High Throughput Sorting on FPGAs using High Bandwidth Memory

Bastiaan Feenstra

CE-MS-2020-03

Delft University of Technology

Abstract

In this thesis we explore the acceleration of sorting algorithms on FPGAs using high bandwidth memory (HBM). The target application is an FPGA as an accelerator in an OpenCAPI enabled system, that enables the accelerator to access main memory of the host at a bandwidth of 25 GB/s for either read or write. We explore under what read and write access patterns the HBM bandwidth of 460.8 GB/s can be met and identify specific circumstances under which this bandwidth can be achieved. The sorting algorithm is implemented in hardware as two steps: partitioning and sorting. We design two partitioning architectures and one sorting architecture. The sorting architecture sorts buckets generated in the partitioning step and is based on merge sort. It uses HBM and wide merge trees to reduce the number of passes through a memory. The architectures themselves are to be instantiated multiple times on the FPGA to achieve a higher sorting throughput. Simulating each architecture at 225 MHz, they are all designed to output up to 3.6 GB/s of 8+8 byte key-value pairs under ideal conditions. We measure the first and second partitioning architectures and identify a bottleneck in HBM for the former, resulting in only 0.44 GB/s with a (uniformly) random input, due to a strided access pattern. With a sorted input, the throughput is 2.18 GB/s. The second partitioning architecture is not affected by this and achieves a throughput of approximately 1.7 GB/s for both types of input. The sorter performs best, sorting buckets at approximately 2.7 GB/s. Synthesis results show that the target FPGA has enough resources for tens of partitioners and sorters, allowing to create a sorting hardware that saturates system bandwidth.

TU Delft Delft University of Technology

Quantum & Computer Engineering

High Throughput Sorting on FPGAs using High Bandwidth Memory

by

Bastiaan Feenstra

to obtain the degree of Master of Science at the Delft University of Technology, to be defended publicly on Wednesday February 26, 2020 at 10:00 AM.

Student number:	4510984	
Project duration:	February 2019 – February 2020	
Thesis committee:	Dr.ir. Z. Al-Ars,	TU Delft, supervisor
	Prof.dr. H.P. Hofstee,	TU Delft
	Prof.dr. K.L.M. Bertels,	TU Delft
	Dr. J. Fang,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft Delft University of Technology

Contents

List of Figures

List of Tables

1 Introduction

This year, mankind generates more data than it has in all the years before. We are living in an ever-increasing digital world, becoming more and more connected. Self driving cars are almost a reality. Yesterdays mass broadcast television is today's individual stream. We are in the second machine age.

However, this growth is at risk of slowing down. While we enjoyed decades of Moore's law to effectively increase performance of our microchips, some say Moore's law is dead. Others say that Moore's law is slowing down. Either way, to improve the performance of an application, definitely is not a case of waiting two years until a new processor comes out anymore.

Today, we achieve more performance through paralellization. One type of processor that has been found to be specifically good at this, is the GPU. In essence, it is a processor that operates with groups of cores that are locked to the same program counter. Modern day GPUs contain thousands of such cores. At the extreme end, custom chips are developed (application specified integrated circuits, ASICs) to facilitate the compute needs. Before ASICS, we have one more candidate: field programmable gate arrays (FPGAs).

FPGAs are flexible chips that can be configured to perform any task. New interconnects (OpenCAPI) enable accelerators such as FPGAs to be placed inside a server, to have access to its memory and accelerate specific tasks. One example of a fundamental task in computer systems is the act of sorting data.

1.1. Goals

The goal of this thesis is to implement a high performance sorting algorithm on an FPGA. To do this, we look into what type of sorting algorithms are suitable for FPGA acceleration. After choosing a suitable algorithm, we explore writing a hardware implementation. We also aim to answer the question of what kind of sorting throughput can be expected when using new technologies as high bandwidth memory (HBM) and OpenCAPI. The implementation should be of high-performance to saturate the bandwidth of the interface to main memory, where the input data is assumed to reside. The target FPGA is a high-end model with 8 GB of HBM.

1.2. Contributions

We make the following contributions. Through simulation we characterize the behavior of HBM under various read and write access patterns. The sorting algorithm we select has two steps: a partitioning step and a sorting step. We design and build two different architectures for the partitioning step and one architecture for the sorting step. Each architecture is designed to output 1 input element/cycle. When implemented in hardware we expect to run multiple instances of the architectures in parallel. By using multiple instances we can saturate the bandwidth of the interface to main memory. All three architectures are designed to utilize HBM. We explore some of the parameters that exist in the implementation and analyze their effects on throughput or resources. With these results we provide helpful insights on the use of HBM.

1.3. Thesis overview

In Chapter 2 we describe the technologies that will be used and provide an overview of (some of the) sorting algorithms and describe the existing work. In Chapter 3 we take a deep-dive into the FPGA specific memory technologies and perform an early characterization of HBM. In Chapter 4 we present the designs of the different architectures and describe how they were implemented. The throughput and area results of these implementations is evaluated in Chapter 5. In Chapter 6 we conclude the project and discuss future work, including the improvements that can be made to the different architectures.

2

Background

In this chapter we begin with a look into the technologies that will be used, in Section 2.1. In Section 2.2 we describe various definitions related to sorting, explore multiple sorting algorithms, sorting networks and describe some of the existing work.

2.1. Technologies

2.1.1. FPGA

A field-programmable gate array (FPGA) is an integrated circuit whose logic can be programmed after manufacturing, hence ”field-programmable”. An FPGA consists of many logical blocks, an interconnect fabric and memory as shown in Figure 2.1. FPGAs also frequently implement some more dedicated resources such as DSP blocks, and recently even processors. The configuration of FPGAs is typically described using a hardware description language (HDL) such as VHDL or Verilog, in which the desired functionality of the FPGA is specified. It is also possible to decribe the configuration in high-level languages using high-level synthesis (HLS) tools such as Vivado HLS [13] and Altera OpenCL [21]. The processes of synthesis and implementation then translate the description and map it onto an FPGA.

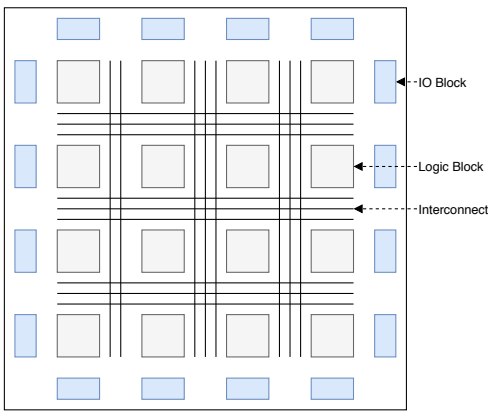


Figure 2.1: Structure of an FPGA.

CPU vs GPU vs FPGA

A traditional CPU is very sequential: a processor executes the instructions of a process one by one. Multiple processes are executed 'in parallel' by switching the active process many times per second, executing its instructions for a short while. Admittedly, modern consumer CPUs contain multiple cores that work in parallel, yet the number of processes and threads that are executed in such systems is much higher. Certain instructions in CPUs have also been added to perform the same instruction on multiple data in parallel (SIMD-instructions). GPUs can also be seen as wide SIMD processors, with for example 32 SIMD 'lanes'. Although an FPGA could be programmed to do the same task as a CPU of executing instructions sequentially, it is intrinsically parallel. With the ability to program the behavior of the entire FPGA, it can perform many different tasks truly in parallel.

2.1.2. HBM

High Bandwidth Memory is a relatively new form of memory. It was adopted by JEDEC as standard JESD235 in October 2013 [2]. In June 2015 AMD released the R9 Fury X, the first consumer graphics card with 4 GB of HBM. In Figure 2.2 this is shown, a large GPU die in the center with four stacks of HBM surrounding it. AMD and SK Hynix developed HBM to create a new type of memory that requires less space, less energy and provides more bandwidth than the traditional memories. Today the are found in high-end consumer GPUs, data center GPUs, high-end FPGAs and other high-performance applications. In terms of interface, the traditional approach of DDR memory interfaces is fast and narrow, where the approach of HBM is to have a slow but wide interface.

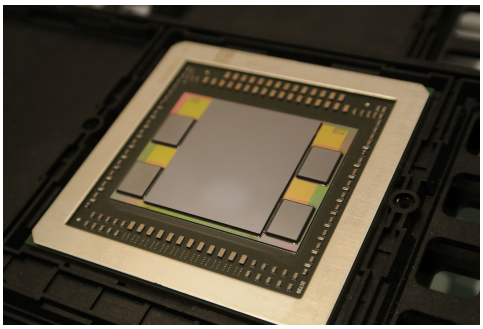


Figure 2.2: AMD Fiji package. 4 stacks of HBM and a GPU on an interposer [23].

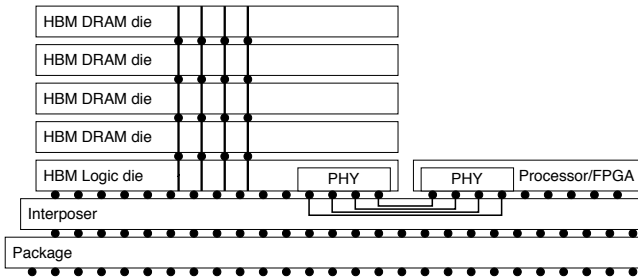


Figure 2.3: Schematic side view of a 4Hi HBM stack on a package.

HBM stacks multiple HBM DRAM dies vertically as shown in Figure 2.3. The dies are interconnected by through-silicon vias (TSVs) and microbumps. The bottom layer of the HBM stack is a logic layer[1]. The HBM stack and processor (shown as logic die here) are both mounted on an interposer. This interposer is a large silicon die with no logic, providing connections between HBM and processor. This is also known as a 2.5D configuration. The HBM2 specification lists multiple configurations: 2, 4, and 8 layers high[2] for a memory capacity of 2, 4 and 8 GiB with 8 Gib dies.

HBM1 has a maximum bandwidth of 128 GB/s, while the maximum bandwidth of HBM2 is 256 GB/s when it was introduced. HBM operates in one of two modes: legacy mode or

[1]The logic layer is optional in terms of the HBM specification.
[2]A typical notation for a 4-layer high stack is 4Hi, with an implicit (optional) logic layer.

pseudo channel mode. Legacy mode refers to the HBM1 specification, as HBM2 introduced the pseudo channel mode. In legacy mode, each channel is a fully independent interface to a specific part of the memory. The data width is 128 bits. In pseudo channel mode, each memory channel is split in half. The address and command bits of the pseudo channels are shared but the data interface is split into two 64 bit channels. Read and write transactions have a fixed burst length of 4 (BL4) providing 256 bits per transaction. In legacy mode the burst length is 2 (BL2), providing the same 256 bits per transaction.

In December 2018, the HBM standard was updated to support 16 Gib dies (up from 8 Gib) and additional height options of 12 and 16, the latter not at full bandwidth. This update also increased the per-pin bandwidth from 2 Gb/s up to 2.4 Gb/s, for a total of 307 GB/s [4]. Later in August 2019, SKHynix announced HBM2e which increases the per-pin bandwidth up to 3.6 Gb/s [5]. A ”Low-Cost HBM” also appears to be in development for more mass-market products. It is expected to offer a lower cost, at the loss of ECC and some bandwidth [15].

2.1.3. AXI

The ARM Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) protocol is an interconnect protocol introduced in 2003 with the AMBA 3 AXI specification [1]. It is targeted at high-performance, high-frequency designs and is used to connect modules, devices and peripherals in embedded systems.

The AXI protocol uses a basic handshake mechanism with valid and ready signals. A sender that produces some data asserts the valid signal when it has a valid signal at its output. The receiver asserts ready when it is ready to receive the data. Data is exchanged when both the valid and ready signals are asserted. The specification specifically mentions two rules regarding the valid and ready signals. First, when valid is asserted, it must remain so until the receiver (slave) asserts the ready signal. Second, the valid signal must not depend on the ready signal. The ready signal can wait for the assertion of valid.

One AXI3 interface has 5 channels. All of them use this handshaking mechanism:

- Write address channel
- Write data channel
- Write response channel
- Read address channel
- Read data channel

As could be inferred from this list, the write address and data can be issued separately. The write data can be issued before the write address and vice versa.

In the AXI protocol, a transaction can consist of multiple transfers: the data may be sent over multiple cycles. This is also referred to as the burst length or the number of beats in a transaction. The AWLEN and ARLEN signals specify this and can be set to 1 up to 16. The master must complete all transfers in the burst during a write transaction. During a read transaction, the master can disregard the read data but all transfers in the burst must be received. The master must assert the WLAST signal on the last transfer of a write transaction.

How data in a burst is written to the memory is defined by the burst signals ARBURST and AWBURST. Three modes exist:

- The first mode is a fixed-address burst, where the address remains constant. This mode is for access to the same memory, corresponding to a FIFO.
- The second mode is for normal sequential memory, where the write address is incremented automatically.
- The third mode is a wrapping type, which increments the address until it wraps to a lower address at the wrap boundary.

The size of each transfer in a burst is defined by signals ARSIZE and AWSIZE for reads and writes respectively. These can have values that are a power of 2, up to 128 bytes. While the minimum number of bytes in a transaction is defined by this signal, writing data in smaller quantities is possible. The write strobe (WSTRB) signal is a per-byte write-enable.

The AXI protocol supports out-of-order transactions. A set of transactions is out-of-order if they complete in a different order than they were submitted. Each read or write transaction is accompanied by an ID, that can be seen as a virtual channel. Transactions with different IDs can complete out of order. Transactions with the same ID complete in order. Transactions that are issued from different interfaces have no ordering restrictions.

The protocol does not define ordering restrictions between read and write transactions with the same ID. Therefore, the master must wait for the write acknowledge signal before a read is issued related to that write. If the master wants to read old data and write new data to an address, the master must wait for the read data to arrive before issuing the write transaction.

The SNAP framework, described in Section 2.1.5, uses the AXI protocol as its interface. It makes use of the next versions: AXI4 and AXI4-Lite. AXI4 introduces some changes and removed some signals that were not discussed [1]. AXI4-Lite is a simplified version of the AXI4 interface for devices that do not need the full functionality of AXI4.

2.1.4. OpenCAPI

The Open Coherent Accelerator Processor Interface (OpenCAPI) is an interface to connect accelerators or advanced memories to a host system[25]. It provides a high bandwidth, low latency interconnect allowing an accelerator to have direct access to memory of the host system.

In the classical situation, a system with an FPGA attached as an accelerator, named Attached Functional Unit (AFU), transfers data as follows. Assume some software running on the host CPU has some data that it needs to send to the AFU. To transfer this data the software needs to communicate to the device driver and copy the data into the device driver memory. The device driver can then move data from its memory to the AFU. When the AFU is done processing the data, any output data is written to device driver memory. Finally, the output data is then copied to the software memory. Therefore, there is a lot of CPU and memory overhead involved in this system.

This is different in an OpenCAPI enabled system. The software only needs to signal to the AFU where the data is located in memory and how much. Because the CPU and AFU

share the same memory space, the AFU itself issues reads to the host memory and accesses the data directly. The virtual to physical address translation is performed on the CPU which reduces the complexity of the accelerator. Additionally, because the AFU works in the virtual address space, the security of the memory is enforced. The AFU does not have access to memory of the kernel or that of other applications.

Previous versions (CAPI 1.0 and CAPI 2.0) were built on top of PCI-E version 3.0 and 4.0 respectively [26]. With OpenCAPI (also referred to as CAPI 3.0) the OpenCAPI Consortium moved away from PCI-E and now uses 25G links.

2.1.5. SNAP

SNAP, short for Storage Network and Analytics Programming, is a framework to develop FPGA-based accelerators utilizing (Open)CAPI. Currently published versions of SNAP support CAPI 1.0 and CAPI 2.0, but a separate OpenCAPI version of SNAP should be on its way. Furthermore, the interfaces of SNAP for CAPI 1.0, CAPI 2.0 or OpenCAPI remain the same.

In SNAP terminology, an application is a program running on the host CPU. An action is a program running on an FPGA. SNAP can abstract not just host memory, but also on-FPGA DDR or flash and NVMe storage in the form of the AXI4 interface. SNAP itself has the logic to interface with these different resources, such that the programmer only has to work with AXI4 interfaces.

SNAP provides an API for C/C++ code which enables the transfer of data such as an input and output memory address to the accelerator, and the API for an accelerator to interface with main memory. Aside from Verilog and VHDL, SNAP also supports High Level Synthesis (HLS): C-code which is synthesized into register-transfer level (RTL) code in an HDL. With the Power Service Layer Simulation Engine (PSLSE) an action can be simulated without an FPGA and on non POWER systems [28].

The word size for SNAP actions is 64 bytes. Host memory access is done through 128 byte cache lines. When a write of less than 2 words is issued to host memory, the memory needs to be read before being written. This is a less efficient memory access compared to one where two full words are written.

