCIe bus. Instead the Intel HARP Program offers an abstraction layer for communicating over this connection. The software uses an OPAE layer as described in Section 3. The acceleration card abstracts this by using a FPGA Interface Manager (FIM) as described in section 4.1. The AFU interfaces with the FIM using CCI-P (Section 4.2).

3 OPAE

OPAE is an abstraction layer used to implement a uniform communication method between FPGA(s) and a CPU. It consists of a set of drivers, user-space libraries, and tools to discover, enumerate, share, query, access, manipulate, and reconfigure programmable accelerators [16]. For users this means that the provided abstraction makes resource access and management easier. Since users don't have to deal with infrastructure components such as register access, shared memory and synchronisation and reconfiguration. OPAE has bindings for C, C++ and Python software. An important part to facilitate the communication between the FPGA(s) and the CPU is CCI-P.

3.1 Drivers

OPAE provides drivers for communication interfaces that the FPGA supports. The FPGA used for this project uses PICe 3.0 x16. However, the developer does not require any knowledge of the protocol since OPAE makes an abstraction. By making the abstraction, the software for communication through different physical channels is always identical.

3.2 Libraries

This abstraction is exposed to the developer as a library. Using this library, the software can communicate with FPGA.

4 FPGA Interface Manager (FIM)

The FPGA Interface Manager (FIM) is a container which includes all region which is statically configured on the FPGA. This static region is provided by the platform to abstract external interfaces. This makes an AFU design more portable across different FPGA hardware and speeds up the development of an AFU. The FIM consists of a FPGA Interface Unit (FIU), a Core Cache Interface (CCI-P), an External Memory Interface (EMIF) and a High Speed Serial interface (HSSI). This is shown in Figure 3.5. On other FPGA hardware the FIM might include other interfaces [6].

4.1 FPGA Interface Unit (FIU)

The FPGA Interface Unit (FIU) manages all traffic to the host machine. A main task is to translate PCIe 3.0 x16 signals to CCI-P. Another important part of the FIU is the FPGA Management Engine (FME). This FME includes error monitoring and reporting, power and temperature monitoring, configuration bootstrap, bitstream security flows and remote debug access to ease the deployment and management of FPGAs in a data center environment [7].

4.2 CCI-P

Core Cache Interface Protocol (CCI-P) abstracts a couple of physical interfaces such as PCIe and UPI to transfer data between the FPGA(s) and the CPU.

CCI-P has the following features:

- MMIO request
- Memory request
- FPGA caching hint
- Virtual channels

An overview of some important CCI-P signals that an AFU should implement is given in Figure 3.6.

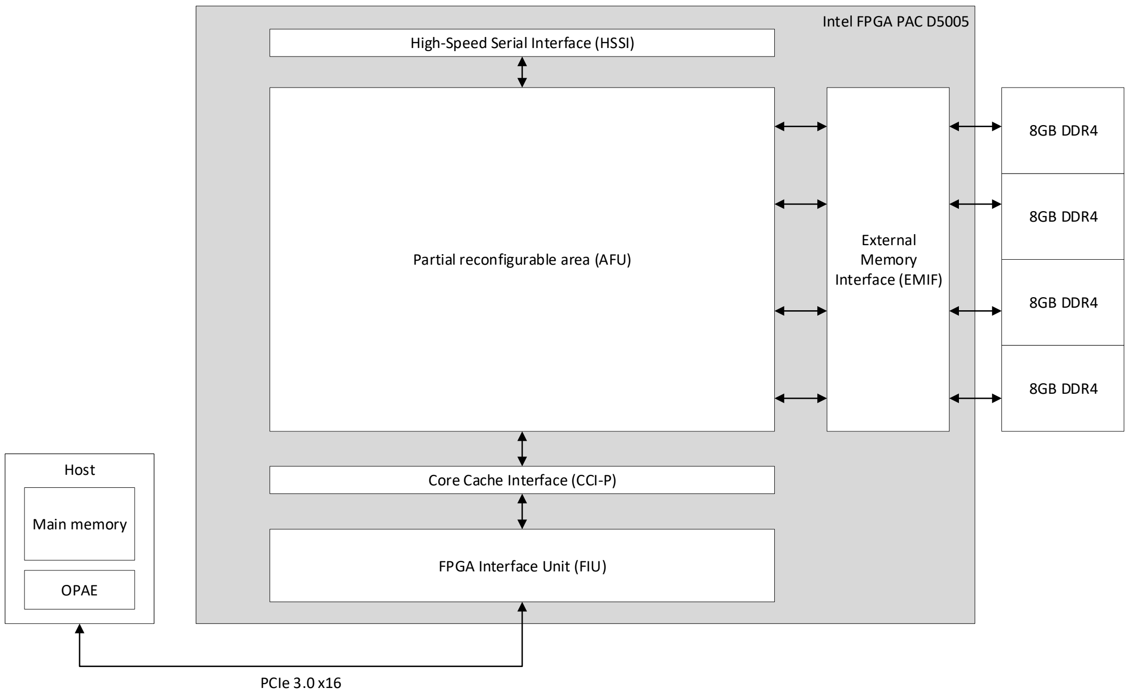

Intel FPGA PAC D5005
High-Speed Serial Interface (HSSI)
Partial reconfigurable area (AFU)
External Memory Interface (EMIF)
8GB DDR4
8GB DDR4
8GB DDR4
8GB DDR4
Host
Main memory
OPAE
Core Cache Interface (CCI-P)
FPGA Interface Unit (FIU)
PCIe 3.0 x16

Figure 3.5: FPGA Interface Manager, this is an overview of important external interfaces embedded into the FIM.

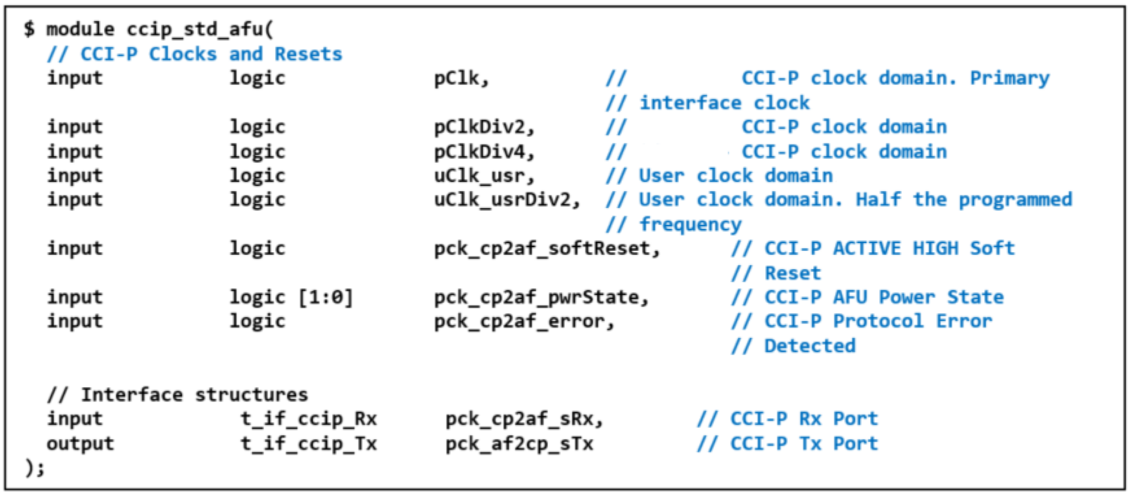
MMIO request

MMIO requests are used to send messages with a small payload (4B, 8B, 64B) [14] from CPU to I/O memory of the AFU and receive messages from I/O memory to the CPU (4B, 8B). The AFU receives MMIO read requests over `pck_cp2af_sRx.c0` and drives read responses over `pck_af2cp_sTx.c2`. The AFU receives write requests over `pck_cp2af_sRx.c0`. The maximum amount of outstanding MMIO requests is 64.

Virtual channels

For a direct physical link one can make use of Virtual Channels (VC). Virtual channels are introduced to make an abstraction of the communication interface/protocol. A FPGA can have multiple physical connections to the CPU. To speed up development these are abstracted. However, the chosen FPGA class has only a single physical connection and therefore all Virtual Channels map to the same physical interface. There are three Virtual Channels available:

- VLO, a low latency virtual channel that is mapped to VH0.
- VH0, a high latency virtual channel that is mapped to PCIe.



```
$ module ccip_std_afu(
  // CCI-P Clocks and Resets
  input         logic          pClk,         //          CCI-P clock domain. Primary
                                             // interface clock
  input         logic          pClkDiv2,     //          CCI-P clock domain
  input         logic          pClkDiv4,     //          CCI-P clock domain
  input         logic          uClk_usr,     // User clock domain
  input         logic          uClk_usrDiv2, // User clock domain. Half the programmed
                                             // frequency
  input         logic          pck_cp2af_softReset,      // CCI-P ACTIVE HIGH Soft
                                                         // Reset
  input         logic [1:0]    pck_cp2af_pwrState,       // CCI-P AFU Power State
  input         logic          pck_cp2af_error,          // CCI-P Protocol Error
                                                         // Detected

  // Interface structures
  input          t_if_ccip_Rx    pck_cp2af_sRx,       // CCI-P Rx Port
  output         t_if_ccip_Tx    pck_af2cp_sTx        // CCI-P Tx Port
);
```

Figure 3.6: Overview of CCI-P signals [5].

- VH1, a high latency virtual channel that is mapped to VH0.
- VA (Virtual Auto), in general a combined channel that uses all available physical links to achieve the highest bandwidth but on this platform mapped to PCIe.

According to [12] use the VA for producer-consumer type flows, VH0 channel for the latency sensitive flows and VH0 for data dependent flow. Since VH1 and VL0 are mapped to VH0 on this specific accelerator platform it does not matter which channel is selected.

Theoretical bandwidth

PCIe 3.0 has a theoretical bandwidth of 8 GT/s. The accelerator card is connected using a PCIe 3.0 X16 connection. This means that 16 lanes of PCIe 3.0 are used. Therefore the combined theoretical bandwidth is 8 GT/s $\cdot$ 16 lanes = 128 GT/s, which equals to 16 GB/s. Each CCI-P requests can read a maximum of 512 bits per cache lane. The maximum amount of cache lanes is 4 meaning that the maximum data transfer size is $\frac{512\text{ bits} \cdot 4\text{ cache lanes}}{8\text{ bits}} = 256$ Bytes.

These specifications are obtained from the *Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual* [9], Table 3 and Table 40.

Bandwidth measurement

The first step to discover some key characteristics of the FPGA. An important property is the throughput. The throughput is a combination of the measured read and write bandwidth. The throughput can be measured by reading and writing arbitrary data from and to the FPGA. To measure the throughput on the Intel compute environment you can make use of a built-in tool called `fpgadiags`. `fpgadiag` contains several tests to diagnose, test and report on the FPGA hardware [**Fpgadiag**]. To perform certain tests the tool requires the user to program a certain bitfile to the FPGA for diagnoses. The througput test (trput) requires the bitfile nlb mode 3. This bitfile contains a implementation of a loopback interface that is used for diagnosis tools. This bitfile can be found in the Acceleration Stack installation folder which is located on the Intel compute environment when the correct environment variables are set. Running the tool gives the following output:

```
[basnijkamp@iam-ssh1 ~]$ fpgadiag -m trput
Cachelines Read_Count Write_Count  Rd_Bandwidth   Wr_Bandwidth
1024  914706692   914706592   11.708 GB/s    11.708 GB/s
```

Read and write to main memory

CCI-P provides the ability to access host memory using physical addresses. CCI-P maps reads and writes to individual channels. Channel 0 is used for sending read requests and receiving read responses. Channel 1 is used for sending write requests and receiving write responses. Both channels have a similar interface. Each request contains a header which specify some specifics of the requests. These headers can have special properties like the specification of the number of cachelines used for the request and the specification of caching behaviour.

4.3 Clock specifications

Table 3.3 provides a list of available internal clock frequencies which can be used to synchronize the sorting accelerator hardware. We use the most common clock frequency *pClk* to prevent complications and synchronization issues with the CCI-P Interface.