2.2. Sorting

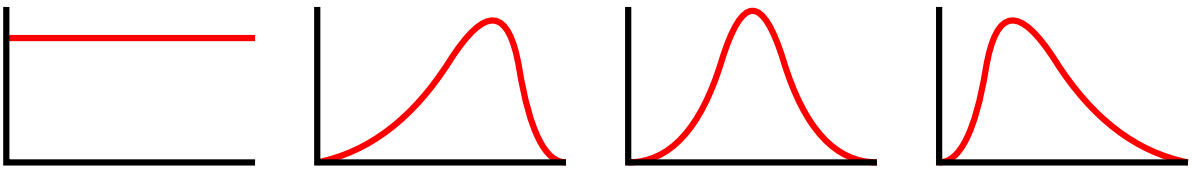
The act of sorting is to arrange elements in a certain order. In a typical sorting operation, the elements are a set of integers that should be arranged in ascending order. The number on which the order is defined is named the key. This number is typically related to some other information. For example, if we view a folder with images on a computer. Most cameras store images as a number that is incremented every subsequent photo. To sort the images we need to move the images according to this key however, physically moving images in order would be inefficient. To resolve this, the key is often accompanied by another value, which can be the pointer to a memory address or the index into an input array. In this case, this can be the pointer to the location of the image. Therefore, an 'element' may refer to a key, or a tuple consisting of the key and some value.

If the sorting algorithm preserves the order of elements that are considered equal, as they appear at the input, the sorting algorithm is said to be stable. An in-place sorting algorithm is a sorting algorithm that swaps elements in the input when sorting. As such it needs no, or very little constant, additional memory for sorting. An online algorithm is an algorithm that can process the input when it is given the elements one by one. An offline algorithm must be given the entire input at once to function.

With internal sorting, the number of elements to be sorted is small enough such that the sort can be performed within main memory. Conversely, with external sorting, the number of elements is too large to hold in the main memory. In this case, the data will be partially stored on slower memory, typically disks.

Comparison based sorting algorithms compare elements of the input with each other to perform the sort. In non-comparison based algorithms, elements of the input are not compared to each other. Two examples of non-comparison based sorting algorithms will be described in Section 2.2.1: counting sort and radix sort.

The distribution of a set of elements refers to how elements are located in an interval. In a uniform distribution the chance to observe a specific value is equally likely for all numbers. For example, when a fair dice is thrown, the chance of throwing a 1,2,3,4,5 or 6 is equally likely. If we throw this dice many times and write the result in an array, then this (with very high probability) becomes a uniform distribution.



(a) Uniform. (b) Negatively skewed. (c) Normal distribution. (d) Positively skewed.

Figure 2.4: Histogram of various distributions.

Another common distribution is one that looks like a bell curve: a normal distribution. As shown in Figure 2.4c, with a normal distributions most of the values lie close to the mean. Sometimes the data is skewed. We have two types of skew: negative and positive skew. This skew refers to the "tail" in the graph, not the skew of the curve itself. With a negative skew most of the data is distributed on the right side, as shown in Figure 2.4b. With a positive skew most of the data is distributed on the left size, as shown in Figure 2.4d.

2.2.1. Sorting algorithms

Insertion sort
One of the most simple approaches to sorting an input is insertion sort. With insertion sort, elements are considered one at a time. Starting with a list of 1 element, the next input is compared to each element of the sorted list until a larger value is found. It is inserted at this position by then moving all the elements that come after it down by one position.

Binary insertion sort
A modification of the insertion sort is the binary insertion sort. In insertion binary sort the element is compared to the center element of the list. If we are sorting in ascending order, and the element is less than the center, we evaluate the center of the first half (the first quartile). If it is larger than the center, we evaluate the center of the second half (the third quartile). This is performed recursively until the insertion position is found. Then, any elements that come after it are pushed down one position. This way the number of comparisons can be reduced.

Counting sort
The counting sort algorithm is a non-comparison based sorting algorithm for keys that are small integers. The input is enumerated and for each possible input value, a specific counter is incremented. A prefix sum over all counters is then performed. The result of this prefix sum indicates the starting positions of keys with some value. The input is again enumerated and the keys can be written to the output one by one. The position is indicated by the corresponding result of the prefix sum. It is incremented when it is used to write a key to that position.

Selection sort
In selection sort, the lowest element in the (remaining) input is found by enumerating it, repeatedly. After every enumeration, the lowest element is written to the output list.

Bubble sort
Bubble sort repeatedly iterates through the input, comparing (and swapping) every adjacent pair of elements until the input is sorted. After every iteration, the last element that took part can be excluded from the next iteration because the maximum value of that iteration is pushed to the rightmost position.

If the input to the bubble sort algorithm is sorted this can be detected in the first iteration. In this case, only $n - 1$ comparisons are made.

Shell sort
The shell sort algorithm sorts different subsets until the input is sorted. In the first pass, the subsets are created by selecting elements with a relatively large gap. For example, with input elements $a_i \in A$ and a gap of 5 one subset is the set of $(a_1, a_6, a_{11}, ...)$. The second subset in the same pass is $(a_2, a_7, a_{12}, ...)$. The next passes use a smaller gap. Eventually, after the gap of the pass is 1, the input becomes sorted.

Merge sort
Merge sort works by repeatedly merging sequences into larger sequences until the entire input is sorted. For example, with 16 inputs, we begin by sorting every adjacent pair. This results in 8 sorted subsets of length 2. These are then merged into 4 subsets of length 4, then 2 of length 8 and finally 1 set of length 16.

Radix sort
The radix sort algorithm works by evaluating the individual digits of the keys in the input. These can be bits, numbers and characters for strings. In the first round, the keys are put into buckets corresponding to the first digit, which can be the most significant digit (MSD) or the least significant digit (LSD). In the subsequent rounds, each bucket is split again depending on the next digit. This continues until all the digits have been evaluated, or when all buckets have fewer than 2 elements in it. Then the input has been sorted. Since no comparisons are made, the radix sort algorithm is a non-comparison based sort.

Bucket sort
The bucket sort algorithm is an algorithm that splits the input into a number of buckets. Each bucket holds an exclusive range of values. For example, say we have an input set with keys ranging from 0 to 99, and 10 buckets. The first bucket will hold values from 0-9, the second holds values from 10-19, etc. Then the buckets are individually sorted again, either by another bucket sort or another sorting algorithm.

We can see that the bucket sort algorithm will work well when the input is uniformly distributed over some range. This will cause the buckets to be evenly sized, which reduces the sorting problem to a smaller one. If the input is not uniformly distributed and the entire input ends up in a single bucket, we can see that the problem is not reduced.

Quick sort
In the quick sort algorithm, a value is selected from the input, called the pivot. The input is then split based on this pivot, smaller items are put into the first subset and larger items put into the second. This happens recursively until the input is sorted.

The strategy to choose the pivot can significantly affect the sort. A worst case is for example: if the input is sorted and the last element is chosen as the pivot. In this case, every iteration reduces the problem size by one. Ideally, the problem size is halved every time, which occurs when the pivot is the median.

Sample sort
Sample sort can be seen as a generalization of quick sort (or Bucket sort). Recall that quick sort repeatedly forms two subsets by splitting the input by some value found in the input (the pivot). The sample sort algorithm uses multiple pivots named splitters instead, to split the input into multiple buckets. These buckets are then sorted by repeated application of the sample sort or by using another sorting algorithm.

Where the bucket sort algorithm assumes that the input is uniformly distributed to split the input into equally sized buckets, the sample sort algorithm does not make this assumption. Instead, the values by which the data is split are sampled from the input itself. Therefore, the bucket ranges self-adjust to the input. How well they split the input, in terms of how equally sized the buckets become, is then determined by the input itself and the sampling method.

A strategy to improve the splitters such that they better represent the input is to perform

oversampling. With some oversampling ratio k and b buckets, more samples than splitters are obtained. When the samples are sorted (and deduplicated) the samples at position $1k, 2k, 3k, \ldots, (b-1)k$ are chosen to be splitters.

To handle inputs with many duplicate values, the implementation may also introduce equality buckets. If an element is equal to a splitter, the element is put into the corresponding equality bucket. Since each equality bucket contains only equal values, these buckets do not have to be sorted.

Tree sort
The tree sort algorithm considers elements one at a time. A tree is built by inserting elements one at a time. Finally, when the entire input has been inserted, the tree is traversed in order such that the elements come out in a sorted order.

Heap sort
The heap sort algorithm works in two steps. First, the input is turned into a heap: a heap is a complete binary tree where the parent of each node has a higher or equal value (a max-heap). If all parent nodes are equal or smaller, we have a min-heap. The second step repeats until all nodes have been removed from the heap, according to the following procedure. The largest value (the root) is swapped with the last node and no longer considered part of the tree. A procedure now turns the tree into a heap again.

2.2.2. Sorting networks

A sorting network is a model of wires and comparators that are used to sort a sequence. The length of the input is bound by the number of inputs to the sorting network, which is typically in the single or low double-digit space. In sorting networks, the sequence of comparisons does not depend on the input, i.e., the structure is oblivious to the input. Only the inputs to the comparison operations are potentially swapped. Sorting networks are a method of sorting that is commonly seen in hardware implementations since the compare (and swap) operations can be implemented with relative ease in circuits. Next we introduce two definitions related to sorting networks:

- The length of a sorting network is the number of compare-and-swap operations.
- The depth of a sorting network is the number of parallel steps that the network takes before the input is sorted.

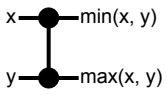


Figure 2.5: Notation of a compare (and swap) module that sorts two inputs.

In sorting networks the data flow is from left to right. Figure 2.5 shows the notation for a comparator. An alternative notation indicating the sorting direction uses an arrow pointing from $\min(x, y)$ towards $\max(x, y)$. Comparators that do not connect to the same wires in a period can execute in parallel. Designing a sorting network that is optimal in terms of its depth or length can be a difficult problem[3]. Additionally, when observing the structure of a sorting network in some cases it can be difficult to see that it works, or how it works.

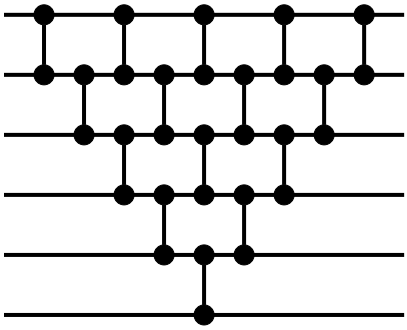


Figure 2.6: A simple sorting network for 6 inputs. 15 comparators with a depth of 9.

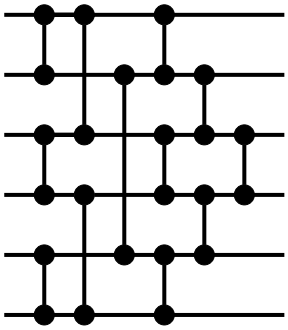


Figure 2.7: A sorting network for 6 inputs. 12 comparators with a depth of 5. [16]

Figure 2.6 and Figure 2.7 show two different sorting networks for n=6 inputs. The former is the construction for both the insertion sort and bubble sort algorithm, which result in the same network when their parallelism is fully utilized. The latter is a depth optimal network using fewer comparators and with a lower depth.

[3]Optimality refers to finding a minimum length or depth network for an n-input sorting network.

Odd-even mergesort

The odd-even mergesort is a merge network designed by K.E. Batcher [10]. It is also known as Batcher's odd-even mergesort. It merges two sorted sequences into one sorted sequence. The construction is a recursive one, as follows.

With sorted sequences $a_1, \ldots, a_m$ and $b_1, \ldots, b_m$ with $m > 0$ as a power of 2, if $m > 1$:

1. Apply odd-even mergesort to the odd subsequence $a_1, a_3, \ldots, a_{m-1}, b_1, b_3, \ldots, b_{m-1}$ and to the even subsequence $a_2, a_4, \ldots, a_m, b_2, b_4, \ldots, b_m$.
2. Add comparators between the i^{th} output of the even merge and the $i + 1^{th}$ output of the odd merge.

Else, add a comparator between the two inputs.

To merge two sorted sequences of length $m > 0$, where m is a power of 2, takes $\log_2(m) \cdot m + 1$ comparators and the depth of the merge network is $\log_2(m) + 1$.

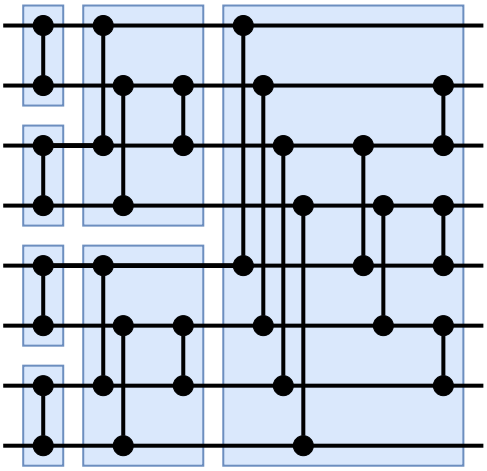


Figure 2.8: Sorting network for 8 inputs using multiple odd-even merge networks. Each highlighted area represents an odd-even merge network.

Figure 2.8 shows how multiple odd-even merge networks form a sorting network for 8 inputs. After the first set of merge networks, we obtain 4 sorted sequences of length 2. After the second set of merge networks, we have 2 sorted sequences of length 4. The final merge network merges these into 1 sorted sequence of length 8.

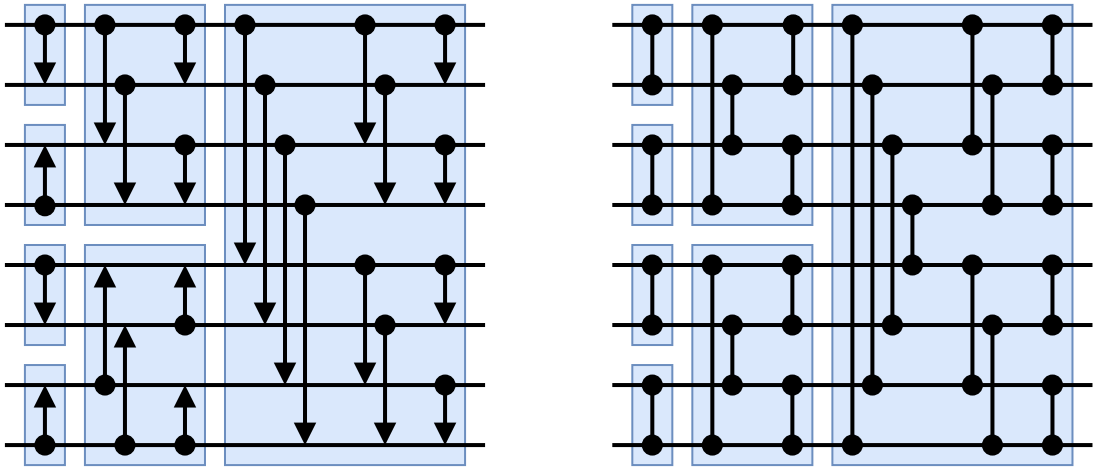
For arbitrary sequences of length n, where $n = 2^p$ for some integer $p \geq 0$, the depth of the sorting network using odd-even merge networks is $\frac{1}{2}p(p+1)$. The length of the network is $(p^2 - p + 4)2^{p-2} - 1$.

Bitonic sorter

The bitonic sorter, also known as Batcher's bitonic sort is a network that sorts a bitonic sequence. A sequence is called bitonic if it is a sequence that monotonically increases and then monotonically decreases. A cyclic shift of a bitonic sequence is also called a bitonic sentence. The construction is as follows.

With the input sequence $a_1, a_2, \ldots, a_n$ for n inputs:

1. Add comparators between the pairs $a_i, a_{i+n/2}$ for $i = 1, \ldots, n/2$.
2. Use a bitonic sorter for both the first $n/2$ and last $n/2$ outputs of these comparators when $(n/2) > 1$.



(a) In this representation the arrows indicate the sorting direction: the largest result of the comparator is placed at the arrowhead. The second of every pair of bitonic sorters sorts in reverse order, to produce a bitonic sequence for the next bitonic sorter.

(b) A unidirectional representation of the sorting network.

Figure 2.9: Two representations of a sorting network using bitonic sort. Each highlighted area represents a bitonic sort.

If we look at how the bitonic sorter sorts a bitonic sequence then we see that it takes a divide and conquer approach, where the bitonic sequence is repeatedly split into bitonic sequences of half the size. Since elements of the first half are all less than the elements of the second half, eventually the splits cause the output to be fully sorted.

To build an n-input sorting network from the bitonic sort, the bitonic sort can be used repeatedly to sort larger sequences until the output sequence is equal to n. Every pair of bitonic sorters form the input to the next bitonic sorter. Since the input needs to be bitonic, the second sorter in each pair sorts in reverse order. This is shown in Figure 2.9a. The unidirectional notation as shown in Figure 2.9b can be obtained by transforming the bidirectional network. Effectively, the first half of this sorting network works by merging until there is a single bitonic sequence. The second half, hosting the final bitonic sort, works oppositely, by dividing the problem into smaller ones repeatedly.

An advantageous property of a sorting network built from bitonic sorters is that the number of comparators in each parallel step is constant: $\frac{n}{2}$.

2.3. Existing work

2.3.1. CPU-based sorting implementations

The scalable dynamic skew-aware parallel sorting algorithm (SDS-Sort) [11] is a high performance sorting implementation for supercomputers. They achieve a sorting throughput of just under 2 TB/s with 130000 CPU cores. While they do not strictly follow the sample sort procedure, there are many similarities. They begin with the input spread over a large number of nodes, where each node may have multiple cores. Each node locally sorts the local input using the C++ standard library functions std::sort or std::stable_sort. The result of this sort is locally sampled for its splitters. Then the local splitters are sampled to form global splitters. The local data is distributed based on this split and finally merged using std::merge.

2.3.2. GPU-based sorting implementations

An implementation of sample sort on GPUs is found in [17]. With 64-bit elements for a 32-bit key and 32-bit value, they report a sorting rate of approximately 2 GB/s. The process of finding a bucket given a sample is done by traversing a search tree as described in [20]. After sampling, they sort using quicksort, allocating one thread block per bucket. The sorting of each bucket is scheduled according to the bucket size, to improve load-balancing.

In [9], also an implementation using the GPU for sample sort, the first step is performed repeatedly until each bucket can be sorted by an individual GPU thread. They increase the throughput by almost 25% over the implementation in [17].

[14] similarly partitions the input into buckets repeatedly until they obtain very small buckets that are then sorted. Using four high-end NVIDIA GPUs in parallel, and an 8-byte key 8-byte value tuple, they achieve a sorting rate of 28 GB/s.

2.3.3. FPGA-based sorting implementations

The survey in [12] summarizes the state-of-the-art FPGA-based sorting algorithms.

In [27] a merge sorter tree for up to 4096 inputs is described. It is relatively efficient in terms of area and scalability by using a single sorting cell per merge stage, with small buffers between each stage. However, the architecture can only output one 64-bit element/cycle and is thus limited to about 1 GB/s.

[29] describes an odd-even merge sorter implementation that can merge 4 streams at 27.2 GB/s.

In [22] a parallel hardware merge tree is described (PMT). At 32 64-bit elements/cycle with a frequency of 99.2 MHz they achieve a throughput of 24.6 GB/s, merging 32 sorted sequences.

[18] describes the hardware merge sorter SHMS. For an E element/cycle sorter they designed merge networks that are implemented with a pipeline of E-1 stages. Each stage in the pipeline merges E sorted elements with 1 element, to output E elements/cycle. A tree of such merge networks allows for wider mergers. Their best performing configuration merges

32 sorted sequences at 32 elements/cycle. At 311 MHz this achieves a throughput of 77.2 GB/s. They do note that the throughput of the tree is reduced when the data is not random and uniformly distributed, causing stalls. Of note here is that when increasing E, the number of levels of gates remains constant in their merge sorter.

[19] describes a very high-performance hardware implementation of a merge sorter that can merge two sorted sequences. In their highest throughput configuration, they output 32 64-bit elements/cycle, resulting in a throughput of 125 GB/s.

In [24] a hybrid design for a merge sorter is shown. They explain that for a merge sort, the final stages of the merge sort are the limiting factor for the entire design. This is because the stages before the final stage can work in parallel, but the final stage(s) has to merge its input into the single final output. To this end, they use simple merge units in the earlier stages and a bitonic merge for the final stages. They use 32-bit elements consisting of just keys and achieve a throughput of 9.5 GB/s at 16 elements/cycle with a maximum problem size of 2^{17}.

2.4. Analysis of sorting characteristics

We have described multiple sorting algorithms, but more algorithms exist. Knowing how they approximately work, we can make some analysis about how well they will work on a large scale, how they may perform and how well these may be transferred onto an FPGA.

The classic insertion sort algorithm is one that is more suitable for small amounts of data. It needs to enumerate the sorted list to find the insertion position and once found, move every element one position down before inserting the element. Both of these operations are very costly in a large sort. The binary insertion sort version is an improvement, but not enough.

The counting sort algorithm is an algorithm suited towards sorting data with a low number of unique keys and so not suited to sort a large amount of data when we assume the input is a 64 bit integer.

The merge sort is one that parallelizes very well for the initial stages, but leaves the problem of obtaining the final sorted sequence. The final merger should merge many streams at once at a high throughput, a difficult challenge.

The radix sort algorithm is an algorithm that should parallelize well because multiple subsets of the input can be partitioned in parallel, repeatedly.

The quick sort algorithm repeatedly splits the input into two subsets, which can be sorted in parallel. In principle, this makes it a decent candidate for parallelization. Splitting the input into more than two subsets is even more beneficial, which results in the sample sort.

The sample sort algorithm splits the input into some number of buckets. This makes it a viable candidate for acceleration.

2.5. Algorithm selection

The problem at hand could be viewed as a type of external sorting. Although we can assume that all our data is in main memory of the host, the main memory of the accelerator is HBM.

Even though the available bandwidth to main memory is large, it remains an order of magnitude lower than the bandwidth available to HBM. A key element to increasing the throughput of the sort is to keep the number of passes from and to main memory low.

One approach to external sorting is a merge sort. Part of the input can be sorted locally using HBM. The sorted result can be written back to main memory. Eventually, all the sorted subsets need to be merged by a high performance sorter. The existing work shows that this is possible, but the typical number of sorted subsets that can be merged is relatively low.

Instead, we choose to explore the bucket or sample sort algorithm. Both algorithms split the input into buckets that can be sorted independently. They differ in how the splitters are obtained. Unlike with a merge sort, the sorted buckets do not have to be merged because the buckets are defined by exclusive ranges. Since the buckets can be sorted independently, they can be sorted in parallel and do not have a final merge bottleneck. If the input data can be split into enough buckets such that the buckets fit in HBM, the number of passes for data to travel from and to main memory is down to two.

3

VU37P memory technologies

In Section 3.1 we begin with taking a deeper look into the Xilinx VU37P FPGA and its various memory technologies. In Section 3.2 we perform an early characterization of HBM.

3.1. Xilinx VU37P

The target device, the VU37P, is a member of the Xilinx Virtex Ultrascale+ family, the most high-end FPGAs that Xilinx offers [6]. As shown in Table 3.1 it has a large number of configurable logic blocks (CLB) and contains several on-chip or on-board (BRAM, URAM, HBM) memory technologies. These will be described in Section 3.1.2, Section 3.1.3 and Section 3.1.4.

	VU31P	VU33P	VU35P	VU37P	VU45P	VU47P
CLB Flip-Flops (K)	879	879	1,743	2,607	1,743	2,607
CLB LUTs (K)	440	440	872	1304	872	1304
Max. Distr. RAM (Mb)	12.5	12.5	24.6	36.7	24.6	36.7
Total Block RAM (Mb)	23.6	23.6	47.3	70.9	47.3	70.9
Total UltraRAM (Mb)	90	90	180	270	180	270
HBM DRAM (GB)	4	8	8	8	16	16
DSP Slices	2880	2880	5952	9024	5952	9024
GTY 32.75 Gb/s Transceivers	32	32	64	96	64	96

Table 3.1: Specification summary of Xilinx FPGAs with HBM. [6]

The basic building blocks of the FPGA are the CLBs. In the Ultrascale (+) architecture each CLB contains one slice with eight 6-input LUTs and 16 storage elements. The storage elements can be configured as flip-flops or latches [3, 6]. There are two types of slices: the SLICEL for logic and SLICEM for memory. The LUTs in the SLICEM can be configured as a LUT, RAM, or a shift register. Each LUT has six independent inputs and two independent outputs. The LUTs are also known as function generators because they can implement any 6-input boolean function, or two 5-input boolean functions (sharing their inputs), or two boolean functions of three and two or fewer inputs.

3.1.1. Distributed RAM

Distributed RAM is memory built from the configurable logic of the FPGA itself: the CLBs. Multiple LUTs can be combined in different configurations to store up to 512 bits per SLICEM. This is 512 for a single port RAM and 256 bits for a dual-port RAM. By sharing the clock, write and address inputs between multiple LUTs, a 64x8 (depth x width) single port distributed RAM can be built form a single SLICEM. Another configuration is an octal port 64x1 RAM. Distributed RAM is the only memory that can perform asynchronous reads. By registering the outputs inside the SLICEM, the reads can also be configured to be synchronous. Writes of the distributed memory are synchronous. Typically, small memories are implemented using distributed RAM.

3.1.2. BRAM

FPGAs contain dedicated memory resources called Block RAM (BRAM). The BRAM primitive stores 36 Kib of data in the Xilinx Virtix Ultrascale(+) family (RAMB36E2). This primitive can also be configured as two independent 18 Kb memories (RAMB18E2) [8]. Each BRAM has two independent access ports, typically referred to as port A and port B. Both ports have an input (write) and an output (read) that can be configured with read-before-write (read-first), write-before-read (write-first) or no-change behavior. Read and write ports can also be configured with independent port widths.

An address collision is when both ports of the memory access the same location in the same cycle. With a common clock, if both ports are set to read-before-write and one port has write enabled, the old contents are read. If the port with the write enabled is configured to write-before-read or no-change then the newly written data is read for that port, but the other port has non-deterministic read data. If both ports write different data to the same address the memory is written with non-deterministic data.

When used as a simple dual-port (SDP) memory a BRAM can be configured 512-deep and 72-wide. BRAM can be cascaded with hardware implementations for signal and control logic. Dedicated logic in the BRAM also allows it to be used as FIFOs without using additional logic for the administrative part of a FIFO. The content of the memory can be initialized by the configuration bitstream. The typical latency for BRAM is 1 cycle.

3.1.3. URAM

In addition to BRAM, Xilinx FPGAs in the Ultrascale+ family have another memory resource named UltraRAM (URAM). URAM (partially) fills a gap that exists between the on-board memories and DRAM. Compared to BRAM it is more restricted but it offers more memory. The URAM primitive is a single-clocked synchronous memory with two ports. With 288 Kib, URAM primitives have eight times the capacity of BRAM, configured as a 4096-deep 72-wide blocks. URAM has 2 ports that can both perform either a read or write operation per cycle.

The read-write behavior of URAM is different from BRAM. When using both ports, port A always executes before port B within a cycle. Thus when A reads and B writes to the same address (a collision) we have read-before-write behavior. Conversely, if A writes and B reads to the same address we have read-after-write behavior. If both ports write to the same address, the data on port B is written.

Deeper memories can be generated with 16 URAM blocks per clock region per column, with dedicated resources for cascading. Unlike BRAM, URAM cannot be initialized with user-defined values but initializes to 0 on powerup or reset. Typically URAM has a latency of 1 to 3 cycles, depending on the target frequency.

3.1.4. HBM on the VU37P

The Xilinx VU37P FPGA incorporates two 4Hi HBM2 stacks for a total capacity of 8 GiB and a total advertised bandwidth of 460.8 GB/s. The organization of HBM with the FPGA is shown in Figure 3.1. Each stack has 8 hard memory controllers that each provide 2 pseudo channel interfaces to the FPGA logic [7]. Each interface is an AXI3 port, which is described in Section 2.1.3. The throughput of one port is 1/16th of the stack its throughput, thus providing 14.4 GB/s each. Each AXI3 port is 256 bits wide and has a maximum frequency of 450 MHz[1], together we have $\frac{256 \cdot 450}{1000} = 14.4$ GB/s.

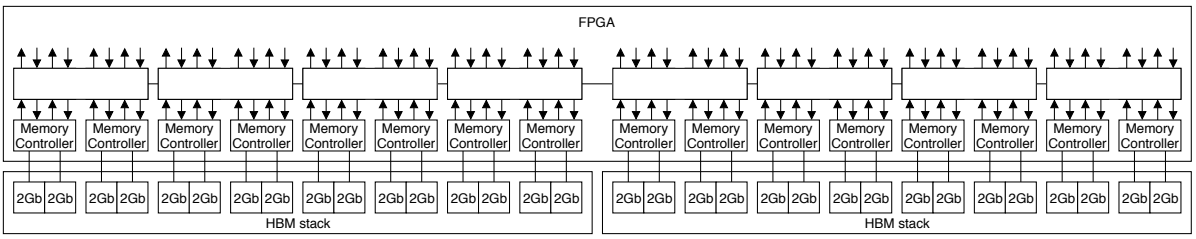


Figure 3.1: Xilinx HBM FPGA organization.

Figure 3.1 also shows the switching network that Xilinx includes in the FPGA. In the traditional HBM architecture, each pseudo channel has access to its own section of memory. Xilinx has implemented a switching network positioned between the AXI3 interface and HBM. The switching network allows any port to access the entire memory, even across the two HBM stacks. This is done by a 32x32 crossbar. This crossbar consumes no LUT resources on the FPGA if enabled.

3.2. HBM characterization

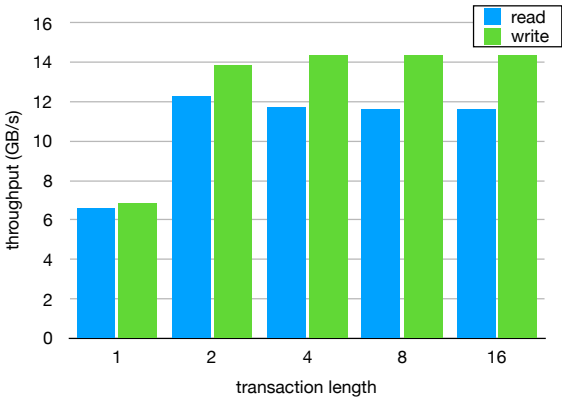
Although the maximum bandwidth of HBM is known, the circumstances in which it can be achieved are not. Several tests were performed to obtain an indication of the performance that can be expected with various read and write patterns to HBM. We explore what happens when the access pattern is sequential vs random. What level of granularity is desired? How does simultaneous read and write affect the throughput? How does mixing sequential and random reads and writes affect the throughput?

The results were obtained from simulation with Questasim 10.6a, Xilinx HBM IP version 1.0 (rev2) and the Vivado 2018.3 provided traffic generator module. The traffic that is generated can be configured in a text file, as documented in [7]. The clock frequency was set to 450 MHz, the maximum frequency for the HBM IP. The results were timed from the first cycle of the first transfer to the final write confirmation or final read return on a single AXI port. Each test transfers 128000 bytes on the read and/or write channel.

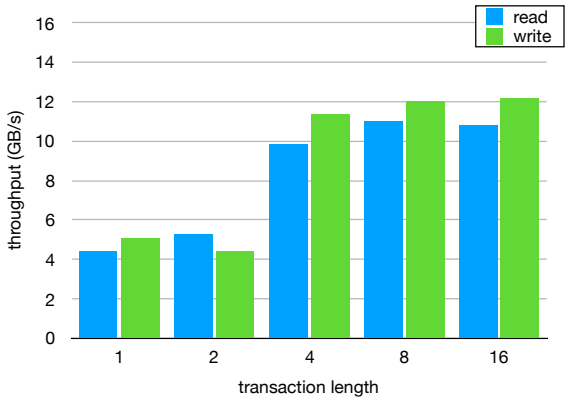
[1]The XVU37P with speed grade -2 supports up to 450 MHz, but the -1 speed grade model supports up to 400 MHz.

A transfer in the AXI3 protocol can consist of multiple beats, where multiple 256 bit transfers are part of the same transaction. The HBM IP supports a transaction length of one up to 16. Figure 3.2a shows the achieved throughput with a sequential access pattern. With transaction lengths larger than 1 the write throughput is close to the port its maximum of 14.4 GB/s. The throughput of read transactions is generally somewhat lower than the write transactions. The lowest result is when the transaction length is 1. Both the read and write throughput are approximately half of the throughput when the transaction length is larger than one.

Figure 3.2b shows the result of varying the transaction length but now with a random access pattern. Compared to a sequential access pattern the read throughput drops by approximately 1 GB/s for transaction lengths of 8 and 16 and slightly more for the length of 4. A bigger gap exists when the transaction length is 2. The performance when the transaction length is 1 is similar to the performance with a length of 2.



(a) Sequential access pattern.



(b) Random access pattern.