Clock name	Clock frequency	Notes
pClk	250 MHz	Primary interface clock. All CCI-P interface signals are synchronous to this clock.
pClkDiv2	125 MHz	Synchronous and in phase with pClk. 0.5x the pClk clock frequency.
pClkDiv4	62.5 MHz	Synchronous and in phase with pClk. 0.25x the pClk clock frequency.
uClk_usr Min	10 MHz	Minimum user-defined clock. This clock is not synchronous with the pClk. You can adjust this clock using .
uClk_usr Default	312.5 MHz	Default user-defined clock. This clock is not synchronous with the pClk. You can adjust this clock using OPAE.
uClk_usr Max	600 MHz	Maximum user-defined clock. This clock is not synchronous with the pClk . You can adjust this clock using OPAE.
uClk_usrDiv2 Min	10 MHz	Minimum user defined clock that is synchronous with uClk_usr and 0.5x the frequency. **Note:** You can use OPAE to set the frequency to be a value other than half the uClk_usr frequency.
uClk_usrDiv2 Default	156.25 MHz	User defined clock that is synchronous with uClk_usr and 0.5x the frequency. **Note:** You can use OPAE to set the frequency to be a value other than half the uClk_usr frequency.
uClk_usrDiv2 Max	600 MHz	Maximum user defined clock that is synchronous with uClk_usr and 0.5x the frequency. **Note:** You can use OPAE to set the frequency to be a value other than half the uClk_usr frequency.

Table 3.3: List of available internal clock frequencies on the Intel PAC D5005 FPGA Accelerator Card.

The clock specifications from Table 3.3 are obtained from the *FPGA Interface Manager Data Sheet* [6], Table 4.

4.4 Intel FPGA Basic Building Blocks

Intel provides a set of basic building blocks to bring some additional features to the CCI-P interface. These features are bundled into three building blocks: BBB_cci_mpf, BBB_ccip_async and BBB_ccip_mux. Accompanied with these building blocks samples are provided to make it easier to start with AFU development. For certain functionality libraries on the software side are also included [17].

BBB_cci_mpf

Memory Properties Factory (MPF) enhances CCI-P by adding features such as virtual memory, ordered read responses, read/write hazard detection and masked (partial) writes [17].

BBB_ccip_async

This package provides a clock crossing shim for designs that require a lower clock frequency than CCI-P can interact with the interface [17].

BBB_ccip_mux

CCI-P Multiplexer makes it possible for multiple CCI-P agents to share a single CCI-P interface [17].

4.5 Local memory interface

The FPGA has its own local memory available. The FIM provides a Avalon Memory Mapped (Avalon-MM) slave interface to access each memory bank. This interface consists of the signals shown in Figure 3.7.

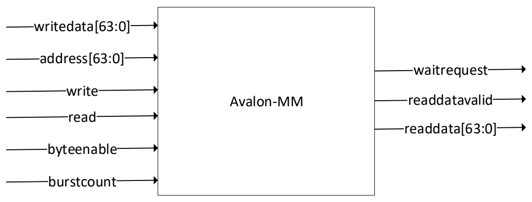


Figure 3.7: Signals of the Avalon Memory Mapped interface.

5 Accelerator Functional Unit (AFU)

The Accelerator Functional Unit (AFU) is the reconfigurable part of the FPGA which can be used for an accelerator design. This design is written in RTL and must follow some rules in order to fit onto the FPGA. An AFU has 2 communications paths to the host:

- FPGA to host: The FPGA can access host memory using CCI-P which uses multiple channels to provide simultaneous read and writes. The maximum throughput is achieved when using 4 Cache Lines (CL) on a channel [9].

- Host to FPGA: The second communication path is from host to FPGA. This implemented using Memory Mapped IO (MMIO). Using these small transactions status registers on the FPGA hardware can be controlled [9].

6 ASE

Intel Accelerator Functional Unit (AFU) Simulation Environment (ASE) provides a hardware and software co-simulation environment for a Intel Xeon Processor in combination with a Intel FPGA Programmable Acceleration Card D5005. The simulation environment provides a model for the CCI-P protocol as well as a model for the local memory attached to the FPGA. The ASE also validates Accelerator Functional Unit (AFU) compliance to the CCI-P protocol specification, Avalon Memory Mapped (Avalon-MM) Interface Specification and the Open Programmable Acceleration Engine (OPAE). ASE only supports single slot simulation to simulate one AFU and one software application at the same time. The ASE cannot guarantee that a simulated design can be synthesised. However, the environment provides some sanity checks to discover common errors like protocol correctness, illegal memory transactions and data hazards.

Although ASE is a comprehensive tool for functional simulation it has some limitations. ASE is a transaction-level simulator. It doesn't model PCIe packet structures or protocol layers. It cannot model caching and it cannot simulate accurate timings or latency of the design.

Chapter 4

Sorting Algorithms and Existing Research

Sorting is a common algorithmic operation that has been researched for years to optimise performance. There are a variety of different sorting algorithms. Each sorting algorithm has its own advantages and uses. A common property of these algorithms is that the time required to sort a set increases as the set is increased, also known as asymptotic order. To reduce this increase in time, some algorithms attempt to sort in parallel, using multiple processor cores.

1 Frequently used sorting algorithms

Although there are many different sorting algorithms, some are more commonly used than others. Here we discuss some of the most popular sorting algorithms used in computer science. The algorithms can be distinguished on the basis of their efficiency on small and large data sets, complexity and performance.

Insertion sort

Insertion sort is a simple algorithm that iterates over each element and inserts the new element into a sorted list. This is efficient for small datasets and especially when that dataset is mostly sorted. However, for large datasets this becomes inefficient as the algorithm has to perform long iterations over the entire dataset. Insertion sort is often used as part of more complex algorithms. [20] [21]

Selection sort

Selection sort iterates over all elements and tries to find the minimum of the unsorted part and places it at the beginning. The operation requires no more than N swaps. [20] [21]

Merge sort

Merge sort merges two presorted data sets into one list. It starts by merging elements in pairs of two. When this operation is completed it merges it into small lists of four and this continues until the whole set is sorted/merged. Merge sort scales really well for large data sets since the worst running time is O(n log n). For a comparison sort algorithm (insertion sort, merge sort etc.) O(n log n) is the best possible performance. [20] [21]

TimSort

Tim sort is a combination of merge sort and insertion sort. The algorithm tries to find sub-sequences which are already sorted and uses this to sort the other parts more efficiently. This algorithm is designed for real-word data. Real-world data often includes already sorted sub-sequences which makes it possible to sort a list more efficiently. TimSort is highly standardised among many different programming languages like Python, Java and Perl. [21]

Heapsort

Heapsort is similar to selection sort. It takes the same approach of finding the smallest/largest element and placing this at the end. A key difference with selection sort is how the data is stored during the sort. In heapsort, the data set is converted into a heap data structure. This heap makes searching for the smallest/largest element more efficient. With selection sort it can take up to O(n) to find the element, when using a heap it only takes O(log n). [20] [21]

Quick sort

Quick sort partitions the dataset by selecting a pivot. The algorithm places the pivot in the correct position in the list and then orders all smaller items before that pivot and all larger items after that pivot. A difficult task is to determine a pivot by which the algorithm becomes most efficient. [20] [21]

Shell sort

Shell sort improves on insertion sort by moving elements in multiple positions instead of just one position. By first sorting elements that are far apart, the distance between element positions becomes progressively smaller. When a list is partially sorted, these gaps are small, making it efficient. [20] [21]

Bubble sort

Bubble sort is a simple algorithm that compares the first two elements, swaps them if necessary, and continues this process until the end of the list. This is inefficient for large unsorted datasets. Bubble sort is efficient on datasets that are nearly sorted. However, the elements must not be significantly out of place. [20] [21]

Algorithm	Best	Average	Worst
Insertion sort	n	n^2	n^2
Selection sort	n^2	n^2	n^2
Merge sort	$n\,log(n)$	$n\,log(n)$	$n\,log(n)$
TimSort	n	$n\,log(n)$	$n\,log(n)$
Heapsort	$n\,log(n)$	$n\,log(n)$	$n\,log(n)$
Quick sort	$n\,log(n)$	$n\,log(n)$	n^2
Shell sort	$n\,log(n)$	$n^{\frac{4}{3}}$	$n^{\frac{3}{2}}$
Bubble sort	n	n^2	n^2

Table 4.1: Frequently used sorting algorithms and their time complexity [20] [21].

2 Sorting networks

2.1 Combinational sorting networks

Sorting networks can be implemented in hardware with combinational, synchronous, and pipelined circuits. In a combinational sorting network, the network does not use a clock for synchronisation. Instead, it uses only a large block of combinational logic through which the signal is propagated. A key characteristic of such a combinational implementation is that no registers are used between circuits. You can only provide a single set to be sorted in the network at all time, when a new set needs to be sorted, it must wait until the sorting network is finished and the signal is propagated to the output register. The time required to sort a new set after a set is inserted into the circuit, the latency, is entirely determined by the length of the total combinational signal path between the input and the output of the sorting network. The throughput is $\frac{1}{Latency}$ sets per second. [15]

2.2 Synchronous Sorting Networks

The combinational sorting network can be made synchronous by inserting registers between the stages of the combinational circuit. In this case, a register is added after each comparator. By adding registers, the length of the signal path is divided into several stages. A clock is used to move the data from stage to stage in the circuit. With the registers, the design requires more area but can be clocked higher because the maximum clock speed is now determined by a shorter maximum path (the path from register to register, rather than the entire combinational circuit). This makes the latency $\frac{\text{No. stages}}{f_{clk}}$. The registers make the circuit synchronous, but not fully pipelined, since signals not processed in a stage are not buffered, making the throughput $\frac{1}{2}f_{clk}$. [15]

2.3 Pipelined Sorting Networks

To fully pipeline the synchronous sorting network, additional registers must be added. By adding registers to each signal after each state, and not just to the processed signals, all signal paths have the same length. Therefore, the throughput is completely defined by f_{clk}. [15]

2.4 Basic sorting element

Each sorting network implementation shares the same basic sorting element. This element is a building block for sort trees. The functionality of this building block is Compare-Exchange. The building block evaluates two inputs and checks which one is greater/smaller compared to the other.

If the inputs are not yet in order an exchange will be made and therefore swapping the 2 inputs. A basic Compare-Exchange block is shown in Figure 4.1.

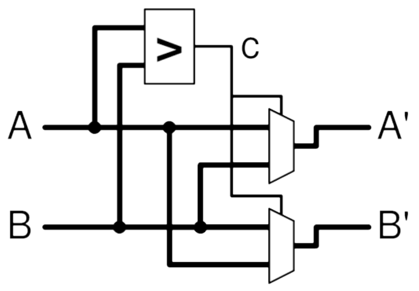


Figure 4.1: Compare-Exchange element [18]*: figure 3a.*

An alternative to the Compare-Exchange element is the Select-Value element, as shown in Figure 4.2.

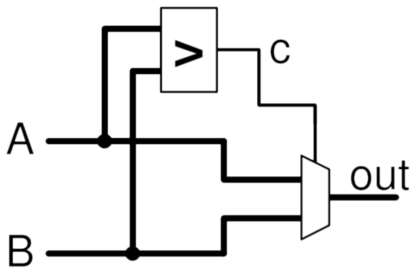


Figure 4.2: Select-Value element [18]*: figure 3b.*

One advantage of using basic sort elements to assemble a sort tree is flexibility. One only needs to change the mapping of the input and output types of the basic sort element to make the sort network compatible with a different sorting requirement such as fractions or larger numbers by increasing the number of wires to a sort element [18].

3 Batcher's even-odd merge sort network

Research [13] shows that the Batcher's even-odd merge sort network is most efficient in hardware resource usage and throughput.

3.1 Comparator unit

A comparator unit has two inputs and 2 outputs, one output produces the minimum of both inputs and the other output produces the maximum of the inputs (Figure 4.3a). For easier representation of comparator networks we use a simplified version of a comparator represented by a vertical line between two horizontal lines as shown in Figure 4.3b.

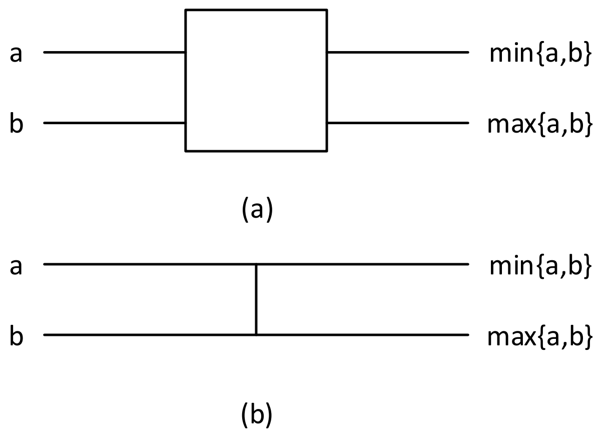


Figure 4.3: Comparator unit.

3.2 Odd Even Merge Network (OEM)

An Odd Even Merge Network is a circuit that consists of multiple comparator units to merge two sorted arrays of the size N into a sorted array of size $N \cdot 2$. N should be a power of 2 ($n = 2^k$). The first step of a OEM is to extract the odd elements to one side and the even elements to the other side. The second step is to merge the odd side and even side recursively and in parallel. The outputs are interleaved meaning that the first input of the odd side is the first output and the first output of the even side is the second output and so on as shown in Figure 4.4.

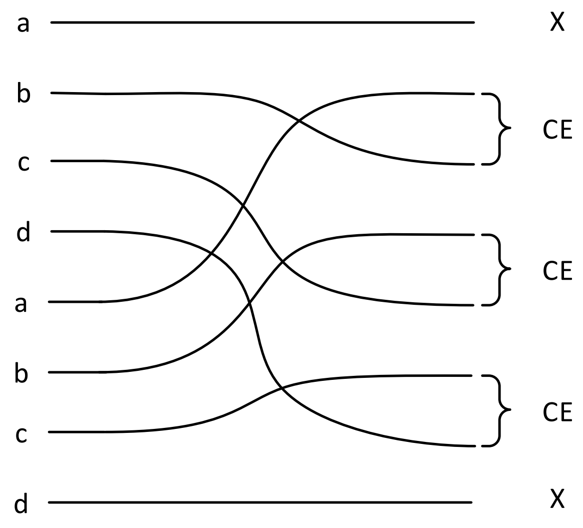


Figure 4.4: OEM interleaved, CE is short for Compare and Exchange.

A basic OEM is a 1x1 OEM which merges 2 sorted arrays of size $N = 1$. This contains a single comparator unit and is shown in Figure 4.5a. A 2x2 OEM uses the same steps as a 1x1 OEM, we interleave the outputs and recursively compare and exchange (Figure 4.5b). In Figure 4.5c a 4x4 OEM is shown, in this figure you can easily identify a 2x2 OEM is used to compose the 4x4 OEM. This shows how a Odd Even Merge Network can be composed of smaller (reusable) building blocks. When increasing the size of an OEM the number of comparisons and the number of parallel execution comparison steps increases.

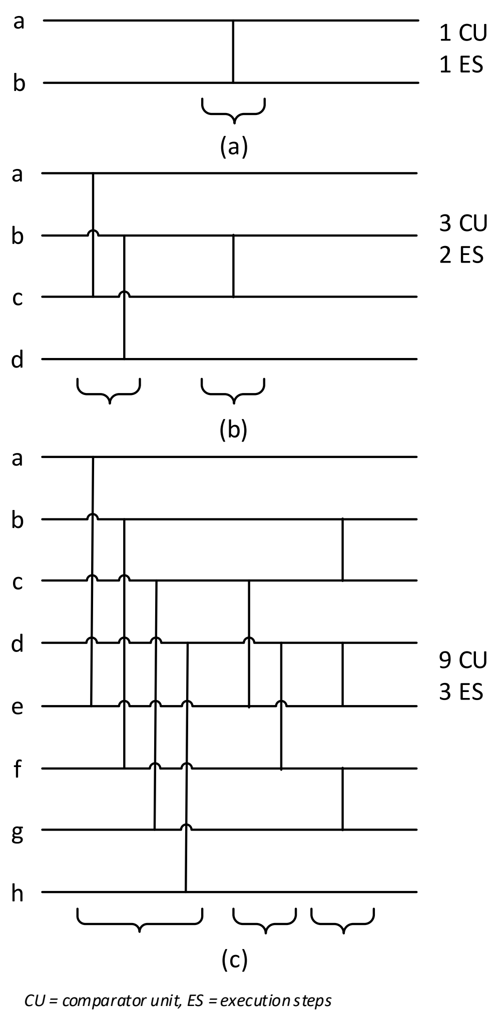


Figure 4.5: 1x1 OEM (a), 2x2 OEM (b) and a 4x4 OEM (c).

3.3 Odd Even Merge Sort (OEMS)

We can use a OEM as described in Section 3.2 to compose a network which not only merges two sorted arrays but can sort an unsorted array of size $N \cdot 2$. We call this network an Odd Even Merge Sort Network (OEMS). A 1x1 OEMS is exactly the same as a 1x1 OEM since both inputs are already sorted. When composing a 2x2 OEMS we first want to make sure that the two input arrays are sorted before passing it to a merge network. This is achieved by adding 2 extra comparator units as shown in Figure 4.6.

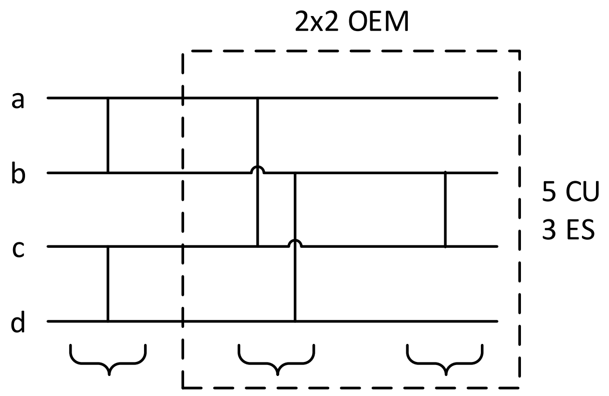


Figure 4.6: 2x2 OEMS.

Odd Even Merge Sort Networks support a high level of parallelism since many comparison operations are independent and can be executed in the same time slot. This feature makes it a very good solution since a FPGA focuses on parallelism. A regular CPU might have multiple processing cores but each core is operated sequentially and can achieve only a limited level of parallelism.

3.4 Analysing hardware cost

Analysing complexity of Odd-Even Merging

$$T(2N) = 1 + log_2(N) \tag{4.1}$$

Where $T(2N)$ is the time required for an Odd Even Merge Network to merge two sorted lists of size N.

$$CC(2N) = 1 + N \cdot log_2(N) \tag{4.2}$$

Where $CC(2N)$ is the comparator count of an Odd Even Merge Network of size N.

Analysing complexity of Odd-Even Merge Sorting

$$T(N) = \frac{log_2(N) \cdot (log_2(N) + 1)}{2} \tag{4.3}$$

Where $T(N)$ is the time required for an Odd Even Merge Sort Network to sort an unsorted list of size N.

$$CC(N) = \Theta(N \cdot log_2^2(N)) \tag{4.4}$$

Where $CC(N)$ is the comparator count of an Odd Even Merge Sort Network of size N.

These analyses are obtained from *CS 662: Batcher's sort* [22].

Chapter 5

Concept

In this chapter, we discuss the general concept and architecture of the sorting accelerator, a number of solutions are being explored and a final solution is chosen.

1 Data delivery / Communication

Transferring data to the FPGA to process is key in every solution that is being explored. To achieve a high throughput the data delivery must be as fast as possible to ensure that the bottleneck of data transfer is limited to a minimum. This optimised data delivery can be implemented using CCI-P (as discussed in Section 4.2). All four cache-lines must be used in order to get the maximum throughput. Each cache line is a bus with a width of 512 bit. Therefore, a total of 2048 bits can be transferred each read/write request. MPF is used to add more features to the base CCI-P implementation. A valuable feature is Virtual to Physical address translation (VTP). By using VTP we can use virtual addresses instead of physical ones; therefore it is possible to store all data in one continuous virtual memory block while in reality the data doesn't need to be aligned continuously. This makes the implementation easier because reading the data no longer requires the usage and implementation of advanced hardware to jump between addresses.

2 Exploration

When designing the sort algorithm we can define three possible implementations. A hybrid solution, full sort on FPGA using direct access to host memory and sort on FPGA using large chunks of data.

2.1 Hybrid solution

In a hybrid solution, we make use of the CPU and the FPGA. Firstly we move the data to the FPGA, since the transfer size of the data is small we can quickly sort these chunks immediately when they come in. When a chunk is sorted it is sent back into the main memory, so the chunk can be merged with other chunks that are already sorted. This solution spreads the workload of sorting between the FPGA and CPU. An advantage is that the AFU on the FPGA is relatively simple and doesn't require much control logic to buffer and iteratively merge the chunks of data. A big disadvantage is the CPU load. A goal to use an FPGA as accelerator is not only to make it faster but also more resource efficient by dedicating specialised hardware for sorting so that the CPU cycles can be saved for other tasks on the server.

2.2 Full sort on FPGA

A second solution is to move the whole sorting chain, consisting of sort and merge, to the FPGA. A big advantage of this approach is the minimum amount of CPU cycles that is required. The CPU needs to initiate and prepare the data for the sort but in between it can use its CPU cycles for other tasks which is specifically beneficial in datacenters where CPU resources are shared [1]. When less CPU resources are required it improves power usage as well since the FPGA solution is more efficient compared to the CPU counterpart [19]. The solution works as follows, the sorting part is similar to the hybrid solution (see Section 2.1). The FPGA reads chunks of data using direct memory access using CCI-P (see Section 4.2).

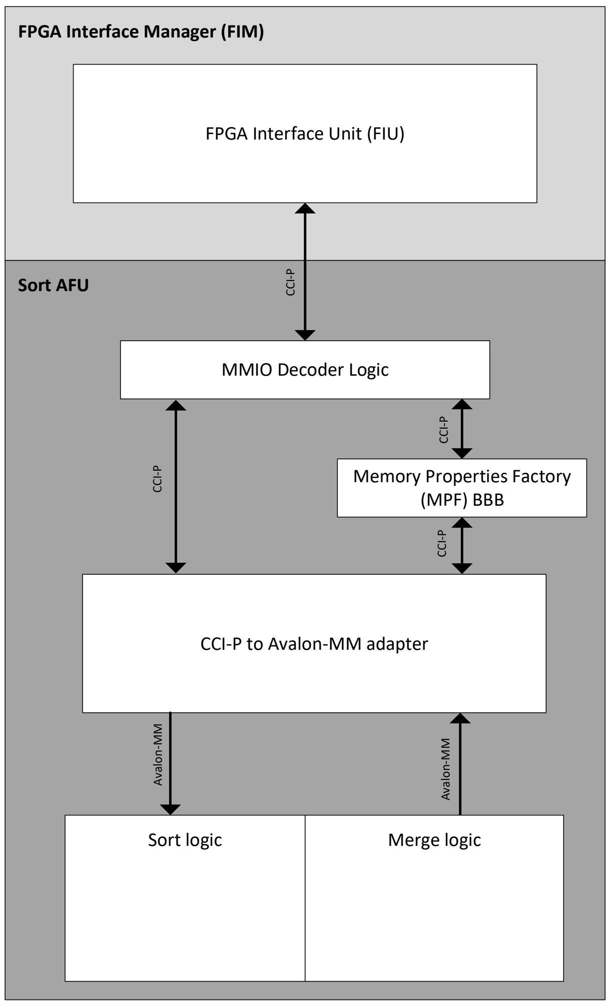

FPGA Interface Manager (FIM)
FPGA Interface Unit (FIU)
Sort AFU
CCI-P
MMIO Decoder Logic
CCI-P
CCI-P
Memory Properties Factory (MPF) BBB
CCI-P
CCI-P to Avalon-MM adapter
Avalon-MM
Avalon-MM
Sort logic
Merge logic

Figure 5.1: Concept 2 full sort on FPGA.

2.3 Full sort on FPGA using local memory

The third solution is similar to the Full sort on FPGA (see Section 2.2). Data is transferred in chunks using CCI-P and on arrival sorted. The key difference is that the data now not will be transferred back to main memory using the relatively slow CCI-P interface but to the local memory using direct access instead. The FPGA has 32 GB of local memory available, this faster memory, (compared to accessing the main memory using CCI-P, 15.360 GB/s instead of 12.644 GB/s) is used to store the results of the initial sort that happens when the data arrives. When all data is transferred to the FPGA or when the memory of the FPGA is filled the data will be merged using an OEM (Odd Even Merge Network). Merging requires a lot of read/write operations to the memory where the presorted data is stored. Since this solution uses the local memory, which has a higher throughput and lower latency than accessing main memory over the CCI-P interface, the OEM can operate faster than the other solutions. A downside of this solution is that to be sorted data is limit to 32GB per execution because of the physical memory limit of the FPGA configuration. This makes the solution less scalable and therefore makes it slightly more difficult to calculate timings.