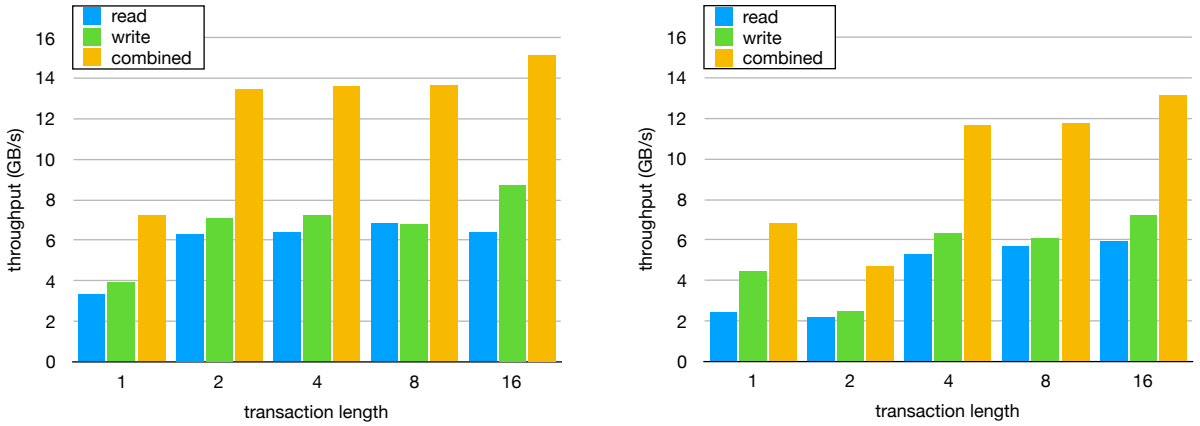
Figure 3.2: Achieved throughput for varying transaction lengths with different access patterns.

An application may read and write to and from HBM simultaneously. In tests with simultaneous read and writes, the read and writes are performed in their own section (half). 128000 bytes are issued on both read and write channels, as soon as possible. Thus, when either the reads or writes finish before the other, the remaining portion is effectively non-simultaneous. The assumption here is that every byte that is written shall also need to be read. An architecture that uses both the read and write port can run at half the frequency (225 MHz) and still saturate the port its 14.4 GB/s bandwidth, since both read and write ports are 256 bit wide. Again we will look at how the transaction length affects the throughput.

Figure 3.3a shows the throughput for reads is greater than 6 GB/s when the transaction length is greater than 1. The throughput for write transactions is slightly higher at around 7 GB/s. The combined throughput is close to the specified bandwidth of 14.4 GB/s at around 13.5 GB/s and even exceeds it for a transaction length of 16. The throughput almost halves with a length of 1, similar to the non-simultaneous test.

Figure 3.3b shows simultaneous read and writes with a random access pattern on both. At 11.6, 11.8 and 13.1 GB/s the throughput with transaction lengths of 4, 8, 16 respectively is high but slightly lower than the maximum. With transaction lengths of 2 and 1 the throughput drops significantly, at around half that.

A realistic access pattern for real applications could be a mix of sequential reads and random writes or random writes and sequential reads. From previous results, we know that



(a) Sequential read and writes.

(b) Random read and writes.

Figure 3.3: Achieved throughput for varying transaction lengths with sequential or random simultaneous read and writes to separate sections in the same pseudo channel.

for a sequential access pattern we ideally have a transaction length greater than 1. For the random access pattern, we will assume the worst-case: a transaction length of 1.

Figure 3.4 shows the achieved throughput for both scenarios. We see that the random access channel is lowest as expected, but the throughput of the sequential channel exceeds the throughput as measured in Figure 3.3a and the combined throughput still reaches 13.2 and 11.2 GB/s respectively.

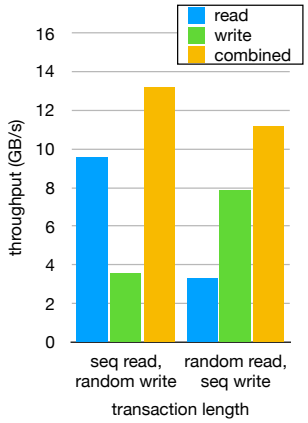
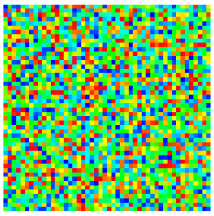


Figure 3.4: Achieved throughput with either sequential or random access pattern on the read or write port, simultaneously in separate sections.

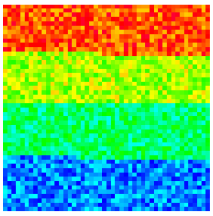
4

Algorithm hardware design and implementation

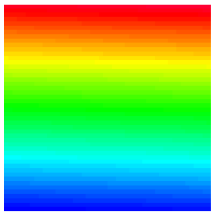
The implemented sorting algorithm is a two-step process, shown in Figure 4.1. In the first step, the input data is partitioned or split into buckets. In the second step, each bucket is sorted. The design minimizes the number of times the input data is transferred from and to main memory down to two, once for partitioning and once for sorting. The advantage of the buckets is that they can be sorted fully independently, thus concurrently, as they hold elements within a certain range without overlap with other buckets. Additionally, sorted buckets can be written to their final position immediately since the size of all other buckets is also known in the second step.



(a) Image representing one random input set.



(b) The input set partitioned into 4 buckets. Because the color represents the sorting key, we can visually make out the buckets and their boundaries.



(c) Each bucket sorted: the input set is sorted.

Figure 4.1: Visual representation of the algorithm algorithm. The color of each pixel represents its value to sort.

In the following, we start by discussing the general architectural concepts we use, in Section 4.1. Two architectures were designed for the first step (the partitioning step) as described in Section 4.2 and Section 4.3. The architecture for the second step (the sorting step) is described in Section 4.4.

4.1. Architectural concepts

Fundamental to the design approach was to design relatively simple engines that output only one element/cycle. To achieve a higher throughput for the overall sort, multiple engines are instantiated and operate in parallel. This is possible for both steps of the algorithm. In the first step, multiple engines can work on partitioning parts of the input data in parallel. In the second step, the very nature of the buckets is that they can be sorted in parallel. Therefore, the descriptions of architectures describe a single partitioning engine or a single sorting engine.

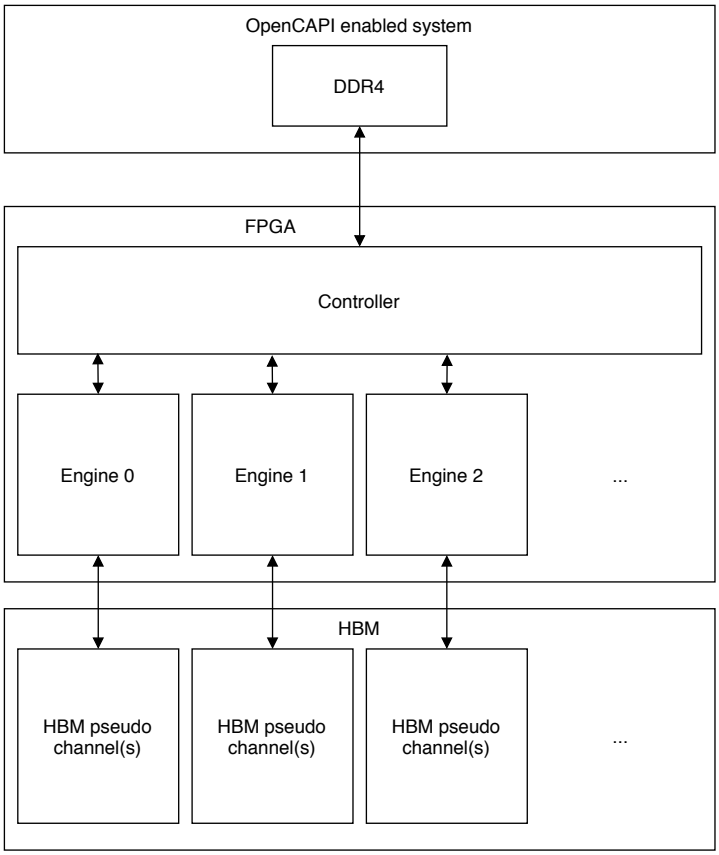


Figure 4.2: Overview of the system. The engines represent instances of the architectures.

An overview of the system is shown in Figure 4.2. The engines are instances of the architectures described in Section 4.2 or Section 4.3 in the partitioning step instances and Section 4.4.1 in the sorting step. Depending on the architecture, one engine may use multiple pseudo channels of HBM. The controller is described in Section 4.1.1. Each architecture is designed to process an input stream at 1 element/cycle. On average their output is 1 element/cycle.

4.1.1. Controller

A controller is required to perform the administrational tasks for the different architectures. The controller, not developed in this thesis, shall be responsible for reading and writing data from and to main memory via OpenCAPI. It needs to feed data into the partitioning engines and write their output to main memory. Then, it must read the buckets from main memory and feed it into the sorting engines.

If after the partitioning step the buckets are not of equal size, it is beneficial for the controller to allocate buckets to sorting engines in order of their size. This is because the sorting engines overlap (pipeline) streaming in a new bucket and streaming out a sorted bucket. If these two buckets are not equally sized, the second bucket may be ready for streaming out before the first bucket was completely flushed, preventing the processing of the third bucket.

To achieve the desired throughput the controller is expected to have buffering for reads and writes from and to main memory, since each engine has an input and average output rate of 1 element/cycle. How the data is stored in main memory is up to the implementation of the controller.

4.1.2. Splitters

The partitioning step splits the input data into buckets. The values by which the input is split are named splitters. Depending on the knowledge of the input data a preprocessing step may be required to obtain splitters. If the distribution of the input is known, it is simple to generate the splitters. In other cases, it is necessary to sample the input data. The samples must then be deduplicated and sorted to be used as splitters. Since the number of splitters is multiple orders of magnitude lower than the input size, this could be done by the host CPU. In the worst-case, the input data may not be suitable for sampling at all. In that case, the inability to find good splitters means this algorithm may not be a suitable choice.

4.1.3. bucketFinder module

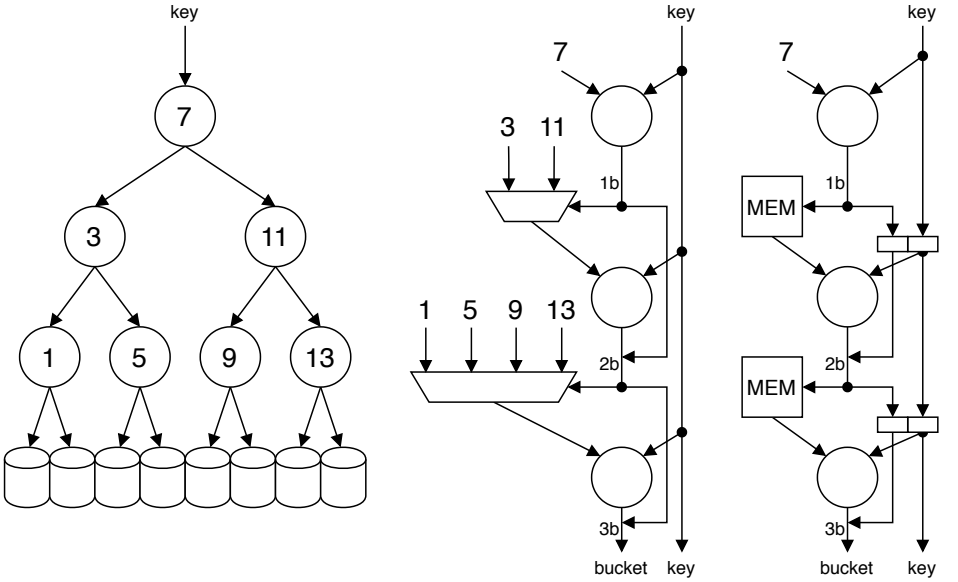
Finding out to which bucket an element belongs is fundamental to the partitioning step. The bucketFinder module is designed to do this and is used in the architectures described in Section 4.2 and Section 4.3. It can be viewed as a pipelined perfect binary tree. Each node is a comparator with one splitter. The tree is organized such that for all nodes the left child is smaller and the right child is larger, as shown in Figure 4.3a. Elements are inserted in the root of the tree and move downwards. Which child they move to depends on the comparison with the value of that node. Their exit position together with the comparison result of the leaf define the bucket.

A tree for b buckets requires $b-1$ nodes or splitters. Assuming that the tree is pipelined such that it can hold one element per level, it can hold $\log_2(b-1)$ elements. If we define the utilization u of the tree as the number of elements in it, divided by its size: $u = \frac{\log_2(b-1)}{b-1}$ we can see that the utilization of the tree quickly drops as b grows. This is a problem if the tree were implemented naively into hardware because most of the hardware is inactive at any time.

Observe that the tree in Figure 4.3a can be collapsed horizontally such that each level consists of a single comparator node. The comparison values must now be dynamic, apart from the root node. The comparison result of the previous node and any of the nodes before it fully define which splitter to select. This way we have a virtual tree, but physically each level has only one comparator, as shown in Figure 4.3b. As the tree in an implementation grows to hundreds or thousands of nodes this approach is likely to become infeasible due to the amount of multiplexing necessary, but this example helps in understanding the next approach.

Instead of having a 2^i-wide multiplexer at level i in the tree, notice that the multiplexer at each level can be replaced with a memory. A memory at level i stores 2^i splitters. The bits

that are the previous comparison results form the read address for the splitter. To support synchronous memories a set of registers is inserted in between every level of the tree. As the virtual tree grows in size to hundreds or thousands of nodes, the number of comparators is $\log_2(b-1)$. Only the required memory depth grows as fast as the number of nodes does. This approach will enable the use of on-device memories such as BRAM.



(a) A perfect binary tree with some comparison values.

(b) Emulation of the tree with a single comparison cell per level, using multiplexers.

(c) Emulation of a pipelined tree using a single comparison cell per level and memories storing the values.

Figure 4.3: Three approaches to implementing a binary tree.

An assumption here is that the memory is able to deliver 1 element/cycle with a delay of 1 cycle. In reality, the desired number of buckets in an application may grow to the extent that such a large memory can no longer be synthesized on an FPGA. In that case, the large memory may need multiple cycles before returning data. URAM, for example, also typically has 1-3 cycles. This problem can be resolved by adding additional registers to the data and comparison outputs, without negatively affecting throughput.

After finding out to which bucket an element belongs, the next step is writing the element back to main memory. However, for performance reasons (OpenCAPI write cache line size, main memory read, etc) it would be naive to write every element back directly. Writes to main memory need to consist of multiple elements for performance. As a result, two architectures were designed that use HBM as a mechanism to 'gather' elements belonging to the same bucket.

4.2. Partitioning architecture 1

4.2.1. Design

Architecture 1 is designed to guarantee the size of transfers to main memory, where all elements of the transfer belong to the same bucket. To do this the available HBM is used as a write buffer. A fixed-size circular write buffer is allocated for each bucket. When a write buffer reaches the target transfer size the data is read from HBM and written to main memory by a flushing module. To avoid stalling the input stage of this architecture, the target transfer size should be smaller than the depth of each write buffer. This depth is restricted by the available memory and the number of buckets. To reach the target throughput for this architecture, it is instantiated multiple times. Therefore, the architect must choose between the number of instances and number of buckets, depending on the available bandwidth to host memory as well as other implications such as the transfer size.

Assuming that each partitioner has access to a pseudo channel of HBM, each partitioner has access to 2 Gib (256 MiB) of storage. With an element size of 16 bytes, the capacity of a single pseudo channel is 2^{24}=16777216 elements. With, for example, 8192 buckets, this results in a per-bucket buffer capacity of 2048 elements (32 KiB). An implementation could flush the buffer to main memory at half that when the buffer holds 1024 elements or 16 KiB of data. This exceeds the minimum advisable transfer sizes greatly, with respect to ideal HBM read sizes as discussed in Section 3.2 and the SNAP cache line size as discussed in Section 2.1.5.

The design needs multiple memories for bookkeeping. It needs to store a per buffer write pointer and the number of elements in each buffer. To prevent overwriting of data or reading data that has not been confirmed to be in the buffer yet, it also needs to keep track of the number of elements in transit to the memory[1]. Since the write buffer is used as a circular buffer, the write pointer only needs to be incremented. The other information needs to be subtracted as well, when data is confirmed to be in HBM or when data has been read from HBM. The design needs a way to decide what buffer to flush, and probably have a small queue for buffers that are ready to be flushed.

4.2.2. Throughput estimation

This design performs reads from HBM in the order of multiple KiB of sequential data, which is then transferred to main memory. Writes are issued at only 16 bytes with a pattern depending on the input. Since we are sorting data, and as a worst-case, we will assume that this access pattern is random. The implementation of the architecture could use one pseudo channel, with the consequence that reads and writes are issued simultaneously. Similar behavior is tested in Figure 3.4 which shows a read throughput of 9.6 GB/s and a write throughput of 3.6 GB/s. What is different about that test and the write access pattern of this design is that in this design reads and writes span a single section, whereas in the test they spanned two exclusive sections. Because the data is written and read once, the effective throughput shall be limited by the slowest of the two. In this case, the write throughput at 3.6 GB/s.