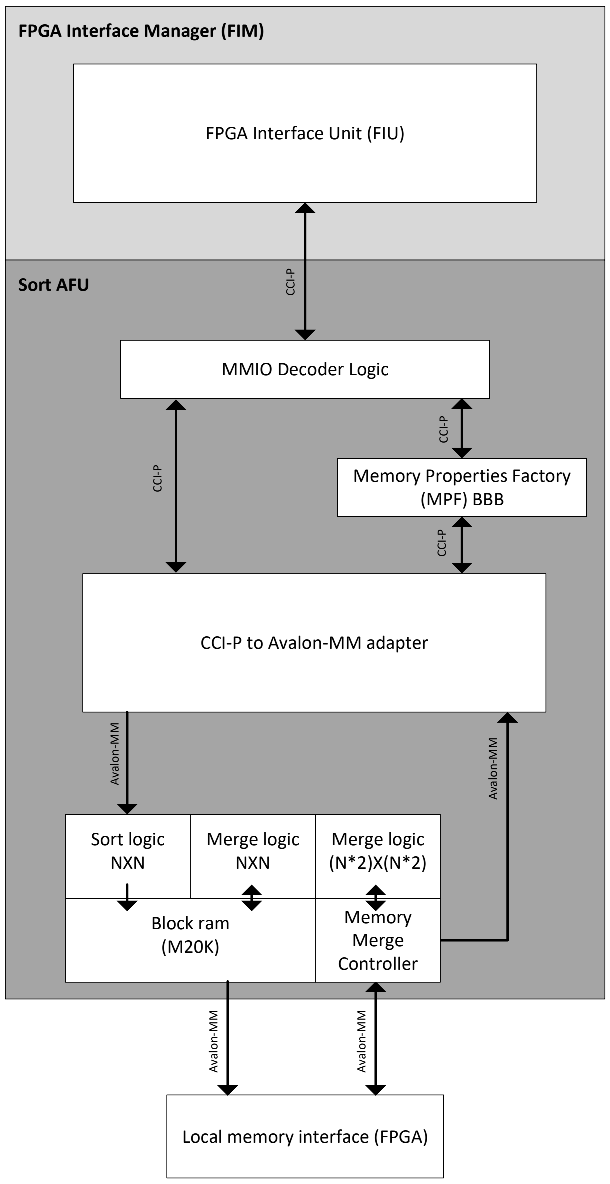


Figure 5.2: Concept 3 full sort on FPGA using local memory.

Concept 3 has been chosen as the final solution. This concept reduces the maximum amount of load of the CPU and makes the most use of the available FPGA resources compared to the other given concepts. By using faster local memory of the FPGA for merging, the overall throughput is increased since the sorting network is less dependent on slower memory due to memory locality (the local memory of the FPGA can be accessed at higher speeds and lower latency than the host memory).

Chapter 6

Implementation

In the following chapter a more detailed description of the implementation is given. With the use of textual and visual descriptions the created sorting accelerator is explained in detail.

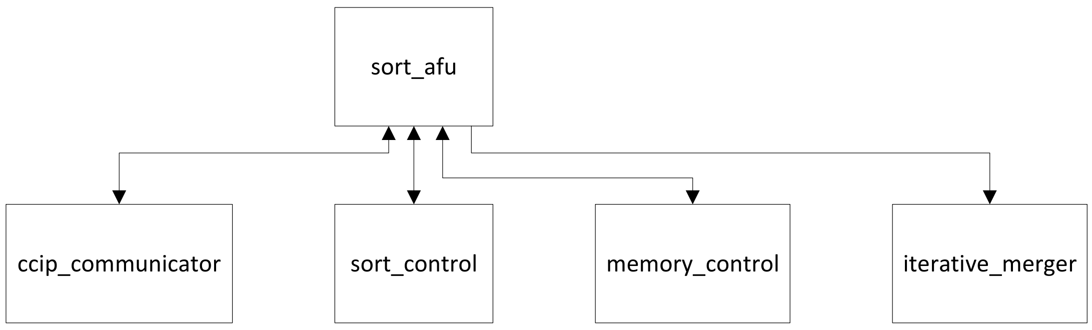


Figure 6.1: Architecture of sorting accelerator.

Figure 6.1 shows the general architecture of the sorting accelerator. The next sections will go in detail how each subcomponent's works and is interlinked.

1 sort_afu

The sort_afu is the top-level unit of the sort accelerator. This unit instantiates all the subcomponents needed for the complete sort flow. sort_afu uses a state machine to control the flow. The state machine is shown in Figure 6.2. In state IDLE, the sorter waits for a new sorting job provided by the software interface. When a sort request is received and the data to be sorted is stored into main memory, the state machine transitions to the READ _SORT _STORE state. In this state, the sort_afu signals the ccip_communictator to begin reading data from main memory. This data is then passed to the sort controller, which already performs some early processing on this data. The sort control data is written to local memory using the memory_control unit. When this data is stored in local memory, the state machine transitions to the MERGE _AND _WRITEBACK state. In this final state, the data is read from local memory and iteratively merged by the iterative_merger. When this merge is complete, the data is transferred back to main memory using the ccip_communicator.

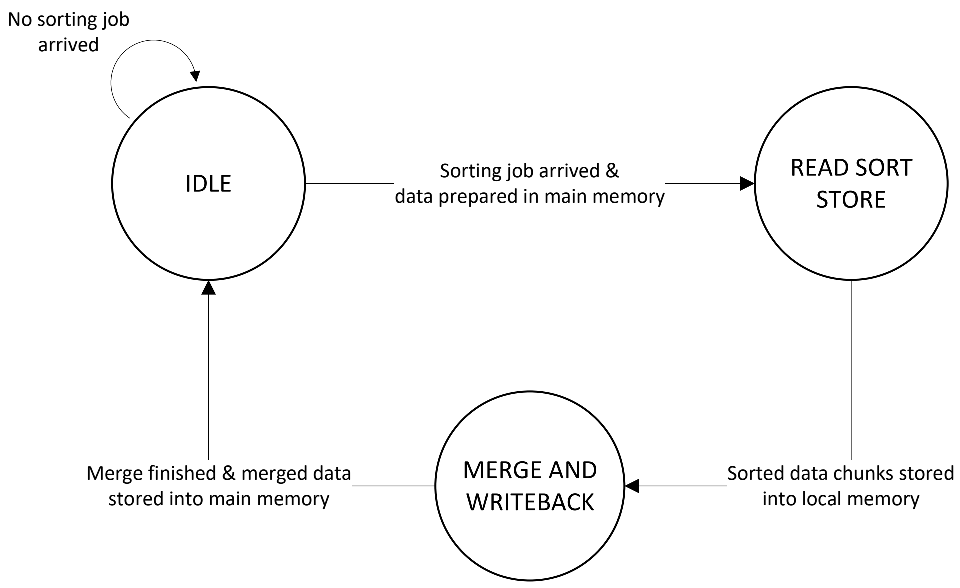

No sorting job arrived
IDLE
Sorting job arrived & data prepared in main memory
READ SORT STORE
Sorted data chunks stored into local memory
MERGE AND WRITEBACK
Merge finished & merged data stored into main memory

Figure 6.2: State machine of sort_afu.

2 ccip_communicator

The ccip_communicator is responsible for all communication between the CPU and the accelerator card. This unit uses the CCI-P interface to read data from and write data to main memory. To simplify the control flow, the ccip_communicator uses its own state machine. The state machine consists of the following states: IDLE, READ and WRITE and is shown in Figure (6.3). In the READ state, it sends read requests for a specific memory range. This range is provided by the software interface and is guaranteed to be sequential since we use virtual addressing. Virtual addressing is a feature provided by the MPF BBB (Memory Properties Factory Basic Building Block). It also ensures that requests are received in the correct order. During the same state, the ccip_communicator waits for the read responses and forwards them as output. When the 4th cache line has been read, the ccip_communicator signals this to the sort controller. The sort controller can then process the data from all 4 cache lines simultaneously. The WRITE state writes the data back to the main memory in several iterations, depending on the size of the input data and limited by the local memory on the FPGA (32 GB).

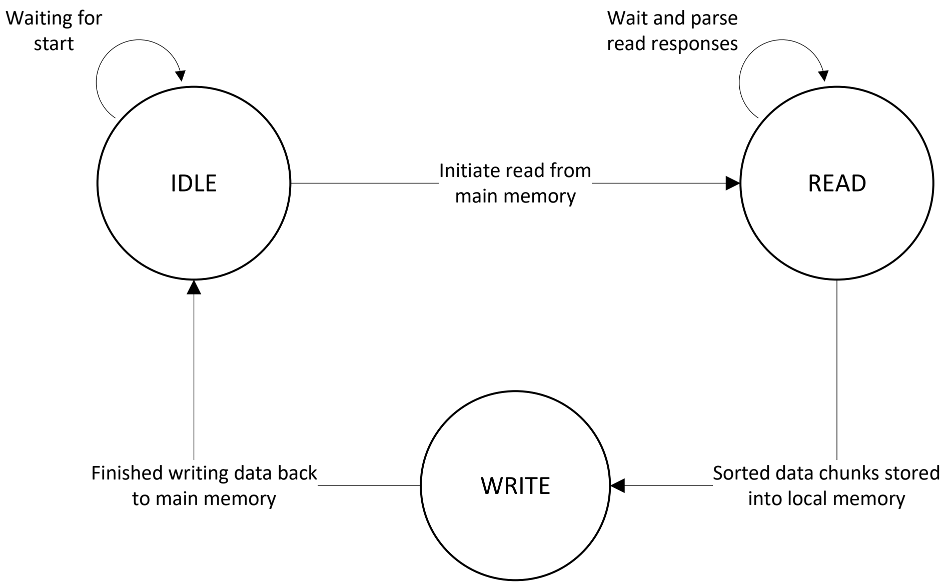

Waiting for start
Wait and parse read responses
IDLE
Initiate read from main memory
READ
Finished writing data back to main memory
WRITE
Sorted data chunks stored into local memory

Figure 6.3: State machine of ccip_communicator.

3 Sort Controller

The sort controller receives data from the ccip_communicator and instantiates a 32X32 OEMS (Odd-Even Merge Sort) network. Since not all cache lines arrive on the same clock cycle, the sort controller must buffer the data before passing it to the OEMS. The Sort Controller uses an OEMS of size N=32 because using 64 bit values with an OEMS of N=32 results into a output of 2048 bits every cycle. This matches the maximum amount of data that can be transferred each clock cycle using CCI-P (see section 4.2: Theoretical bandwidth).

Implementation of OEMS 32x32

The 32x32 Odd-Even Merge Sort (OEMS) network is derived by mixing lower order mergers into a single entity. The most basic merger is a 1x1 OEM(S) that compares two 64-bit data entries and outputs the lower one on the first output and the higher one on the second output. With this basic merger we can build a merge tree of arbitrary size, as explained in (see OEM 3.2). Therefore, we need to use one instantiation of a 16x16 OEM, two of an 8x8 OEM, four of a 4x4 OEM, eight of a 2x2 OEM and sixteen of a 1x1 OEM. It is important to note that between each merge stage that has data dependencies and therefore cannot be executed in parallel, a register is added to pipeline the design. By using a pipelined merge/sort network, we can improve the throughput as explained in 2.3.

4 Memory Controller

The memory controller manages all accesses to local memory (memory directly connected to the FPGA). The FPGA has 32 GB of DDR4 memory divided into four memory banks of 8 GB each. A single transfer to memory has a maximum size of 64 bits. Since we have four memory banks, we can access them simultaneously and can transfer $64 \cdot 4 = 256$ *bit* of data per clock cycle. The memory controller receives the data generated by the sort controller. This data consists of 32 sorted entries, each 64 bits in size. Since 64 bits fit exactly into 1 memory address, we can write all this data to memory in $\frac{32 \cdot 64}{256} =$ 8 clock cycles. The memory controller also uses a simple state machine to prevent concurrent reads and writes.

5 Iterative Merger

After sorting all incoming data blocks and replacing the Odd-Even Merge Sort network with an iterative merger. This merger iterates over all sorted chunks and merges them. The implementation of this merger uses a tree of Odd-Even Merge Networks (OEM). These OEMs reuse a portion of the OEMs required by Odd-Even Merge Sort Networks (OEMS). Reusing these components saves area on the FPGA to support even larger merge trees in future implementations.

Chapter 7

Results

In this chapter the result of the synthesis is given and the produced hardware is explained. Furthermore, performance measurements of various sorting algorithms and the sorting accelerator will be given to make a comparison possible between existing solutions and the created accelerator.

1 Hardware implementation of the *Odd Even Merge* networks

1.1 1x1 *Odd Even Merge* (OEM) network

Figure 7.1 shows the implementation of the comparator building block. The comparator block accepts two 64 bit inputs (*unsorted_data_1* and *unsorted_data_2*), compares them and forwards the largest data input to the first output (*sorted_data_1*) and the smallest data on the second output (*sorted_data_2*). The design consists of 1 comparison unit, 3 multiplexers and 2 registers, which consist of 64 D flip-flops each. The unit has a clock (*clk*), *enable* input and a synchronous *reset*. The execution time of this comparator unit is 1 clock cycle ($\frac{1}{f_{clock}}$).

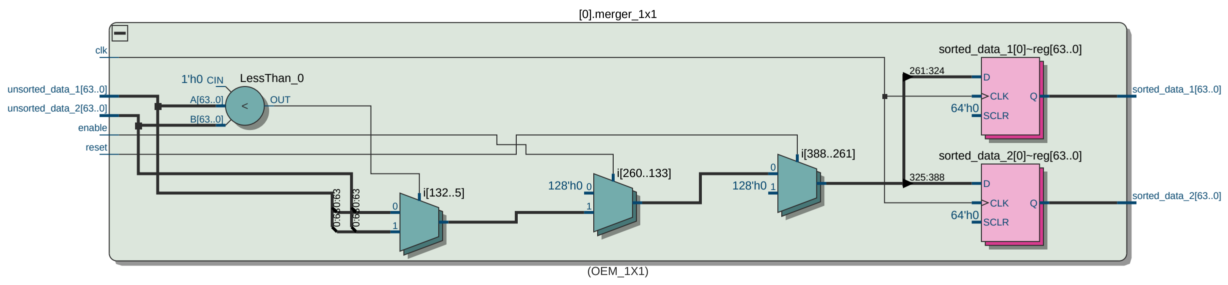


Figure 7.1: Hardware implementation of a comparator building block (1x1 OEM).

1.2 2x2 *Odd Even Merge* (OEM) network

Figure 7.2 shows the implementation of the 2x2 OEM building block. The 2x2 OEM block accepts two sets of 64 bit sorted inputs ($unsorted\_data\_part1$ and $unsorted\_data\_part2$), merges them and gives a sorted list of four 64 bit values. The design consists of 3 comparator units, 8 added multiplexers and 8 registers, consisting of 64 D flip-flops each. The unit has a clock (clk), $enable$ input and a synchronous $reset$. The execution time of this 2x2 OEM unit is 2 clock cycles ($\frac{2}{f_{clock}}$).

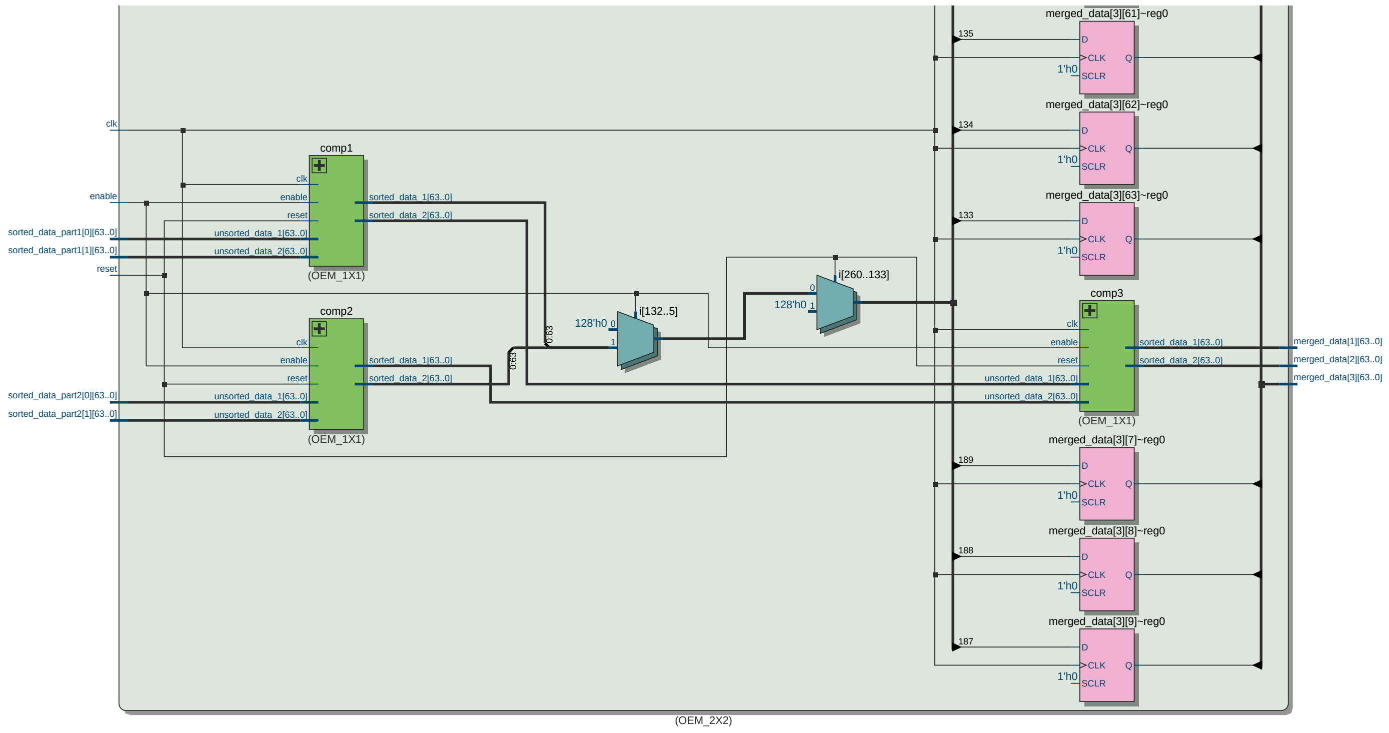


Figure 7.2: Hardware implementation of a 2x2 OEM.

1.3 4x4 *Odd Even Merge* (OEM) network

Figure 7.3 shows the implementation of the 4x4 OEM building block. The 4x4 OEM block accepts 2 sets of 4 64 bit sorted inputs ($unsorted\_data\_part1$ and $unsorted\_data\_part2$), merges them and gives a sorted list of 8 64 bit values. The design consists of 9 comparator units, 32 added multiplexers and 32 registers, consisting of 64 D flip-flops each. The unit has a clock (clk), $enable$ input and a synchronous $reset$. The execution time of this 4x4 OEM unit is 3 clock cycles ($\frac{3}{f_{clock}}$).

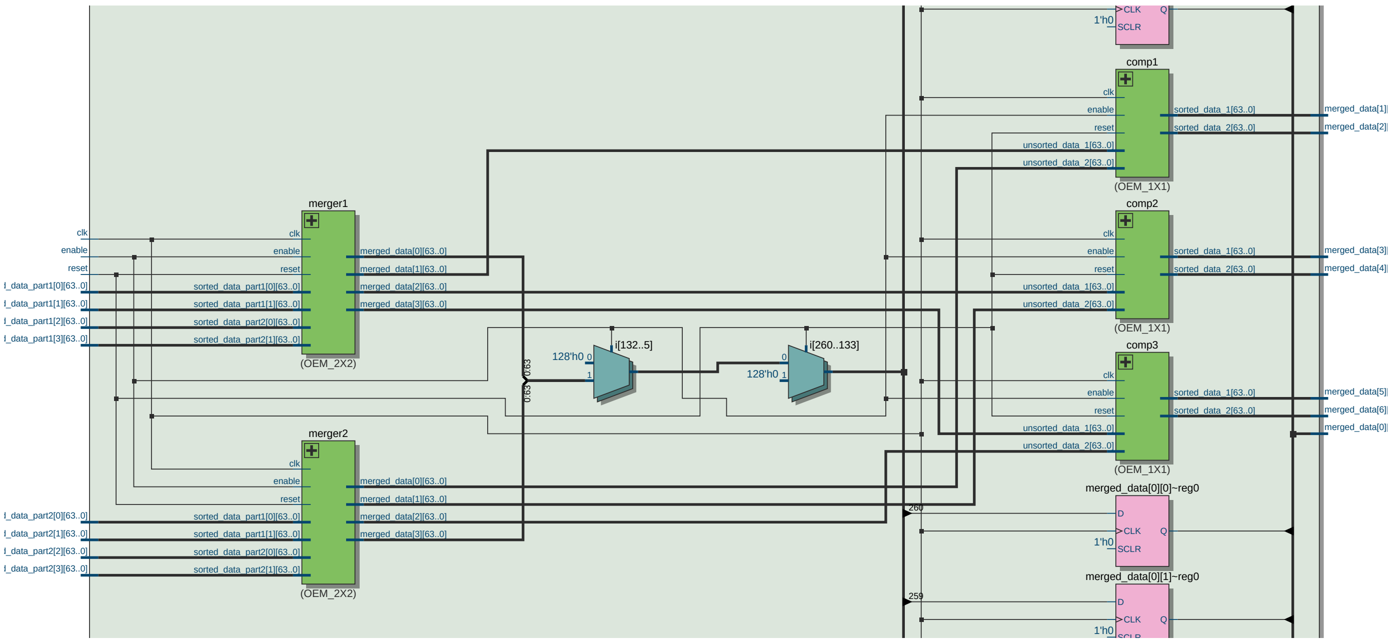


Figure 7.3: Hardware implementation of a 4x4 OEM.

1.4 8x8 ***Odd Even Merge*** (OEM) network

Figure 7.4 shows the implementation of the 8x8 OEM building block. The 8x8 OEM block accepts 2 sets of 8 64 bit sorted inputs ($unsorted\_data\_part1$ and $unsorted\_data\_part2$), merges them and gives a sorted list of 16 64 bit values. The design consists of 27 comparator units, 128 added multiplexers and 128 registers, consisting of 64 D flip-flops each. The unit has a clock (clk), $enable$ input and a synchronous $reset$. The execution time of this 8x8 OEM unit is 4 clock cycles ($\frac{4}{f_{clock}}$).

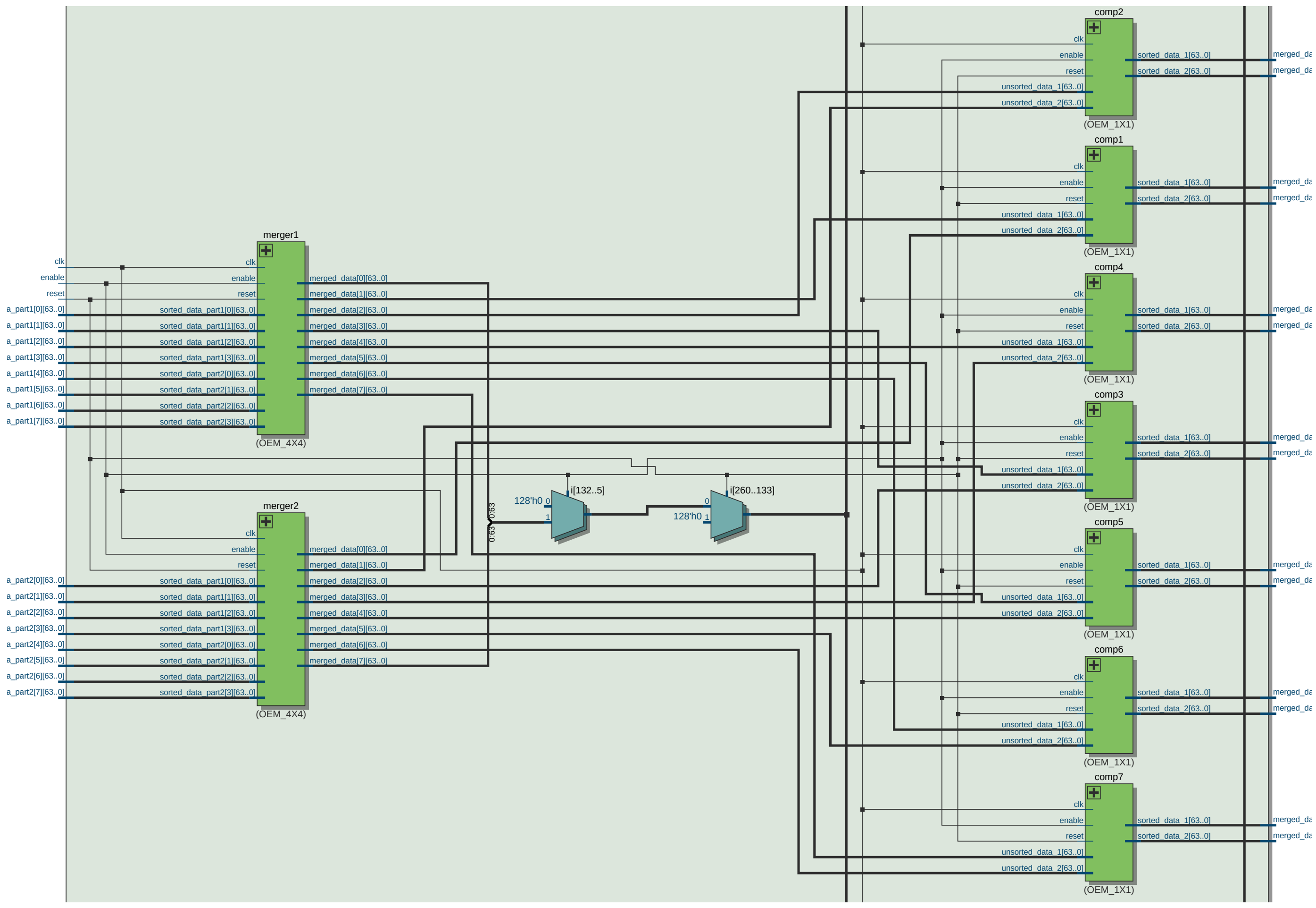


Figure 7.4: Hardware implementation of a 8x8 OEM.