However, the tests in Section 3.2 used the minimum transaction size of 32 bytes. As discussed in Section 2.1.3, the AXI interface has a per-byte write-enable signal. By using this

[1] Meaning that the data has been accepted by HBM, but the write itself has not yet been confirmed.

signal it is possible to issue writes at 16 bytes per transaction. To correct for this difference we assume a 50% penalty in terms of throughput. After this correction, HBM is limited at 1.8 GB/s of 16 byte writes. Assuming the architecture can output 1 element/cycle at, for example, 225 MHz it generates traffic at 3.6 GB/s. If the implementation issues 16 byte writes, the throughput is expected to be limited to 1.8 GB/s.

Alternatively, with two pseudo channels, the read and write operations can alternate such that one pseudo channel performs reads or writes exclusively at a time. Figure 3.3a shows that with a transaction length of 1, the throughput becomes 4.5 and 5.1 GB/s for read and write respectively. With the same 50% penalty assumption, the design is expected to have a throughput of 2.25 GB/s, a 25% increase over the previous case. Although using two pseudo channels increases the effective throughput of the architecture, the HBM bandwidth utilization[2] decreases from $\frac{1.8 \cdot 2}{14.4} = 25\%$ to $\frac{2.25 \cdot 2}{14.4 \cdot 2} = 15.625\%$.

If we do not assume that the input is random but ordered in terms of its buckets, the throughput can increase increase. For example, with a sorted input the write pattern becomes mostly sequential. From the HBM characterization in Section 3.2, we know that this is beneficial. However, when the input is sorted, reads are issued to the same regions as writes which can be disadvantageous.

4.2.3. Implementation

An overview of the implementation of architecture 1 is shown in Figure 4.4. Elements are first put into the bucketFinder module, as described in Section 4.1.3, to find their bucket. When the bucket is known, the write address into the buffer is retrieved by reading the corresponding counter from the first memory (countMem0). The second memory (countMem1) records the number of elements inside the buffer, plus the number of elements that have been accepted by HBM but not yet confirmed to be written. This counter is checked and if the counter plus one exceeds the buffer size, the data pipeline is halted. In that case, the counter is continuously being read from the memory. The flushing module issues a subtraction of the corresponding counter when the buffer has been (partially) flushed. After the subtraction the check passes and the data pipeline continues.

When an element is accepted for writing by HBM, the corresponding bucket is saved in a FIFO. This FIFO is read when the write acknowledge signal is given by HBM and the corresponding counters in the third (and fourth) memory are incremented. These memories hold, for each buffer, the number of elements that have been confirmed, minus the number of elements that the flusher has issued reads for. The fourth memory is a replica of the third to provide an extra read port for the flushing module. The purpose of these memories (countMem2) is to prevent issuing reads before the data is confirmed to be in the memory.

The flushing module is signaled every time FLUSHSIZE elements are being written into HBM. Observe that the write address into each buffer represents the number of elements that have been written to HBM minus one, modulo the buffer size. Thus every time this write address is a multiple of FLUSHSIZE, minus one, the flushing module can be notified that part of a buffer can be flushed. By restricting FLUSHSIZE to be a power of 2, and BUFFERSIZE also to be a power of 2, the lower $\log_2(\text{FLUSHSIZE})$ bits are all 1s when the flushing module must be notified.

[2] Bandwidth utilization in terms of the number of pseudo channels used, at 14.4 GB/s each.

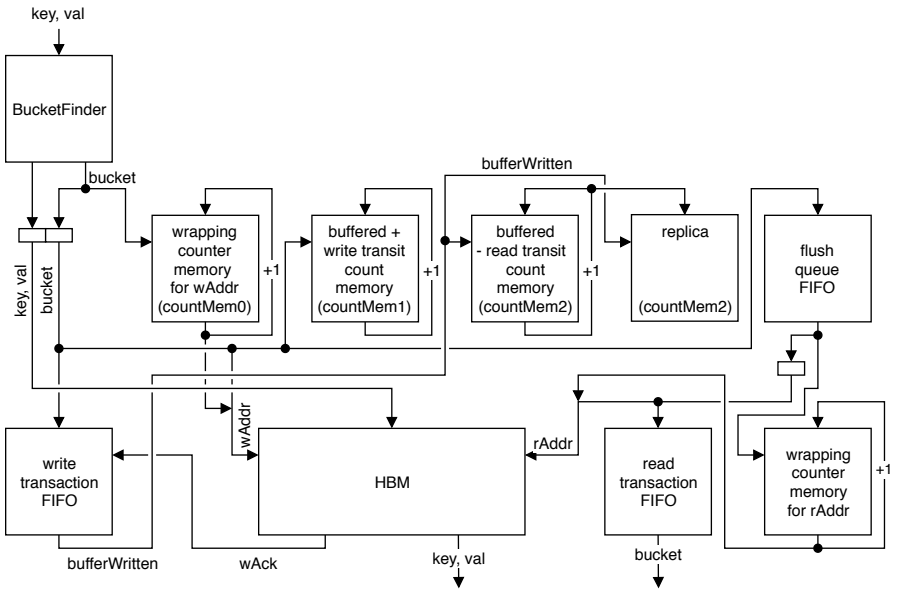


Figure 4.4: Simplified overview of the implementation of architecture 1.

The flushing module itself has a FIFO to queue flushing operations. The flushing module withholds issuing read transactions to HBM until it verifies that all the elements for the flushing operation have been confirmed in memory. This is done by reading from countMem2. The flushing module also has a memory that stores the read address for each buffer. The bits of the bucket and read address are simply concatenated to form the read address into HBM.

The flushing module sends a subtraction signal to countMem2 when reads are being issued. Because of the read latency of HBM, reads for multiple flushes may be issued to HBM. If the next flush is for the same bucket as the previous, this immediate subtraction prevents the flusher from 'thinking' that there are enough elements confirmed in the buffer for the second flush, if there are not. To handle the multiple outstanding flushes, in terms of reads issued to HBM, another FIFO stores the bucket for which the reads were issued.

In terms of memory and FIFO sizes, the required size of the memories are known. Their depth must correspond to the number of buckets and their width to the counters they store. The depth of the FIFOs depends. The first FIFO, holding the bucket number for each write transaction, is of the same depth as the maximum number of outstanding write transactions, which is expected to be low. The third FIFO, holding the bucket number of outstanding read transactions, is as deep as the maximum number of outstanding flushing operations, for which read transactions are issued to HBM. This is also expected to be very low. The second FIFO, holding the flush queue, should be relatively deep since the input data pipeline must be stalled when it becomes full. Specifically, the buffers may fill at an equal rate. In this case, all the buffers may become ready for flushing in a very short time period, in which case the depth should be roughly equal to the number of buckets to prevent stalling.

The latter example also reveals a potential flaw in the current implementation. For example, if the input data is such that it uniformly fills the buffers until they are ready for flushing, and then changes such that it fills a single bucket. The flushing module will first flush all the outstanding flushing operations before getting to the full buffer. Especially if the number of buckets is high, the buffer depth is low, and the flush size is relatively high. An improvement to the architecture

to handle such cases is to have multiple queues with different priorities. Depending on the number of elements in the buffer, the flush signal can be written into a low or high priority queue.

4.3. Partitioning architecture 2

4.3.1. Design

Unlike architecture 1, as described in Section 4.2, architecture 2 is not designed to guarantee the size of transfers of elements belonging to a single bucket. It works by processing batches instead. This simplifies the design of writing data back to main memory. The design of architecture 2 essentially performs a counting sort, described in Section 2.2.1, with respect to the bucket.

The input batch is first fed through a bucketFinder, performing a batch histogram operation. The batch is also temporarily stored in HBM. After one entire batch is processed, the size of each bucket for that batch is known. The elements are then read from HBM and written into a second section in HBM, in bucket order. Assuming that both of these two steps process data at 1 element/cycle, the throughput of this architecture is half that. To prevent this 50% penalty, the execution of both steps can be overlapped for different batches. Since the read and write behavior is fully sequential in the first step, the next batch can use the same memory, as long as the data of the current batch is not overwritten. Due to the random nature of the writes in the second step, it requires two separate sections in HBM. With two sections, one section can be flushed out to main memory while data is written into the other section, simultaneously.

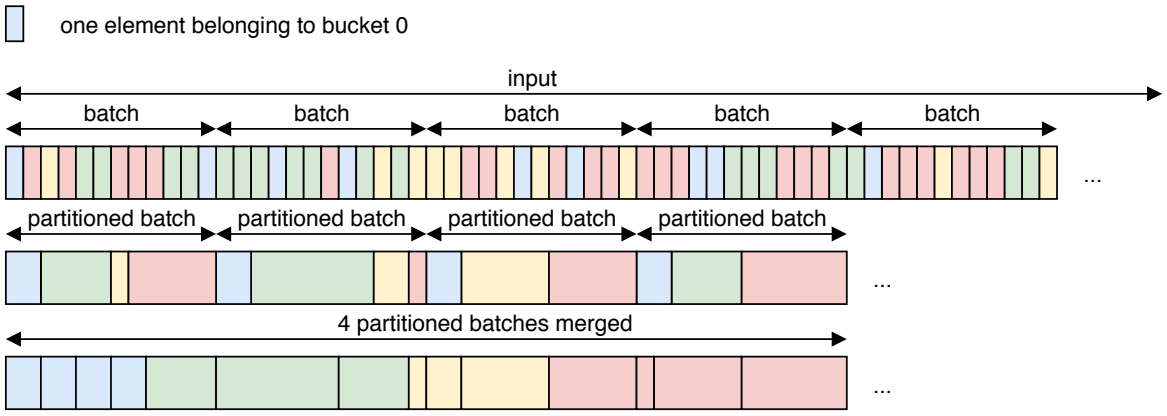


Figure 4.5: The input is split into multiple batches, sized to fit a batch on the FPGA. In the first step the batch is temporarily stored on the FPGA. Then the elements of that batch are moved in the order of their bucket. When multiple batches are processed at once (by using multiple engines), the partitioned batches could be merged in main memory.

On average the number of elements in each bucket for a single batch is $\frac{\text{batchsize}}{\#\text{buckets}}$ but this depends on the input. In the implementation, an option is to couple the flushing process of each engine, such that their partitions are merged when data is flushed to main memory. Figure 4.5 visualizes this. Merging the output has two advantages: when reading data to sort it is less scattered in memory and because the counters for each bucket for a batch are combined, less counter data needs to be stored. A disadvantage can be that the engines no longer operate independently, becoming limited by the slowest engine. However, they should be able to run at the same throughput, depending on the implementation.

4.3.2. Throughput estimation

Architecture 2 writes data to HBM twice. The first time data is written to and read from HBM is sequential. Because this is sequential we can issue write transactions with lengths greater than one, which is beneficial for throughput. The second time data is written is similar to that as described in Section 4.2.2, with the difference that reads and writes are now issued to separate sections in memory. We will assume that the write pattern is random the second time data is written. The reads, which occur simultaneously, are sequential. Therefore the predicted effective throughput is the same as architecture 1, as discussed in Section 4.2.2: approximately 1.8 GB/s, limited by HBM due to the write behavior.

The same option exists to use more than one pseudo channel for the second HBM pass, which again allows a pseudo channel to move from simultaneous read and writes to exclusive read or writes. The gain is also expected to be the same resulting in a throughput of 2.25 GB/s.

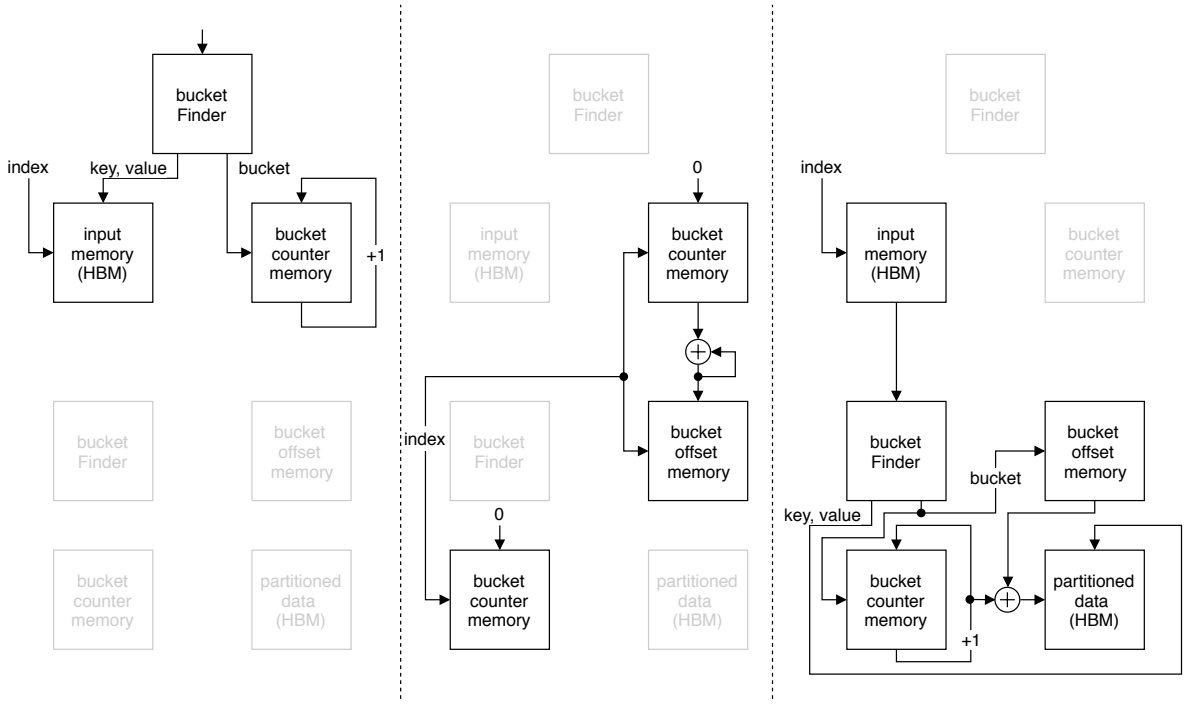
When the input is sorted in terms of its bucket, the write pattern becomes sequential. Unlike architecture 1, in this design reads and writes are issued to separate sections, so the expectation is that this will increase throughput. However, measurements in Section 3.2 are not conclusive about the resulting throughput. For example, if we compare Figure 3.2a with Figure 3.2b (with transaction length = 1) we see that the random writes increase from 5.1 to 6.8 GB/s, a 33% increase. On the other hand, if we look at Figure 3.3a and Figure 3.3b instead, we see that the write throughput dropped by a small amount.

4.3.3. Implementation

Figure 4.6 shows a block diagram of architecture 2. Although the input data is moved twice, the architecture processes the data in three steps. In a fourth step, the data can be read from HBM sequentially and flushed to main memory.

1. In the first step input data is streamed in, to the bucketFinder module as described in Section 4.1.3. A memory stores a counter for each bucket. The corresponding counter is incremented when an element is output by the bucketFinder, while the data is also written into HBM.
2. The actual size of each bucket for the current batch is known after the first step. To be able to move the elements in order of the buckets in step three, the write offset of each bucket is required. A prefix sum over the counters of step one is performed and the results are written into another memory. A zero is written simultaneously to each counter that is read to prepare it for the next batch. The second counter memory, which is active in step three, is also cleared.
3. In the third step, the batch is read from HBM and fed into another bucketFinder to again obtain the bucket of each element. When elements exit the bucketFinder, the offset of the corresponding bucket is read from a memory. The sum of the offset and the second counter form the write position into HBM. Because of the random nature of the input, the entire batch must be written before flushing can start.

The output memory of step three is split into two sections. This allows a flushing module to stream out data from the valid section, while simultaneously writing a new partitioned batch into



(a) Step 1: find buckets and bucket counters for batch.

(b) Step 2: prefix sum bucket counters to obtain write offsets, clear counter memories.

(c) Step 3: move data into bucket order.

Figure 4.6: Block diagram of the implementation of architecture 2, showing the active parts for each step. Grey blocks are inactive in that step. The 'index' denotes an integer that is part of the control logic. It is incremented when reading or writing an element in step 1 and 3, or a counter in step 2, to loop over the memory.

the other section. The active and inactive blocks in Figure 4.6 show that steps one and three can run concurrently on different batches, with one restriction. Step one must not overwrite data that still needs to be read by step three when using one section in memory: the index of step one must be lower than that of step three while step three is active. Because step 2 can not run concurrently with steps one or three, it delays execution of the pipeline by a number of cycles equal to the number of buckets. The significance of this delay is low (negligible) due to the difference in magnitude of the number of buckets and batch size.