1.5 16x16 *Odd Even Merge* (OEM) network

Finally, Figure 7.5 shows the implementation of the 16x16 OEM building block. The 16x16 OEM block accepts 2 sets of 16 64 bit sorted inputs ($unsorted\_data\_part1$ and $unsorted\_data\_part2$), merges them and gives a sorted list of 32 64 bit values. The design consists of 69 comparator units, 512 added multiplexers and 512 registers, consisting of 64 D flip-flops each. The unit has a clock (clk), *enable* input and a synchronous *reset*. The execution time of this 16x16 OEM unit is 5 clock cycles ($\frac{5}{f_{clock}}$).

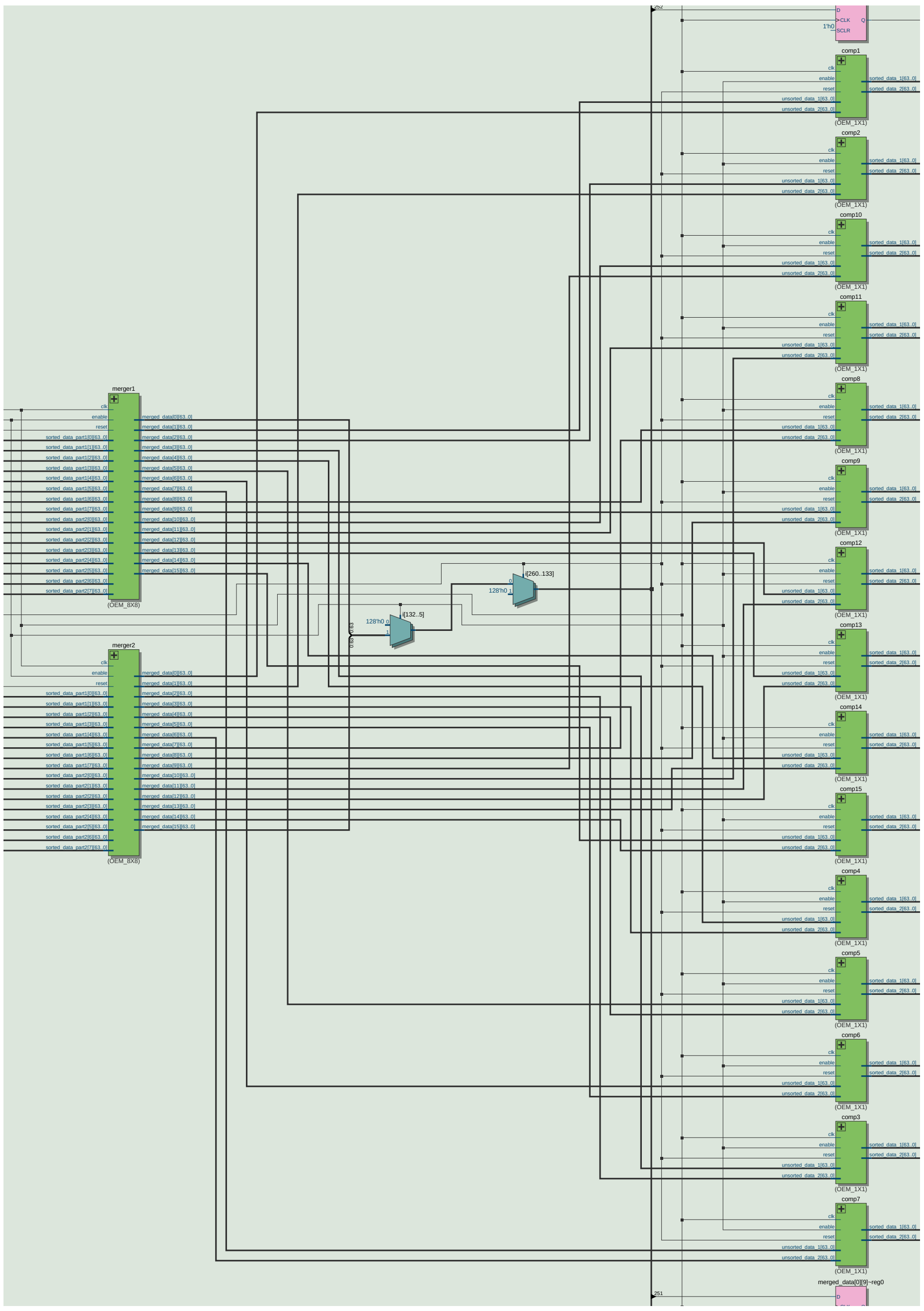


Figure 7.5: Hardware implementation of a 16x16 OEM.

2 Sort Controller unit

2.1 Hardware implementation

32x32 *Odd Even Merge Sort* (OEMS) network

Figure 7.6 shows the implementation of the 32x32 *Odd Even Merges Sorter* (OEMS) which is the main component of the Sort Control unit. This sorter accepts four sets of 8 64 bit unsorted inputs, sorts them and merges these values resulting one set of 32 64 bit outputs. The OEMS consists of 199 comparator units. The execution time of this 32x32 OEMS unit is 15 clock cycles.

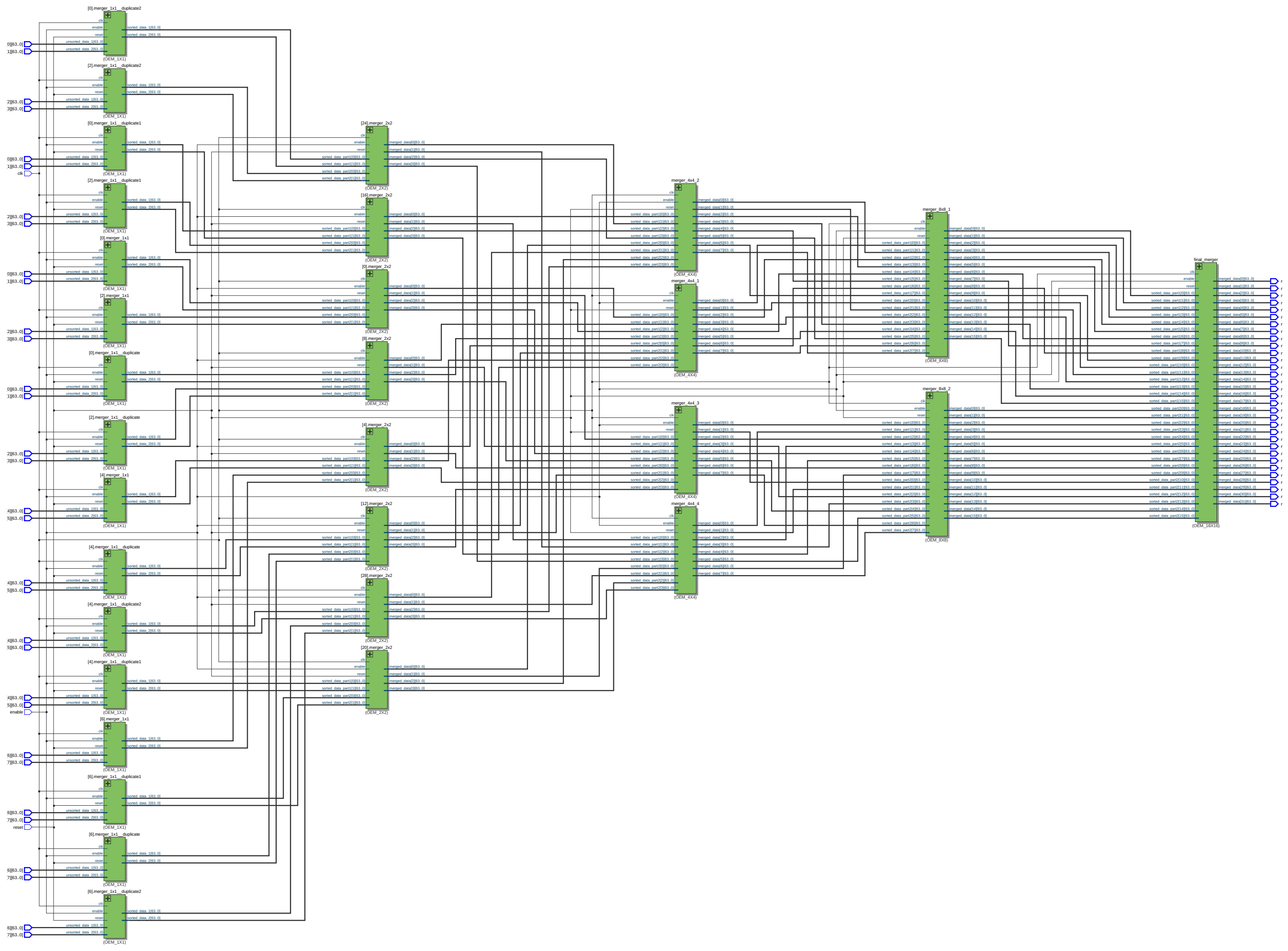
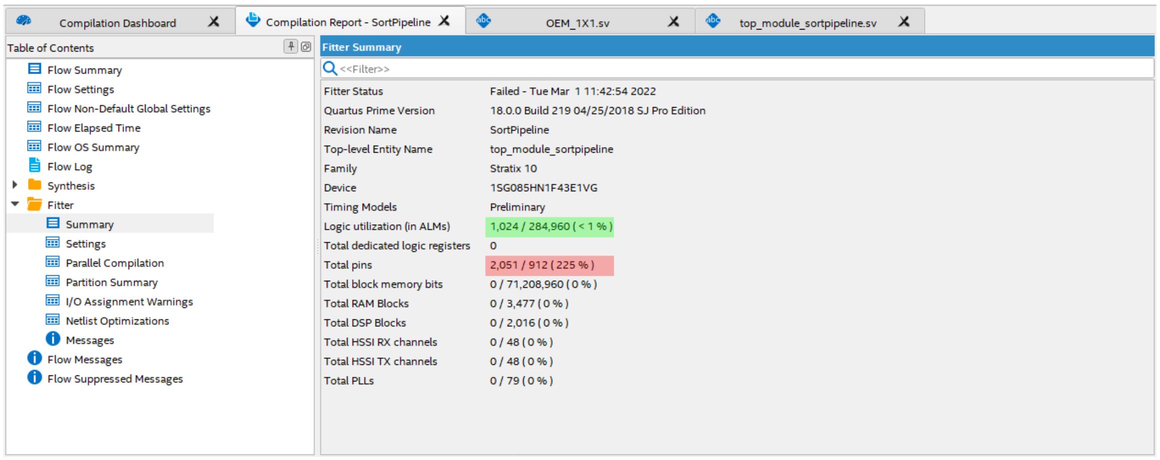


Figure 7.6: Hardware implementation of a 32x32 OEMS.

2.2 Hardware usage

Figure 7.7 shows the logic utilization of the Sort Controller unit (highlighted in green), the design is disconnected from the total design and therefore the total I/O pins (highlighted in red) is not an accurate representation. The logic utilization of the Sort Controller unit is 1.024 ALMs (*Adaptive Logic Modules*).

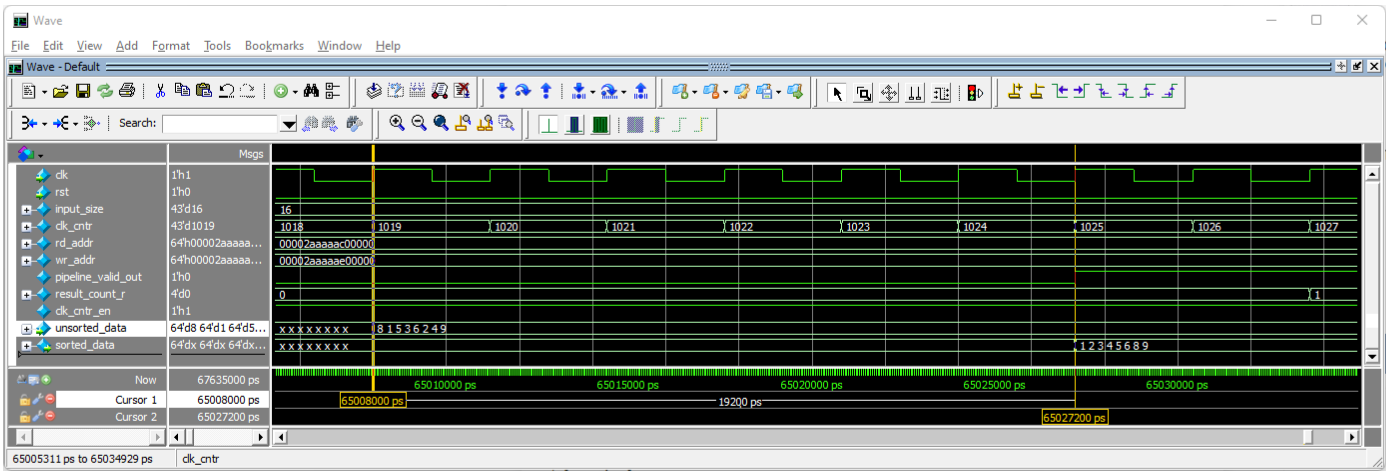


Fitter Status	Failed - Tue Mar 1 11:42:54 2022
Quartus Prime Version	18.0.0 Build 219 04/25/2018 SJ Pro Edition
Revision Name	SortPipeline
Top-level Entity Name	top_module_sortpipeline
Family	Stratix 10
Device	1SG085HN1F43E1VG
Timing Models	Preliminary
Logic utilization (in ALMs)	1,024 / 284,960 (< 1 %)
Total dedicated logic registers	0
Total pins	2,051 / 912 (225 %)
Total block memory bits	0 / 71,208,960 (0 %)
Total RAM Blocks	0 / 3,477 (0 %)
Total DSP Blocks	0 / 2,016 (0 %)
Total HSSI RX channels	0 / 48 (0 %)
Total HSSI TX channels	0 / 48 (0 %)
Total PLLs	0 / 79 (0 %)

Figure 7.7: Logic utilization of the Sorting Controller unit (Post Fitter).

2.3 Simulation results

Figure 7.8 shows the simulation result of the sorter unit (32x32 OEMS). An array consisting of eight 64 bit unordered values is provided at *unsorted_data*. After 15 clock cycles the result is propagated to the output *sorted_data* and *pipeline_valid_out* is set to high.



	Msgs
dk	1'h1
rst	1'h0
input_size	43'd16
dk_cntr	43'd1019
rd_addr	64'h00002aaaaa...
wr_addr	64'h00002aaaaa...
pipeline_valid_out	1'h0
result_count_r	4'd0
dk_cntr_en	1'h1
unsorted_data	64'd8 64'd1 64'd5...
sorted_data	64'dx 64'dx 64'dx...
Now	67635000 ps
Cursor 1	65008000 ps
Cursor 2	65027200 ps

Figure 7.8: Simulation of the Sort Controller unit

3 Iterative Merger unit

3.1 Hardware implementation

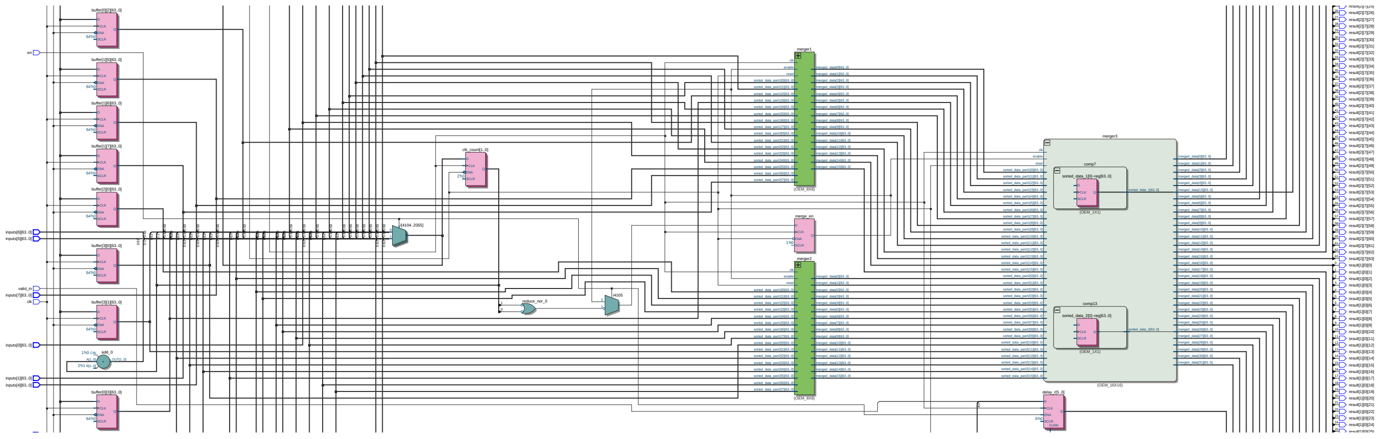
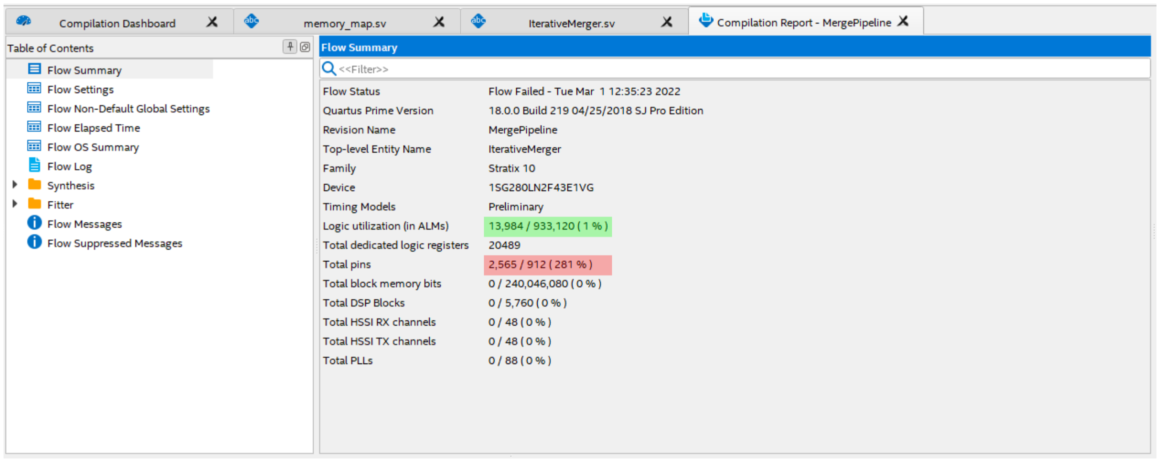


Figure 7.9: Hardware implementation of the Iterative Merger unit.

3.2 Hardware usage

Figure 7.10 shows the logic utilization of the Iterative Merger unit (highlighted in green), the design is disconnected from the total design and therefore the total I/O pins (highlighted in red) is not an accurate representation. The logic utilization of the Iterative Merger unit is 13.984 ALMs (*Adaptive Logic Modules*).

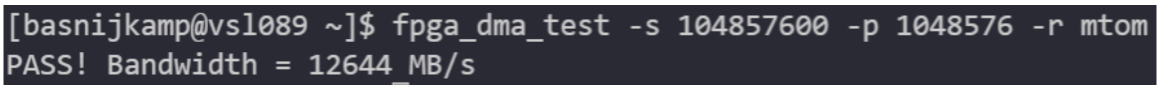


Flow Status	Flow Failed - Tue Mar 1 12:35:23 2022
Quartus Prime Version	18.0.0 Build 219 04/25/2018 SJ Pro Edition
Revision Name	MergePipeline
Top-level Entity Name	IterativeMerger
Family	Stratix 10
Device	1SG280LN2F43E1VG
Timing Models	Preliminary
Logic utilization (in ALMs)	13,984 / 933,120 (1 %)
Total dedicated logic registers	20489
Total pins	2,565 / 912 (281 %)
Total block memory bits	0 / 240,046,080 (0 %)
Total DSP Blocks	0 / 5,760 (0 %)
Total HSSI RX channels	0 / 48 (0 %)
Total HSSI TX channels	0 / 48 (0 %)
Total PLLs	0 / 88 (0 %)

Figure 7.10: Logic utilization of the Iterative Merging unit (Post Fitter).

3.3 Performance of the High Throughput Sort Accelerator

In Section 4.2: Bandwidth measurement, a bandwidth measurement is given of the system with of a optimized implementation of the data transfer between the memory of the FPGA, host memory, and the FPGA.



```
[basnijkamp@vsl089 ~]$ fpga_dma_test -s 104857600 -p 1048576 -r mtom
PASS! Bandwidth = 12644 MB/s
```

Figure 7.11: Bandwidth between host memory and FPGA using *Direct Memory Access (DMA)*

Figure 7.11 shows a bandwidth measurement of 12.644 GB/s using *Direct Memory Access (DMA)*. In Figure 7.8 you can see that a initial setup of the Odd Even Merge Sort Network takes 6 clock cycles and then produces 32 64 bit outputs every clock cycle. The sorter operates using the 250 MHz internal clock for synchronization. This gives us a bandwidth of 64 GB/s (see equation 7.1)

$$\begin{aligned}
&\frac{1}{250\text{ MHz}} = 4\text{ ns}\\
&32 \cdot 64\text{ bit} = 2048\text{ bits} = 256\text{ Bytes}\\
&\frac{256\text{ Bytes}}{4\text{ ns}} = 64\text{ GB/s}
\end{aligned} \tag{7.1}$$

The bandwidth between host memory and FPGA (12.644 GB/s) is lower than the bandwidth of sorting unit (64 GB/s). Therefore, the bandwidth of the sorting accelerator is limited by the bandwidth of this data transfer.

The first phase of the sorting accelerator is to transfer data from host memory to the sorting unit on the FPGA and storing the presorted blocks into the local memory of the FPGA. This results us in a data rate of 12.644 GB/s for phase 1 of the sorting accelerator. It takes 2.52 seconds seconds to fill the local FPGA memory with 32GB of data (see equation 7.2).

$$\frac{32\text{ GB}}{12.644\text{ GB/s}} = 2.52\text{ s} \tag{7.2}$$

The second phase of the sorting accelerator is to merge the presorted blocks in local FPGA memory iteratively and write this data back to the host memory once the complete merge has been executed.

The FPGA has four local memory banks, each local memory bank of the FPGA has 8 GB of DDR4 Memory that operates at 1200 MHz (2400 MT/s). The data width of this memory is 72 bit (64 bit for data and 8 bit ECC (*Error Correction Code*)). The effective bandwidth is $2400\text{ MT/s} \cdot 64\text{ bit} = 15.360\text{ GB/s}$. This available bandwidth is higher than the bandwidth of the data transfer between host memory and the FPGA (12.644 GB/s), which is currently limiting the throughput of the Sorting Accelerator. The Iterative merger operates at 250 MHz, it reads every presorted set of data and merges them iteratively.

The full iterative merge takes $\frac{32\text{ GB}}{256\text{ B}} = 1.25 \cdot 10^8$ iterations. Each clock cycle takes $1/250\text{ MHz} = 4\text{ ns}$. The total time to complete the iterative merge is $1.25 \cdot 10^8 \cdot 4\text{ ns} = 0.5\text{ s}$. This results in a throughput of $\frac{32\text{ GB}}{0.5\text{ s}} = 64$ GB/s. However, the memory available memory bandwidth is slightly slower (4 memory banks $\cdot$ 15.360 GB/s = 61.44 GB/s) than the maximum merge throughput and therefore limiting this throughput. The adjusted time to complete the iterative merge is: $\frac{32\text{ GB}}{61.44\text{ GB/s}} = 0.521$ s. The data can be prepared for the next merge by reading from memory simultaneously with execution of the merging operation. This preparation adds a small startup time for the first merge but this is so small that we consider it negligible. This results in a total throughput of 5.754 GB/s (see equation 7.3).

$$\begin{aligned} &2.52\text{ s} \cdot 2 + 0.521\text{ s} = 5.561\text{ s} \\ &\frac{32\text{ GB}}{5.561\text{ s}} = 5.754\text{ GB/s} \end{aligned} \tag{7.3}$$

Chapter 8

Discussion

In this chapter a discussion will be made to interpret the results given in Chapter 7. A comparison between the synthesised hardware and the predicted hardware usage is given. Furthermore, a comparison of the measurements is explained and a conclusion about the performance of the sorting accelerator is derived.

1 Implementation of hardware components

In Chapter 4.3 a theoretical explanation of Odd Even Merge (OEM) and Odd Even Merge Sort (OEMS) networks is given. With equation 4.2 we can calculate the amount comparators needed for given size N of the network. In Figure 7.1 an OEM of size $N = 1$ is shown. If we fill this in into the equation we get a comparator count of 1 (see calculation 8.1).

$$\begin{aligned} &CC(2N) = 1 + N \cdot log_2(N), N = 1 \\ &CC(2 \cdot 1) = 1 + log_2(1) \\ &CC(2) = 1 \end{aligned} \tag{8.1}$$

In figure 7.2 an OEM of size $N = 2$ is shown. If we fill this in into the equation we get a comparator count of 3 (see calculation 8.2).

$$\begin{aligned} &CC(2N) = 1 + N \cdot log_2(N), N = 2 \\ &CC(2 \cdot 2) = 1 + 2 \cdot log_2(2) \\ &CC(4) = 3 \end{aligned} \tag{8.2}$$

The figure shows the usage of 3 comparators, the synthesized hardware matches the theoretical comparator count.