By using a second bucketFinder in step three the bucket numbers do not have to be stored for each element. Storing the bucket numbers would require a large memory. For example, with a 2 Gib section of HBM and 8192 buckets, $16777216 \cdot \log_2(8192) = 208$ Mib. Adding a second bucketFinder requires only 512 Kib of storage in this example, with some additional logic. An indirect sort of the elements is an alternative but also requires HBM. For example, $16777216 \cdot \log_2(16777216) = 384$ Mib of storage for the indices of 16777216 elements.

The width of each bucket counter memory is $\log_2(\text{batchsize}) + 1$ and depth is equal to the number of buckets. The bucket offset memory is of the same depth and width, to accommodate for the worst-case where all elements of a batch belong to the same bucket. A scenario with a perhaps counter-intuitively realistic chance to occur. For example, if the input is sorted.

4.4. Sorting architecture

4.4.1. Design

The sorting architecture, responsible for sorting each bucket, is a sort that works by merging repeatedly. The output data of each merge is written into a memory. Each memory is double the size of the previous. When the target output memory becomes too large because the memory consumes too many resources or cannot be synthesized to the target frequency, the output is written to one of multiple 'streams' in HBM. When multiple streams have been written into HBM they are merged into one larger stream by a merge tree.

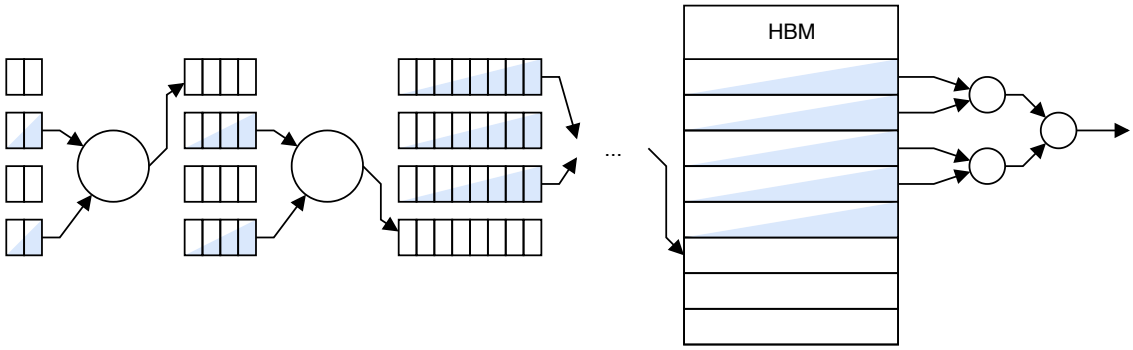


Figure 4.7: Simplified overview of the sorting architecture. Two 2-to-1 merge units and a 4-to-1 merge tree shown.

A simplified overview is shown in Figure 4.7. The merge tree reduces the number of merges using HBM, which are considered to be more expensive at this point, as we can no longer store the intermediate data in BRAM or URAM. For example, with multiple streams of 8192 elements in HBM and a maximum bucket size of 128 MiB $= 2^{23}$ elements, another 10 merges are necessary before the final output length is amplified by $\log_2(\frac{2^{23}}{8192}) = 1024$ and becomes 2^{23} elements long. However, a 1024-input merge tree will also require 1024 read buffers because of the latency of HBM. Furthermore, a 1024-input merge tree is relatively large in terms of resource consumption. Instead, the single merge tree can be replaced by two 32-input merge trees with an additional pass through memory (HBM) since $32^2 = 1024$. The number of merge nodes required for a 1024-input merge tree is 1023, whereas the total number of merge nodes for two 32-input merge nodes is 62. The number of read buffers also decreases from 1024 to 64.

The final memory required is much larger than any before it. It is 32x the size in the example given earlier. Since the memory before the final memory is assumed to also be in HBM, the layout in HBM should be considered. For example, when both stages share one pseudo channel (2 Gib) the first $\frac{1}{32}$th section (64 Mib) can be used for the first pass, and the second $\frac{31}{32}th$ section (1984 Mib) can be used for the second pass. In this case, the second tree can be modified to merge 31 streams of the same size, instead of reducing the size of the 32 streams to 62 Mib. This way, the size of each stream remains a power of 2, which is advantageous in the implementation.

To hide the read latency of HBM, read buffers acting as FIFOs should be placed between it and the merge tree. HBM read transactions are issued when storage in one of the FIFOs becomes available. Reads are issued with a transaction length of 16 (32 elements) until the stream nears depletion. When a stream in HBM nears depletion and the read buffer has enough space, the remaining elements are read. A small FIFO stores information about this

read transaction: which stream the read belongs to and how many elements are read. This information is used when the read data arrives, to write the data into the correct FIFO. Because read transactions are issued only when FIFO space is available, the read data can be always be written to the FIFO. Since the goal is to hide the read latency, the FIFOs need to be sized accordingly. The latency of HBM may vary under different circumstances, which also depends on the input. A worst-case scenario for the buffers is, for example, a sorted input. With a sorted input the tree will consume elements from only one read buffer at a time, leaving all other FIFOs unused. With a random input, the reads from the FIFOs are more evenly distributed, such that the FIFOs can be smaller.

4.4.2. Throughput estimation

By nature of the merge sort, the writes into HBM are sequential. What stream the input is read from is not, but an advantageous property of the additions of FIFOs between HBM and the merge tree, is that reads can also be aggregated by multiple elements. We can assume that reads and writes are both issued at the maximum transaction length of 32 elements, simultaneously. From the HBM characterization in Section 3.2, we know that this transaction length results in a throughput of around 6 GB/s, limited by the read channel.

Let us assume that an implementation uses two HBM pseudo channels. One as input to the first merge tree and another for the second merge tree. Assuming a throughput of 1 element/cycle from the mergers themselves, the maximum throughput of an implementation running at 225 MHz is 3.6 GB/s. This is exactly half of the theoretical maximum of HBM at 7.2 GB/s[3]. Thus, the implementation is limited by itself rather than HBM. Assuming the 6 GB/s figure, the 'simple' improvement of raising the frequency is expected to cause a linear increase in throughput up to approximately 375 MHz.

Assuming that the implementation uses a single pseudo channel for two merge trees then by design, at for example 225 MHz, the architecture requires 7.2 GB/s of bandwidth. While this is theoretically available, assuming the 6 GB/s figure instead, the architecture is limited to an effective throughput of 3 GB/s. In the latter case the bandwidth utilization of the pseudo channel doubles. To utilize the entire available HBM it may be more practical to use multiple pseudo channels per sorting engine, at which point the HBM bandwidth utilization is decreased.

4.4.3. Implementation

The implementation of the sorter uses two memories as the input to every 2-to-1 merge stage. The two memories hold a total of four streams. With storage for four streams, the previous stage can write two more streams to memory while the current stage merges the two other streams. In this way, multiple 2-to-1 stages can be chained together to run at 1 element/cycle. Each stage has ready/valid signaling with both the previous and next stage to assert that the streams in each memory are valid or ready to be written to. The final 2-to-1 merger writes its output to multiple streams in HBM. These write transactions are issued with a transaction length of 16 (32 elements) since the HBM characterization in Section 3.2 shows that is beneficial. When the data is confirmed to be written to HBM, the final merger asserts its data valid signal.

[3] 7.2 GB/s is the theoretical maximum effective throughput. Half of 14.4 GB/s because data is both read and written.

The HBM pseudo channel is organized in a similar way. With a k-wide merge tree behind the final 2-to-1 merger, the available HBM space is split into two sections storing k streams each. HBM returns read data with 256 bits per cycle, but the tree is designed to output 1 element of 128 bits per cycle. Therefore, the memory for the FIFOs is implemented as an asymmetric memory. An asymmetric memory is a memory whose ports have different widths. The asymmetric memory allows 2 elements to be written in each cycle and one element to be read each cycle. When updating the FIFO counters, some care is taken to correctly update the counter. The FIFO count is decremented by 1 when the tree consumes an element from it and no new data is written to it. When data is written to it, the corresponding FIFO count may need to be incremented by 2, 1 or kept the same, depending on the read and whether an element was simultaneously consumed from the FIFO.

The depth of each FIFO is not fixed in the implementation, but is configurable to be a power of 2, larger than or equal to 32. Ideally, the memory is very deep such that it can hide the read latency in all situations, but increasing the depth comes at the cost of resource usage and could impose a lower design frequency limit.

The strategy for issuing HBM reads, writing the read data into the FIFOs is as follows. For each input stream we check:

- Are there more than 31 elements left in HBM? If the number of elements in the FIFO, minus any number of elements underway (from pending read transactions) then this stream can be read with 32 elements.
- Are there are less than 32 elements left in HBM? If the number of elements in the FIFO, minus the number of elements underway (from pending read transactions) then this stream can be read for the remaining number of elements.

A priority encoder selects the first stream that can be read and the read transaction is subsequently issued to HBM. This strategy also works for the initial period, where the FIFOs are initially empty. A consequence of the simple priority encoder is that every other FIFO is filled before the last. The first output of the tree is delayed because it cannot output data until the first element of each input stream is known. However, it is expected that this initialization phase is relatively short compared to the overall merging process, such that it is not a significant factor of the sort.

A naive implementation of a merge tree, without registers, will not synthesize to a desirable frequency. Therefore, the implementation of the tree has all output signals registered. To do this and support a throughput of 1 element/cycle, each node also has an internal buffer of 1. Each node works according to the following logic:

- If the output of a node is consumed and the internal buffer holds a valid element, the element in the buffer is moved to the output register. Inputs are consumed when the internal buffer is not valid.
- If the output of a node is consumed, and the internal buffer holds no valid element, the value that the node itself consumes is forwarded to the output register.
- If the output of the node is not consumed, and the internal buffer holds no valid element, then the consumed value is stored in the buffer.

By doing so all the output signals of each node can be registered. A small number of cycles are required before the tree is initialized and the output becomes valid. Because the size of the output stream is much greater, this is negligible.

The number of 2-to-1 merge stages can be set through a single parameter. The implementation is written with two merge trees, but the modification towards more merge trees is straightforward. Both merge trees use a HBM pseudo channel as their input, for a total of 2 pseudo channels. The width of both merge trees is individually configurable to a power of 2.

5

Results

In this chapter, the results of the different architectures are presented. First in Section 5.1 we explain how correctness of each architecture is evaluated. In Section 5.2 the performance is evaluated. Finally in Section 5.3, the resource consumption of each architecture is evaluated.

The performance and area results describe the results of a single instance of each architecture. The number of instances is limited by the resources available on the FPGA and HBM. This also depends on parameters that exist in the designs. For example, the number of buckets, the maximum bucket size, the size of flushing operations in architecture 1, the batch size in architecture 2, etc.

Questasim 10.6a was used for behavioral simulation. The verification and performance results are obtained from this simulation. Vivado 2018.3 was used for synthesis. Simulation and synthesis both use the HBM IP version 1.0 (rev 2), as provided by Vivado 2018.3.

5.1. Verification

For the functional verification of the designs, the partitioned and sorted data as output by the simulations are written to disk. For architecture 1, the output of each flush operation is written to a separate file. For architecture 2, the output for each batch is written to a separate file. For the sorting architecture, each output bucket is written to a separate file. Specific Python code was written for each architecture to process data in a similar way, to output data in the same way. The Python code has some of the same parameters as the implementations described in Chapter 4 that affect the output. It reads from the same input files as the simulation does. The output is written in the same format to a file and checked to form a known good output. Correctness is finally evaluated by comparing the output files from simulation and Python.

5.2. Performance

Because of the performance of the simulation it was necessary to test with significantly lower quantities than the actual implementation will use. How this has affected the measurement method and result is described separately.

The measurements of both partitioning architectures were done with a (uniformly) random input and a sorted input, both 1 MiB in size. A set of 256 splitters was chosen such that it (approximately) evenly splits the inputs.

5.2.1. Partitioning architecture 1

The per-bucket buffer depth was set to 1024 (16 KiB). The flushsize was set to 128 (2 KiB). These parameters result in each buffer being flushed twice with the sorted input. The measurement period is from the first element being accepted by the partitioner, until the final element is accepted.

The measurement method was different for the random input because it does not guarantee 2 flushes per bucket. Additionally, it does not perform flushes for approximately the first $256 \cdot 128$ elements. To measure it, a counter c stores the number of elements accepted by the partitioner. A second counter stores the number of flushes that have been performed. When the number of flushes is 256, we have c_i elements accepted by the partitioner. The period of $c_{i-256 \cdot 128}$ to c_i then approximately corresponds to a period in which $256 \cdot 128$ elements have been written to, and read from HBM.

The results should be representative for a large scale case because they do not measure overhead, such as initialization time.

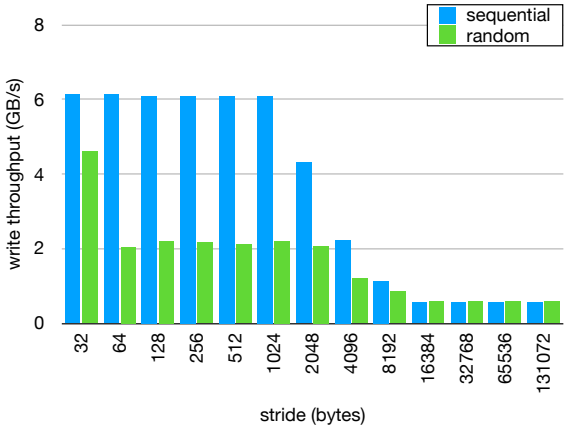
Frequency (MHz)	sorted throughput (GB/s)	random throughput (GB/s)	architectural bound (GB/s)
225	2.18	0.44	3.6
450	2.21	0.55	7.2

Table 5.1: Achieved throughput of architecture 1.

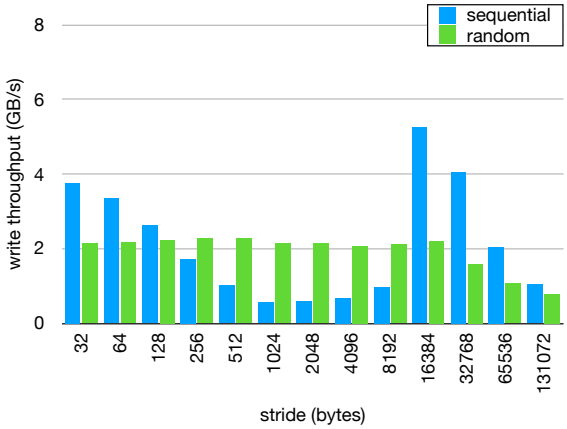
The results are shown in Table 5.1. At 225 MHz, the throughput of the sequential input is 2.18 GB/s. This is slightly better than expected, as was described in Section 4.2.2. However, the throughput of the random input is much lower than expected at 0.44 GB/s. The access behavior of this test is related to the HBM DRAM architecture itself, a component that was not explored in the HBM characterization in Section 3.2. The input set here is such that elements are written to one of the bucket buffers, which were sequentially mapped to HBM. Because of the way the buffers are mapped to the physical memory, its properties come through and form a bottleneck.

Figure 5.1a shows the write throughput for transactions with a given stride, in sequential or random order. The sequential order is not of particular interest but included for reference. In the sequential case, the throughput is approximately 6 GB/s with a stride of 32 through 1024 bytes. The next four steps increasing the stride to 2048, 4096, 8192 and 16384 bytes show a significant decrease in throughput which settles at approximately 0.57 GB/s. In the random case, the throughput with a stride of 32 bytes is 4.60 GB/s, dropping to 2 GB/s for the

next 6 larger strides. Similar to the sequential case, the throughput decreases when the stride becomes 4096, 8192 and 16384 bytes, the exception being that there appears no drop in the 2048 stride.



(a) Using the default address mapping: (ROW, BANK, COLUMN).



(b) Using a custom address mapping: (ROW[13:4], BANK, ROW[3:0], COLUMN).

Figure 5.1: Write throughput when writing with a given stride, in sequential and random order, using different address mappings. Transaction length of 1 (32 bytes) at 450 MHz.

The reason for the drops in performance in the sequential test, when the stride is 2048, 4096, 8192 and 16384 bytes, is the way the addresses are mapped to the physical memory. The default configuration of the HBM IP uses a ROW BANK COLUMN address map. In this configuration bits 10, 11, 12 and 13, corresponding to values of 2048, 4096, 8192 and 16384 bytes respectively, indicate which bank the address maps to. Because there exists parallelism when multiple columns are accessed on the same bank, and multiple banks are accessed, this increases performance. When the test was performed with a write buffer depth of 1024 elements (16384 bytes), most writes occurred in the same bank in different rows, one of the least optimal uses of DRAM. For the random case, a reason for why the throughput with a stride of 32 bytes is higher could be bank group interleaving. Bank group interleaving is a feature that is also enabled in the default HBM IP configuration that ”Enables sequential address operations to alternate between even and odd bank groups to maximize memory efficiency” [7].