In figure 7.3 an OEM of size $N = 4$ is shown. If we fill this in into the equation we get a comparator count of 9 (see calculation 8.3).

$$\begin{aligned} &CC(2N) = 1 + N \cdot log_2(N), N = 4 \\ &CC(2 \cdot 4) = 1 + 4 \cdot log_2(4) \\ &CC(8) = 9 \end{aligned} \tag{8.3}$$

The figure shows the usage of 9 comparators, the synthesized hardware matches the theoretical comparator count.

In figure 7.4 an OEM of size $N = 8$ is shown. If we fill this in into the equation we get a comparator count of 25 (see calculation 8.4).

$$\begin{aligned} &CC(2N) = 1 + N \cdot log_2(N), N = 8 \\ &CC(2 \cdot 8) = 1 + 8 \cdot log_2(8) \\ &CC(16) = 25 \end{aligned} \tag{8.4}$$

The figure shows the usage of 25 comparators, the synthesized hardware matches the theoretical comparator count.

In figure 7.4 an OEM of size $N = 16$ is shown. If we fill this in into the equation we get a comparator count of 65 (see calculation 8.5).

$$\begin{aligned} &CC(2N) = 1 + N \cdot log_2(N), N = 16 \\ &CC(2 \cdot 16) = 1 + 16 \cdot log_2(16) \\ &CC(32) = 65 \end{aligned} \tag{8.5}$$

The figure shows the usage of 65 comparators, the synthesized hardware matches the theoretical comparator count.

2 Analysing Sort Control unit

In chapter 3.4 we discuss how many execution steps the implemented design should take. We can calculate the amount of execution steps of the 32x32 OEMS using equation 4.3.

$$\begin{aligned} &T(N) = \frac{log_2(N) \cdot (log_2(N) + 1)}{2}, N = 32 \\ &T(32) = \frac{log_2(32) \cdot (log_2(32) + 1)}{2} \\ &T(32) = \frac{5 \cdot (5 + 1)}{2} \\ &T(32) = 15 \end{aligned} \tag{8.6}$$

Filling in the equation with N=32 gives us 15 execution steps (see calculation 8.8).

Figure 7.6 shows the implementation of the most important part of the Sort Control Unit. This 32x32 *Odd Even Merge Sort* (OEMS) network consists of 16 1x1 OEM, 8 2x2 OEM, 4 4x4 OEM, 2 8x8 OEM and 1 16x16 OEM. The execution time of this unit is determined by the sum of its subcomponents (see equation 8.9).

$$\begin{aligned} &T(32) = T_{1x1} + T_{2x2} + T_{4x4} + T_{8x8} + T_{16x16} \\ &T(32) = 1 + 2 + 3 + 4 + 5 \\ &T(32) = 15 \end{aligned} \tag{8.7}$$

The result in Equation 8.9 matches the calculated time in Equation 8.8. Therefore the generated implementation equals the expected implementation.

3 Analysing Iterative Merger unit

In chapter 3.4 we discuss how many execution steps the implemented design should take. We can calculate the amount of execution steps of the 32x32 OEMS using equation 4.3.

$$\begin{aligned}
T(N) &= \frac{log_2(N) \cdot (log_2(N)+1)}{2}, N = 32 \\
T(32) &= \frac{log_2(32) \cdot (log_2(32)+1)}{2} \\
T(32) &= \frac{5 \cdot (5+1)}{2} \\
T(32) &= 15
\end{aligned} \tag{8.8}$$

Filling in the equation with N=32 gives us 15 execution steps (see calculation 8.8).

Figure 7.6 shows the implementation of the most important part of the Sort Control Unit. This 32x32 *Odd Even Merge Sort* (OEMS) network consists of 16 1x1 OEM, 8 2x2 OEM, 4 4x4 OEM, 2 8x8 OEM and 1 16x16 OEM. The execution time of this unit is determined by the sum of its subcomponents (see equation 8.9).

$$\begin{aligned}
T(32) &= T_{1x1} + T_{2x2} + T_{4x4} + T_{8x8} + T_{16x16} \\
T(32) &= 1 + 2 + 3 + 4 + 5 \\
T(32) &= 15
\end{aligned} \tag{8.9}$$

The result in equation 8.9 matches the calculated time in equation 8.8. Therefore the generated implementation equals the expected implementation.

4 Interpreting the measurements

4.1 Comparing the performance to existing similar solutions

The measurements give an insight of the performance of the system. Using these results we can compare our solution to existing systems. The sorting accelerator has the most similarities with the system described in *A High Performance FPGA-Based Sorting Accelerator with a Data Compression Mechanism* [13], when comparing the throughput with this system we see that the throughput of the developed sorting accelerator is higher than this existing solution. However, we can't make a good comparison if we only focus on the throughput.

Table 8.1 shows a hardware comparison between the High Throughput Sorting Accelerator (HTSA) and the existing sorting solution.

	High Throughput Sorting Accelerator	**Existing sorting solution** [13]
Throughput	5.574 GB/s	3.200 GB/s
Operating frequency	250 MHz	200 MHz
Bandwidth host memory to FPGA (PCIe)	12.644 GB/s	3.20 GB/s
PCIe Version	3.0 x16	2.0 x8
Available memory	32 GB	4 GB
Memory speed	1200MHz	800 MHz
FPGA hardware	Intel Stratix 1SX280HN2F43E2VG	Xilinx Virtex-7 XC7VX485T
Data compression	-	x

Table 8.1: A comparison between the High Throughput Sorting Accelerator (HTSA) and an existing sorting accelerator

When reviewing the comparison we can see that one property can be differentiated. The PCIe bandwidth difference between both systems is significant. The HTSA uses a newer version of the PCIe bus with double the amount of lanes compared to the other solution. Both implementations conclude that the PCIe bandwidth is limiting factor of the sorting system. Both proposed hardware solutions can sort at a high throughput but are slowed down by the duration it takes to transfer data from host memory to the FPGA and vice versa. The existing sorting solution reduces this limitation slightly by applying a compression algorithm to the data before transferring it to the FPGA. This compression increases the throughput.

Chapter 9

Conclusion

In this thesis, we investigated the feasibility of making a sort accelerator on the Intel HARP platform. A theoretical design was made, implemented and partially tested. The Intel HARP platform provides numerous tools and abstractions to make the development of an AFU easier. However, I found that in practice these abstractions and the platform itself are not ready to be used in a production environment. The base of the platform seems stable but is quite advance to use. Intel provides a set of Building Blocks separately which include functionality which feel like basic functionality for anyone who has limited experience in designing a CPU/FPGA co-design. These building blocks are not bug-free and while being open-source and updated more frequently than the rest of the platform it makes developing cycle for an inexperienced developer unnecessarily difficult. Documentation and examples do exist but are limited. The platform can benefit if more people start using it. But, in order to achieve a larger user base, the platform must put more effort making the platform easily accessible by providing more examples and tutorials focused on clear use cases.

Is it possible to develop a high performant sorting accelerator on the *Intel Hardware Accelerator Research Program (HARP)* platform?

Yes, Although the platform is in early development it shows a great potential in harvesting the power of configurable logic in combination with big scale solutions like datacenters. A powerful property of this hardware is that its packaged in a way that fits into a PCIe expansion slot. This method is a standardized form factor which can be implemented in datacenters without changing the infrastructure.

How does the performance compare to existing sorting solutions?

The performance of the sorting solution is faster than some other solutions, but this is mostly due to the given hardware in combination with the high bandwidth of the PCIe link with the host system. This communication link is often the limiting factor in the sorting accelerator solutions.

What is the benefit of using an accelerator card for sorting instead of a more traditional solution?

An accelerator card can be configured with a hardware design that is more efficient in doing an equivalent task on the GPP/CPU. However, most of the dedicated hardware solution come with the cost of losing flexibility. Flexibility here means not able to repurpose the resource for something else when the hardware is not in use. An accelerator card creates a bridge between these problems, on one side it is flexible since it has the ability of being reconfigured on the go and on the other side it can be configured with an efficient hardware design to perform a specific task like sorting.

How feasible is it to develop a sorting accelerator using the *Intel Hardware Accelerator Research Program (HARP)*?

Are the hardware and tools made available through the platform ready to be used in a production environment?

At the start of this research there were some bugs in the platform which made it difficult to use and recommend. However, during this research the platform evolved into a state that can be used in a production environment. A clear recommendation for this platform is to provide more documentation and examples. One of the goals of this platform is to make it easier for people less experienced in FPGA design to make accelerators. With the current state of documentation the platform has a steep learning curve.

Chapter 10

Future work

- The implementation of sorting accelerator can be improved on some key areas. Since only a small percentage of the available reconfigurable area is being used there is room for additional hardware to optimise the accelerator. A suggestion could be to add compression/decompression hardware in order to increase the bandwidth between the CPU memory and the FPGA.
- Another possible improvement to make more use of the provided embedded on chip memory (M20K, MLAB). This memory is faster than the locally attached DDR4 memory and can be used to cache sorting results.
- An interesting follow-up research could be to incorporate the *High Throughput Sorting Accelerator* into existing software like a SQL database. When integrating this into real-world use cases a more elaborate performance evaluation can be made between systems with and without this accelerator.

List of Figures

List of Tables

Bibliography

[1] Gustavo Alonso. "FPGAs in Data Centers". In: *Queue* 16 (2018), pp. 52–57.

[2] *Application-specific integrated circuit.* Nov. 2021. URL: https://en.wikipedia.org/wiki/Application-specific_integrated_circuit.

[3] *Field-programmable gate array.* Nov. 2021. URL: https://en.wikipedia.org/wiki/Field-programmable_gate_array.

[4] *FPGA Accelerators.* URL: https://wiki.intel-research.net/FPGA.html.

[5] *FPGA Interface Manager (FIM).* Dec. 2019. URL: https://www.intel.com/content/www/us/en/programmable/documentation/buf1506187769663.html#wve1549400075029.

[6] *FPGA Interface Manager Data Sheet: Intel FPGA Programmable Acceleration Card D5005.* URL: https://www.intel.com/content/www/us/en/docs/programmable/683858/current/overview.html.

[7] *FPGA Interface Unit (FIU).* Dec. 2019. URL: https://www.intel.com/content/www/us/en/docs/programmable/683193/current/for-fiu-for-intel-fpga-pac.html.

[8] *High Level Design.* URL: https://www.xilinx.com/products/design-tools/vivado/implementation/partial-reconfiguration.html.

[9] *Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual.* Dec. 2019. URL: https://www.intel.com/content/www/us/en/programmable/documentation/buf1506187769663.html#wve1549400075029.

[10] *Intel FPGA Programmable Acceleration Card D5005 Data Sheet.* Dec. 2019. URL: https://www.intel.com/content/www/us/en/programmable/documentation/cvl1520030638800.html#dqm1520031512342.

[11] *Intel® FPGA Programmable Acceleration Card D5005.* URL: https://www.intel.com/content/www/us/en/products/details/fpga/platforms/pac/d5005.html.

[12] *Introduction.* Dec. 2019. URL: https://www.intel.com/content/www/us/en/programmable/documentation/buf1506187769663.html#cdp1506203431918.

[13] Ryohei KOBAYASHI and Kenji KISE. *A High Performance FPGA-Based Sorting Accelerator with a Data Compression Mechanism.* 2017.

[14] *MMIO Accesses to I/O Memory.* Dec. 2019. URL: https://www.intel.com/content/www/us/en/programmable/documentation/buf1506187769663.html#aan1506275190752.

[15] Rene Mueller, Jens Teubner, and Gustavo Alonso. *Sorting networks on FPGAs.* June 2011. URL: https://link-springer-com.ezproxy2.utwente.nl/content/pdf/10.1007%2Fs00778-011-0232-z.pdf.

[16] *OPAE*. Oct. 2019. URL: https://01.org/opae.

[17] Opae. *OPAE/Intel-FPGA-BBB: Basic Building Blocks (BBB) for OPAE-managed Intel fpgas*. URL: https://github.com/OPAE/intel-fpga-bbb.

[18] Dirk Koch University of Oslo et al. *FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting*. Feb. 2011. URL: https://dl.acm.org/doi/10.1145/1950413.1950427.

[19] Murad Qasaimeh et al. "Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels". In: May 2019. DOI: 10.1109/ICESS.2019.8782524.

[20] *Sorting algorithm*. Feb. 2022. URL: https://en.wikipedia.org/wiki/Sorting_algorithm.

[21] *Sorting algorithms*. URL: https://www.geeksforgeeks.org/sorting-algorithms/.

[22] Roger Whitney. Feb. 1996. URL: http://www.eli.sdsu.edu/courses/spring96/cs662/notes/batcher/batcher.html.