Figure 5.1b shows the write throughput with a custom mapping. Here, the address mapping is organized as: ROW[13:4], BANK, ROW[3:0], COLUMN. By moving some of the ROW bits right, past the BANK bits, varying the write buffer now varies the bank. The throughput suffers when the access pattern is sequential but increases greatly when the stride is 16384 bytes. The write throughput with a random order no longer decreases after 2048 bytes, but now shows a similar decrease after 16384 bytes. Which is as expected, as such strides decrease the variance in addressing multiple banks. When using a custom address map, the bank group interleave feature can not be enabled, and the peak that existed in random addressing with a stride of 32 vanishes.

As a result of the change in address mapping, the throughput of this architecture also changes. The throughput at the maximum frequency of 450 MHz increases. With the random input, it has increased to 0.88 GB/s. However, the throughput of a sorted input is affected negatively: it decreases to 1.21 GB/s.

5.2.2. Partitioning architecture 2

Partitioning architecture 2 works in batches. At most, three batches are being processed at one time (one streaming in, one being re-ordered, one streaming out). Initially there is one, then two, then three batches being processed when the partitioning phase starts. The individual throughput of these first batches, as well as the last, will measure to be higher. Because this will positively affect the results in this small scale, these three have been excluded from the result. The 1 MiB input is divided into 8 batches of 128 KiB. Because architecture 2 stops processing data for a number of cycles every batch equal to the number of buckets, this is approximately 3% of the processing time of a batch[1]. This will become much less significant when the batch size becomes equal to half the memory of the HBM pseudo channel, even if the number of buckets grows.

Frequency (MHz)	sorted throughput (GB/s)	random throughput (GB/s)	architectural bound (GB/s)
225	1.75	1.66	3.6
450	1.81	1.69	7.2

Table 5.2: Achieved throughput of architecture 2.

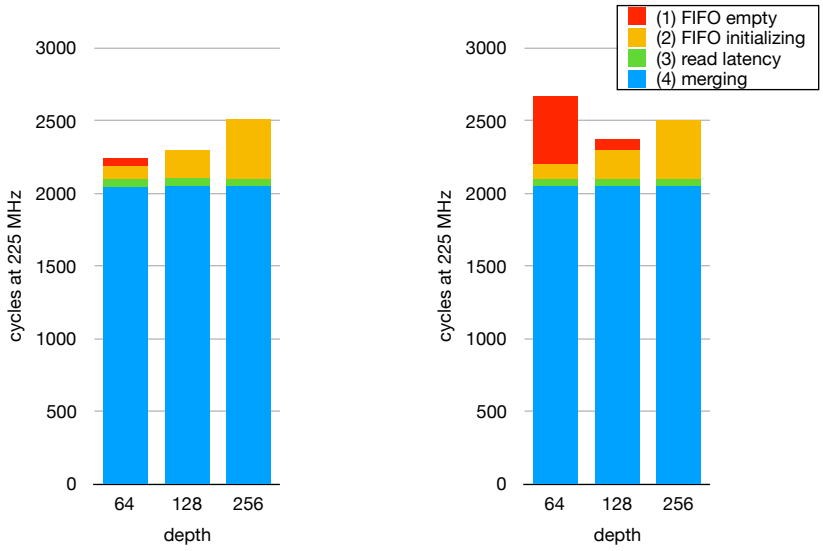
The throughput of this architecture is shown in Table 5.2. At a frequency of 225 MHz, we have 1.75 and 1.66 GB/s for a sorted and random input respectively. This is very close to the prediction in Section 4.3.2. The throughput of the random input is slightly lower because it causes a random write access pattern, which is known to reduce the throughput slightly. We note that the write pattern of the sorted input is not truly sequential in terms of its addressing because two write transactions are issued for every address, since each address holds 2 elements[2]. The throughput is not bound by the architecture as the architectural bound of 1 element/cycle is 3.6 GB/s. Instead, it is bound by the way it interfaces with HBM. This can additionally be confirmed by observing the (lack of) increase in throughput when the frequency is raised to the maximum of 450 MHz.

[1] This refers to step 2 in which a prefix sum is computed. This is described in Section 4.3.3.

[2] The HBM IP ignores the lowest 5 bits of the address, corresponding to the interface width of the AXI3 ports of 32 bytes.

5.2.3. Sorting architecture

For the sorting architecture, the same 1 MiB input is divided into 8 parts (buckets) of 128 KiB. This is done to simulate multiple buckets being sorted consecutively. The random and sorted inputs represent the best and worst case for the architecture, as described in Section 4.4.1. Therefore, the throughput results present an approximate lower and upper bound. Although the input at this small scale can be sorted with a single merge tree, the throughput is evaluated by using two 4-to-1 merge trees. This better reflects the implementation at a large scale, where two merge trees will also be necessary to sort the entire bucket.



(a) Uniformly distributed input.

(b) Sorted input.

Figure 5.2: Cycle decomposition of the first merge tree when varying the input and FIFO depth.

One of the parameters that is configurable in the design is the depth of the FIFOs that are positioned in between HBM and the merge tree, as described in Section 4.4.1. Figure 5.2 shows the cycle decomposition of the first merge tree when the designs run at 225 MHz. The decomposition lists the following four cases, in order of their appearance:

1. Cycles spent waiting on data because a FIFO is empty while the input stream is not depleted. This does not include the cycle count of initialization.
2. Cycles spent initializing (filling up) the FIFOs. Counting stops when there is no FIFO empty, so when the last FIFO receives its first element.
3. Cycles spent waiting for the first read to return data. Counting starts when the merge tree module was notified that its input (in HBM) is ready.
4. Cycles spent merging. Since the tree itself outputs 1 element/cycle, this will be equal to the number of elements being sorted.

Because of the implementation of how reads are issued to HBM, the length of the initial phase where the FIFOs are being filled depends on the depth (and amount) of the FIFOs. Figure 5.2 shows that this phase doubles when the FIFO depth doubles, as expected. A lower depth of 32 was excluded because the implementation does not issue reads until 32 spaces are available, causing many FIFO empty cycles.

The cycle decomposition with a (uniformly) random input is shown in Figure 5.2a. With a depth of 64, we see that a small amount of time (about 2.4%) is spent waiting on data after initialization. The uniformly distributed input is one where elements are consumed from the different FIFOs at roughly an equal rate. Conversely, a sorted input is one that causes maximum utilization of a single FIFO at a time. In Figure 5.2 we see that the time spent waiting on data is more significant: 17.7% when the input is sorted, using the same depth of 64. Doubling the depth to 128 reduces this to 3.2%, and a further doubling eliminates this. Since the architecture of the merge tree is designed to output 1 element/cycle, the merging time (4) is the same.

The decomposition also indicates how the small scale of the test affects the throughput. The total initialization time, the sum of (2) and (3), in the uniform case is 6.5, 10.6, 18% and in the sorted case 5.4, 10.2, 18% for depths 64, 128 and 256 respectively. This is a significant amount of time, which will affect the sorting throughput result.

If we assume that an implementation implements a 32-to-1 tree with 16x larger inputs, corresponding to 8192 elements for each input stream to the first merge tree, then the significance of the initialization time is much smaller. The initialization time of (3) grows by $(32-1)/(4-1)$ about 10x, but the active merging time grows by $(32/4 \cdot 16) = 128x$. Assuming a 10% initialization time, it will become approximately $10 \cdot 10/128 = 0.8\%$. To project the expected throughput on a larger scale, the measured throughput of this test should therefore be increased by 10%.

The buckets in this simulation were equally sized however, in practice they may not be equal, depending on the input. In that case the buckets should be processed in order of their actual size, to maximally overlap the processing of buckets.

Frequency (MHz)	FIFO depth (elements)	sorted throughput (GB/s)	random throughput (GB/s)
225	64	2.34	2.76
	128	2.68	2.76
	256	2.56	2.56
450	128	3.84	4.31
	256	3.84	4.57

Table 5.3: Achieved throughput of the sorting architecture.

Table 5.3 shows the achieved performance when measuring the dataset from the acceptance of the first element to the output of the final element by the second merge tree. With a frequency of 225 MHz, the throughput of the sorted input is approximately 15% lower than the random input when the depth is 64, 3% lower when the depth is 128 and the same when the depth is 256. This is expected because with a depth of 256 the FIFOs do not become empty while sorting. The sorting time of the merging process itself is constant, regardless of the input. When the frequency increases to the maximum of 450 MHz, there is a gap again between the sorted and random inputs. At this frequency, the FIFO depth of 256 is not deep enough to prevent one from becoming empty, reducing the throughput.

Additionally, the small number of buckets in this test also affects the results. The architecture is designed to overlap streaming out a sorted bucket, together with streaming in a new bucket. It can be seen as a pipeline of two stages which take an equal amount of time. If we say that both steps take 1 unit of time, then for n sorts in a non-pipelined version takes $2n$ units of time. The pipeline reduces that to $n-1$. In this case, we have 8 sorts, taking 9 units

of time. However, if the number of sorts doubles then 16 sorts take 17 units of time. When n is larger, we have that the units of time approaches n. Thus, we could say that there is an overhead of $9/8 - 1 = 12.5\%$ that reduces as n increases.

5.3. Resource consumption

Each result reports the resources of a single engine, with one out of two HBM stacks enabled. For a more realistic example of the resource consumption in a large scale implementation, we use a relatively high number of buckets of $b = 8192$ in the two partitioning architectures. Because of the implementations, we keep the assumption that each engine works with a single or two HBM pseudo channels as was defined in Section 4.2.3 and Section 4.3.3. Aside from the HBM IP, no code was written with specific vendor IP in mind. Therefore, all the memories used in the designs are inferred by synthesis. For this reason, any FIFO may also use extra resources, instead of the FIFO logic available inside each BRAM.

We note that in a real implementation, the engines will need to be surrounded by logic and memory to control them and to stream data to and from the main memory using OpenCAPI and SNAP.

5.3.1. Partitioning architecture 1

As described in Section 4.2 the implementation of this architecture uses a single HBM pseudo channel, 256 MiB. For the evaluation of the resources, we use $b = 8192$ buckets, resulting in a per-bucket buffer of 32 KiB. The write transaction FIFO has a depth of 32, the flush queue FIFO is relatively deep at 8192. A small flushing module is included in the results. The flushing module issues read transactions to HBM, which are streamed out. It has a small read transaction FIFO with a depth of 16. The flushing module was configured to flush at 4 KiB.

resource	used	available	utilization (%)
CLB LUTs	13 255	1 303 680	1.02
CLB Registers	2 826	2 607 360	0.11
CARRY8	75	162 960	0.05
F7 Muxes	5 343	651 840	0.82
F8 Muxes	2 574	325 920	0.79
BRAMs	14	2 016	0.69
URAMs	2	960	0.21

Table 5.4: Partitioning architecture 1 FPGA resource utilization from synthesis.

The results of synthesis are shown in Table 5.4. Most of the CLB LUTs are used for the different memories, which the synthesis process has decided to implement as distributed RAM. The F7 and F8 muxes are also used in these memories. 8 BRAMs are used for the bucketFinder, and an additional 4 are used by the flushing module. 2 URAMs are inferred for the last stage(s) of the bucketFinder. The HBM IP itself also consumes some resources: 802 CLB LUTs, 819 CLB registers, 7 CARRY8 and 2 BRAMs which have been included in the result.

If we look at how the resource consumption changes when some of the parameters are modified, we estimate that the main difference is in the different memories implemented in the architecture. If the number of buckets doubles, the depth of most memories also doubles.

This includes the bucketFinder, the four different memories used by the partitioner and the small memory that is used by the flushing module. If the total amount of memory doubles, the four memories of the partitioner become one bit wider. This is a much smaller increase in resources.

5.3.2. Partitioning architecture 2

As described in Section 4.3.3, this architecture uses two HBM pseudo channels. The memory of the second HBM pseudo channel is divided into two sections, restricting the batch size to a maximum of 128 MiB (2^{23} elements). The average size of each bucket with $b = 8192$ buckets is then 16 KiB. In addition to the partitioning architecture itself, a small flushing module is included. The task of the flushing module is simple, it issues read transactions to HBM when the partitioning architecture signals that the data in one of the two output sections is ready. Unlike partitioning architecture 1 it requires no memories to do the administration for every bucket buffer in HBM.

resource	used	available	utilization (%)
CLB LUTs	16 082	1 303 680	1.23
CLB registers	3 982	2 607 360	0.15
CARRY8	181	162 960	0.11
F7 muxes	6 415	651 840	0.98
F8 muxes	3 054	325 920	0.94
BRAMs	23.5	2 016	1.17
URAMs	4	960	0.42

Table 5.5: Partitioning architecture 2 FPGA resource utilization from synthesis.

Table 5.5 shows the results of synthesis. A large amount of CLB LUTs is used as distributed RAM (13 635 out of 16 082). This is for two of the memories the architecture implements: the two bucket counter memories. The equally sized offset memory is synthesized as 5.5 BRAMs[3]. Presumably, this is different because the bucket counters require write-first behavior, whereas the offset memory does not have this requirement. The counter memories can also be synthesized to use 5.5 each BRAMs by adding a synthesis attribute. The first few levels of the bucketFinder module are implemented as distributed RAM. Each bucketFinder also uses 8 BRAMs for the next stages, finally inferring 2 URAMs for the last two stages, which implement memories of 256 Kib and 512 Kib. The HBM IP uses approximately 800 CLB LUTs, 800 CLB registers and 2 BRAMs.

If we look at the increase or decrease in resources when changing the parameters, this will mostly affect the memories that are used in the architecture. A doubling in the number of buckets b generally leads to a doubling of the memory footprint of the bucketFinder[4]. Because the design uses two bucketFinders, the memory usage increases by 4.

3 memories of equal depth and width store the bucket counters and the bucket offset. Their storage requirement is $b \cdot \log_2(maxbatchsize + 1)$. Assuming $maxbatchsize = 2^{23}$ and $b = 8192$ we have $b \cdot 24 = 192$ Kib. Doubling the number of buckets will lead to an increase in total memory by 6. If the batch size doubles, each memory becomes 1 bit wider, a much less significant change.

[3]Half a BRAM is obtained using the RAMB18 primitive, instead of the RAMB36 primitive.
[4]When b is small the number of BRAMs required initially does not double.

5.3.3. Sorting architecture

Parameter evaluation

The memory used by this architecture is determined by the 2-to-1 merge stages, and the (two) merge trees. The output of the final 2-to-1 merge stage is written to HBM. A single 2-to-1 merge stage requires an input memory for 4 elements. With two stages, we require memory for $4+8$ elements. The cost of s merge stages can be written as in Equation (5.1).

$$2^{s+2} - 4 \tag{5.1}$$

The memory required for one merge tree is the depth d of the FIFOs times the width of the tree k, as written in Equation (5.2).

$$dk \tag{5.2}$$

To sort an input of at least N elements, with s 2-to-1 mergers and t k-wide trees we have Equation (5.3).

$$N \geq 2^s k^t \tag{5.3}$$

The current implementation described in Section 4.4.1 restricts the input size to the size of a single HBM pseudo channel, $N = 128$ MiB $/16$ B$= 8388608 = 2^{23}$ elements. It uses two merge trees. Additionally, it restricts the width of each merge tree k to be a power of 2. From Section 5.2.3 we know that a FIFO depth of 128 or 256 is preferred. Using Equation (5.1) and Equation (5.2) with $d = 128$ we obtain the memory size required for the FIFOs of both trees and the 2-to-1 merge stages, in terms of the number of elements:

$$2^{s+2} - 4 + 2(128k)$$

By plugging the values into Equation (5.3) we have the following restriction:

$$2^{23} \geq 2^s k^2$$

By rewriting we obtain:

$$k \geq 2^{11.5 - s/2}$$

Which we plug into the equation that we had, obtaining:

$$2^{s+2} + 2^{19.5 - s/2} - 4$$

Figure 5.3 shows the plot of the equation for FIFO depths of 128, 256 and 512. Analytically we obtain minima of $s = \frac{33}{3} = 11$, $\frac{35}{3} \approx 11.67$ and $\frac{37}{3} \approx 12.33$ respectively. With $s = 11$ and a FIFO depth of 128 we obtain $k = 64$. Here, k is a power 2 of two when s is odd. Otherwise we can use trees of different widths, by rounding k down and up to the nearest power of 2. Rounding k in this manner increases the storage and number of merge nodes in the trees by 6% in the worst case with $N = 8388608$, compared to rounding k up. In general with $N = 8388608$, k is a power of 2 when $s + 1$ is a multiple of t.

To further reduce the resources of the sorting architecture, the number of merge trees can be increased. Figure 5.4 shows the memory required when varying the number of merge

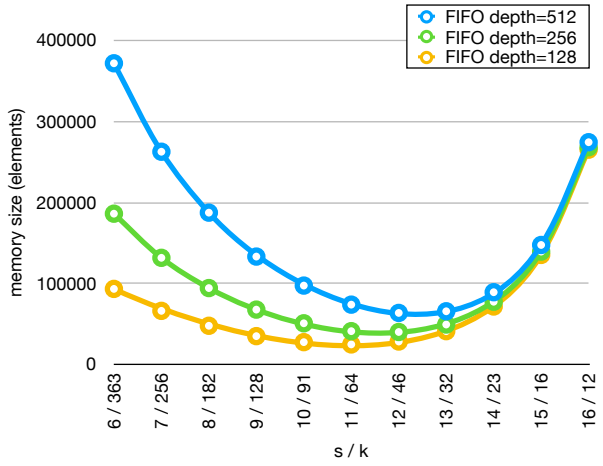


Figure 5.3: Memory size of s 2-to-1 merge stages and the FIFOs of two k-wide merge trees for $N = 8388608$ elements. The resulting value for k is round up when it is not an integer (when s is even).

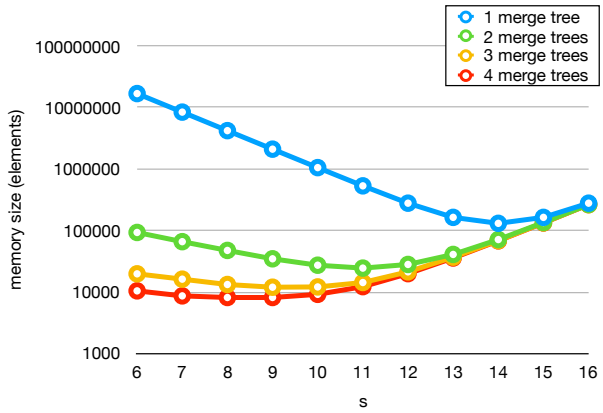


Figure 5.4: Logarithmic plot of the memory size of s 2-to-1 merge stages and the FIFOs for a varying number of merge trees to sort $N = 8366806$ elements.

trees. With a constant bucket size N, we see that the memory required reduces significantly when the number of merge trees increases. Additionally, narrower merge trees require fewer resources, but the HBM usage increases. Both in terms of HBM used, as well as an increase in bandwidth utilization.

Synthesis

From the parameter evaluation we chose the configuration of $s = 11$ 2-to-1 mergers and two $k = 64$ wide trees. This restricts the bucket size to 128 MiB. Of the first HBM pseudo channel $\frac{1}{64}$ of the memory is used. The memory of the second pseudo channel is fully utilized. Both trees use a small FIFO with a depth of 64, to store information about the HBM read transaction.

The results of synthesis are shown in Table 5.6. Synthesis infers BRAMs for most of the 2-to-1 merge stages, until inferring URAMs for the last two stages. Synthesis infers 64 BRAMs as memory and approximately 18 000 CLB LUTs for the FIFOs of the two merge trees. The merge trees use approximately 13 000 CLB LUTs and 16 000 CLB registers each.

A more balanced resource consumption could be obtained by using URAMs instead of BRAMs for the FIFOs. Replacing the 64 BRAMs with 8 URAMs will result in the total BRAM

resource	used	available	utilization (%)
CLB LUTs	64 330	1 303 680	4.93
CLB Registers	59 549	2 607 360	2.28
CARRY8	1 376	162 960	0.84
F7 Muxes	188	651 840	0.03
F8 Muxes	88	325 920	0.03
BRAMs	94	2 016	4.66
URAMs	8	960	0.83

Table 5.6: Sorting architecture FPGA resource utilization from synthesis.

utilization of 1.49% (down from 4.66%), and increase the URAM utilization to approximately 1.67% (up from 0.83%).

Using the formulas from Section 5.3.3 we find that when the input size doubles to 2^{24}, the memory size for the 2-to-1 mergers and FIFOs increases by approximately 28%, increasing when doubled repeatedly. The CLB LUTs and CLB Registers of the merge tree, which take approximately 40 and 50% of the overall sorting architecture, increase by approximately 12.5% every doubling of the input size.

5.3.4. Remarks

In Section 5.2.3 we explored the throughput of the sorting architecture and found that it is lower when the input is sorted. This was solved by using deeper FIFOs. However, the deeper FIFOs come at the cost of additional resources. If a sorted bucket is not read from memory in a sequential manner but a scrambled manner instead, the input to the merge tree is no longer sorted. The elements will be read from the different FIFOs more uniformly as a result, unless the scrambling method sorts the input. Less deep FIFOs reduce the resource consumption of the tree and thus the sorting architecture.

6

Conclusion and future work

In this chapter we discuss the conclusions in Section 6.1. Recommendations for future work, including improvements to the work presented, are discussed in Section 6.2.

6.1. Conclusion

In this thesis, we aimed to design a high-throughput FPGA accelerator for large sorts using HBM and OpenCAPI. The sorting algorithm chosen is the sample sort or bucket sort algorithm, which consists of two steps. The first step is to partition the input into buckets, the second step is to sort each bucket. We presented two architectures for the partitioning step and one architecture for the sorting step. Multiple instances of these architectures (engines) can operate in parallel. The engines themselves are designed to output 1 element/cycle. Synthesis results show that we can run multiple engines on the FPGA, for a linear increase in throughput.

The achieved throughput of partitioning architecture 1 was lower than expected when the input was random at 0.44 GB/s because of how it writes the data to HBM (at 225 MHz). This was improved by changing the way addresses map into physical memory to 0.88 GB/s, but remains lower than the 3.6 GB/s target. Partitioning architecture 2 has a different write pattern and did not show the problem of architecture 1. It shows a throughput of around 1.7 GB/s, very close to the expected throughput of 1.8 GB/s. Both partitioning architectures should be modified to aggregate writes to HBM with a small buffer. With this modification they should both be able to come much closer to their design targets of 1 element/cycle, corresponding to 3.6 GB/s at 225 MHz. For the sorting architecture a throughput of around 2.7 GB/s is achieved. This is higher than the best throughput of both partitioning architectures because the write behavior to HBM is sequential, additionally allowing writes to be aggregated.

Synthesis results show that it is possible to operate multiple engines in parallel. Partitioning architecture 1 utilizes approximately 1.02% of the available resources while partitioning architecture 2 utilizes 1.23%. The resource consumption of the sorting architecture is higher at 4.93%. This is higher but still low overall, and (parametric) modifications can reduce this. Such a reduction is not necessarily required, since with 10 engines the total throughput is around 27 GB/s, more than the OpenCAPI bandwidth of 25 GB/s.

6.2. Future work

- **Implementation**
  The results presented are obtained from simulation and synthesis, not from implementation on the FPGA. A first step is to implement the different architectures. We anticipate that it may be necessary to make some changes to obtain timing closure, specifically for the sorter.
- **Integrate with OpenCAPI & SNAP**
  When the architectures are implemented on the FPGA, the designs should be integrated with a controller to perform reads and writes to and from main memory, using a compatible version of the SNAP framework.
- **Obtaining good splitters**
  The assumption was made that the splitters reasonably split the input data into the buckets. Whether this is true depends on the input data and how the splitters are obtained. Exploring how these can be obtained with a given input set is very relevant. The sample sort, for example, obtains splitters by sampling the input data. A strategy to get better splitters is to oversample.
- **bucketFinder resource improvement**
  The bucketFinder module as described in Section 4.1.3 uses onboard FPGA memory resources, such as distributed RAM, BRAM and potentially URAM. The latter two memories each have two ports, both of which can be used for reading and writing. The splitters are written once during initialization and remain constant during the execution of the partitioning: only reads to those memories are performed. The port that was being used for writing can be used as a second read port. By utilizing both ports for reading when partitioning, the BRAM and URAM resources used can be halved. Pairs of bucketFinders can share their memory, instead of using their own. In architecture 1 and 2, the memory can be shared across engines. In architecture 2, the memory can also be shared within a single engine because it uses two bucketFinder modules each.
- **Architecture 1 pre-HBM write buffers**
  In the same way that HBM is utilized as a write buffer for main memory, a per bucket write buffer for HBM can be used to increase performance. At the very shallow depth of 1 element per bucket, the throughput of the engine can be doubled. With a depth of 1, two elements can be written to HBM at a time corresponding to the minimum transaction length of 32 bytes. A deeper write buffer is expected to further increase the throughput until the architecture becomes limited by the design target of 1 element/cycle and its frequency. The extra write buffer does add a bit of complexity, as the write buffers must be emptied when the partitioning step finishes.
- **Architecture 1 multiple flushing queues**
  An improvement to architecture 1 is to replace the single flushing queue with multiple queues. This can help prevent situations where the input stage must stall if one of the HBM write buffers becomes full. With multiple queues, one could be a dedicated high priority queue, a medium priority queue and a low priority queue. Flushing operations can then be queued to the desired queue depending on the current number of elements in that buffer. In this way, buffers that are almost full are flushed before less filled buffers. Note that there exists a trade-off between FLUSHSIZE and BUFFERSIZE. FLUSHSIZE is ideally large, but to allow for low/medium/high priority queues we need BUFFERSIZE to be a multiple of FLUSHSIZE.

- **Architecture 2 pre-HBM write buffers**
  Very similar to the suggestion of adding pre-HBM write buffers to architecture 1, architecture 2 can also benefit from write buffers. Although HBM is not organized into fixed-size write buffers in architecture 2, the addition of per-bucket write buffers will increase throughput for this architecture.
- **Architecture 2 batch merging**
  Although suggested in Section 4.3.1, the implementation of architecture 2, as described in Section 4.3.3, does not merge multiple batches during the flushing (to main memory) stage. The design does increase in complexity by merging the output of multiple engines, but merging the batches helps when reading the data back in the sorting step. Additionally, some of the administration (per batch bucket counters) can be reduced, saving storage and bandwidth.
- **Architecture 1 and 2 HBM memory management**
  The previously mentioned pre-HBM write buffers can be seen as a simple form of memory management. In addition to this improvement, more intelligently scheduling HBM write transactions can increase bandwidth.
- **Sorting architecture HBM resource improvement**
  The current implementation of the sorter uses the available memory inefficiently. It writes into HBM twice in two equally sized sections. The first time data is written is the result of a relatively small merge, the second time data is written is the result of a much larger merge operation. As a result, most of the available memory in the first pass is not utilized. The design can be modified to fully utilize one or two pseudo channels, as was described in Section 4.4.1.
- **Implementation: make use of URAM technology**
  Some of the memories in the designs can become large, depending on the design parameters. Some designs may have to be adjusted to have memories with a higher latency. They could be written to make use of URAM, as was described in Section 3.1.3. For example, with a small modification, the bucketFinder can utilize URAM. This is especially beneficial in the later levels, when the required memory becomes relatively large.
- **Utilize multiple HBM pseudo channels**
  All architectures are fixed to use one or two pseudo channels of HBM. To saturate the OpenCAPI bandwidth, the total number of engines will likely not use all the available pseudo channels. By using Xilinx' switching network for global access it is possible to use more memories. However, having the architectures themselves use multiple pseudo channel interfaces may be beneficial for throughput, allowing for an interleaved memory organization.

Bibliography

[1] AMBA AXI and ACE Protocol Specification. URL https://developer.arm.com/docs/ihi0022/latest/amba-axi-and-ace-protocol-specification.

[2] JEDEC. High Bandwidth Memory (HBM) DRAM. URL https://www.jedec.org/standards-documents/docs/jesd235a.

[3] Xilinx. UltraScale Architecture Configurable Logic Block, 2017. URL https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf.

[4] JEDEC Updates Groundbreaking High Bandwidth Memory (HBM) Standard., 2018. URL https://www.jedec.org/news/pressreleases/jedec-updates-groundbreaking-high-bandwidth-memory-hbm-standard-0.

[5] SK hynix Develops World's Fastest High Bandwidth Memory, HBM2E, 2019. URL https://news.skhynix.com/sk-hynix-develops-worlds-fastest-high-bandwidth-memory-hbm2e/.

[6] Xilinx. UltraScale Architecture and Product Data Sheet: Overview, 2019. URL https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf.

[7] Xilinx. AXI High Bandwidth Memory Controller v1.0, 2019. URL https://www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf.

[8] Xilinx. UltraScale Architecture Memory Resources, 2019. URL https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf.

[9] Dip Sankar Banerjee, Parikshit Sakurikar, and Kishore Kothapalli. Fast, scalable parallel comparison sort on hybrid multicore architectures. *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW 2013*, pages 1060–1069, 2013. doi: 10.1109/IPDPSW.2013.129.

[10] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring)*, page 307, New York, New York, USA, 1968. ACM Press. doi: 10.1145/1468075.1468121. URL http://portal.acm.org/citation.cfm?doid=1468075.1468121.

[11] Bin Dong, Surendra Byna, and Kesheng Wu. SDS-Sort: Scalable Dynamic Skew-aware Parallel Sorting. *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC16)*, pages 57–68, 2016. doi: 10.1145/2907294.2907300.

[12] Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. In-memory database acceleration on FPGAs: a survey. *The VLDB Journal*, 29(1):33–59, 2020. ISSN 0949-877X. doi: 10.1007/s00778-019-00581-w.

[13] Tom Feist. Xilinx. Vivado Design Suite. *White Paper*, 2012. URL https://www.xilinx.com/support/documentation/white_papers/wp416-Vivado-Design-Suite.pdf.

[14] Gordon C. Fossum, Ting Wang, and H. Peter Hofstee. A 64-GB Sort at 28 GB/s on a 4-GPU POWER9 Node for Uniformly-Distributed 16-Byte Records with 8-Byte Keys. volume 3, pages 373–386. 2018. ISBN 9783030024659. doi: 10.1007/978-3-030-02465-9_25. URL http://link.springer.com/10.1007/978-3-030-02465-9{_}25.

[15] Jin Kim. Samsung. The future of graphic and mobile memory for new applications, 2016. URL https://www.hotchips.org/wp-content/uploads/hc_archives/hc28/HC28.21-Tutorial-Epub/HC28.21.1-Next-Gen-Memory-Epub/HC28.21.122-Next-Gen-Mem-GPU-Kim-SAMSUNG-v02-t1-3.pdf.

[16] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998. ISBN 0201896850.

[17] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. GPU sample sort. *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, pages 1–10, 2010. doi: 10.1109/IPDPS.2010.5470444.

[18] Susumu Mashimo, Thiem Van Chu, and Kenji Kise. High-performance hardware merge sorter. *Proceedings - IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2017*, pages 1–8, 2017. doi: 10.1109/FCCM.2017.19.

[19] Makoto Saitoh, Elsayed A. Elsayed, Thiem Van Chu, Susumu Mashimo, and Kenji Kise. A High-Performance and Cost-Effective Hardware Merge Sorter without Feedback Datapath. *Proceedings - 26th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018*, pages 197–204, 2018. doi: 10.1109/FCCM.2018.00038.

[20] Peter Sanders and Sebastian Winkel. Super scalar sample sort. In Susanne Albers and Tomasz Radzik, editors, *Algorithms – ESA 2004*, pages 784–796, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-30140-0.

[21] Deshanand P Singh, Tomasz S Czajkowski, and Andrew Ling. Harnessing the power of FPGAs using altera's OpenCL compiler. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 5–6. ACM, 2013.

[22] Wei Song, Dirk Koch, Mikel Lujan, and Jim Garside. Parallel Hardware Merge Sorter. *Proceedings - 24th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2016*, pages 95–102, 2016. doi: 10.1109/FCCM.2016.34.

[23] C. Spille/pcgameshardware.de. AMD Fiji GPU package with GPU, HBM memory and interposer. URL

http://www.pcgameshardware.de/AMD-Radeon-Grafikkarte-255597/Tests/Radeon-R9-Fury-X-Test-1162693/galerie/2392895/.

[24] Ajitesh Srivastava, Ren Chen, Viktor K. Prasanna, and Charalampos Chelmis. A hybrid design for high performance large-scale sorting on FPGA. *2015 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2015*, 2016. doi: 10.1109/ReConFig.2015.7393322.

[25] Jeffrey Stuecheli. A New Standard for High Performance Memory, Acceleration and Networks. URL http://opencapi.org/2017/04/opencapi-new-standard-high-performance-memory-acceleration-networks/.

[26] Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. CAPI: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7–1, 2015.

[27] Takuma Usui, Thiem Van Chu, and Kenji Kise. A Cost-Effective and Scalable Merge Sorter Tree on FPGAs. *2016 Fourth International Symposium on Computing and Networking (CANDAR)*, pages 47–56, 2016. doi: 10.1109/CANDAR.2016.0023.

[28] Lukas Wenzel, Robert Schmid, Balthasar Martin, Max Plauth, Felix Eberhardt, and Andreas Polze. Getting Started with CAPI SNAP: Hardware Development for Software Engineers. In *European Conference on Parallel Processing*, pages 187–198. Springer, 2018.

[29] Xianwei Zeng. FPGA-Based High Throughput Merge Sorter. Master's thesis, Delft University of Technology, 2018.