UNIVERSITY OF CALIFORNIA,
IRVINE

Two-Layer Radix Sorter Architecture for a Power-Efficient FPGA Bloom Filter

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Tarun Sai Ganesh Nerella

2022

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my Advisor, Prof. Sang-Woo Jun for providing constant support to me throughout this project. He patiently answered all my questions multiple times and helped me guide in the correct direction whenever I was faced with design choice decisions or when I was stuck with BlueSpec programming bugs. His out of the box thinking is what sowed seeds for this project and achieve phenomenal results in the end. Without his time and effort, this project wouldn't have progressed ahead.

I also want to thank Prof. Sangeetha Abdu Jyothi and Prof. Aparna Chandramowlishwaran for kindly accepting my request and being part of my Thesis defense committee. Their valuable thoughts and questions at the end of my defense made me think outside the scope of my work and understand certain design decisions which I wouldn't have thought otherwise.

I thank all the researchers whose publications motivated this work as well as served as a benchmark to compare our results with.

Finally, I thank other group members from my advisor's lab who helped me provide details and give access to other components so that I could use them directly. Their help ensured that my work completed in a timely manner.

ABSTRACT OF THE THESIS

Two-Layer Radix Sorter Architecture for a Power-Efficient FPGA Bloom Filter

By

Tarun Sai Ganesh Nerella

Master of Science in Computer Science

University of California, Irvine, 2022

Assistant Professor Sang-Woo Jun, Chair

Bloom filters are a very important tool for many applications including genomics, where they are used as a compact data structure for counting k-mers, represent de Bruijn graphs, and more. However, their performance is often bound by the large filter size requirement in genomics, and their random-access nature. Although accelerators such as FPGAs and GPUs can easily remove the computation overhead of the multiple hash functions, the random access performance of off-chip memory is still a bottleneck, calling for costly high-performance memory. The solution we propose is BunchBloomer, which improves the cost-effectiveness of FPGA Bloom filter accelerators by making better use of cheaper, lower-power DDR memory. As a part of this project, I work on creating the architecture of a two-layer radix sorter to group table updates into bursts directed to the same 8 KiB memory region, which can be efficiently cached in on-chip memory. The sorters can sustain high performance data ingestion by processing four 32-bit tuples at once, while still utilizing a reasonable amount of chip space. The overall BunchBloomer device achieves much better power efficiency compared to a traditional multicore server or even a conventional FPGA Bloom filter accelerator equipped with Hybrid Memory Cubes.

Chapter 1

Introduction

Genome sequencing and applications continue to become more and more important in today's world. The information obtained from the sequencing is useful to scientists in a lot of ways such as understanding how the genes work together and help in the growth and development of the organism. However, the processing requirements of these applications are very huge and as a result it is extremely important to to keep up with these requirements so that approaches such as predictive medicine [2] and personalized healthcare [18] within the reach of the general public.

The Bloom filter is a space efficient probabilistic data structure that can quickly test whether and element is (possibly) present in a large set. It uses a fixed size bitmap and a set of Hash functions to map data elements to certain bit positions in the bitmap, which can be accessed in O(1) time. Bloom filters are an important tool that are used in several genomics applications such as k-mer counting [15, 12, 13], genome sequencing, de Brujin graph construction [16, 14, 3], etc. These applications utilize the probabilistic nature of the Bloom filter to speed the look up process, which would have been impractically slow had a deterministic data structure or approach been used.

Even though the Bloom filter theoretically has a fast lookup and is space efficient, it is still both computationally and memory intensive data structure. In most of its software implementations, the Hash function computation becomes a prominent bottleneck [10, 8]. Further, although the computation overhead of Hashing can be easily removed by using high-performance accelerators such as FPGAs, as they can typically achieve sufficient hashing performance, the memory, even a good DRAM, becomes a bottleneck in these cases [4] and undermine the processing capacity of the accelerator. As a result, FPGA Bloom filter accelerators typically focused on having small tables which can fit in the fast on-chip memory [7, 11, 4]. This is not a good idea as our target application typically work which very tables which are multiple GiB in size [3]. The other typical solution is to use a fast and power-hungry off-chip memory such a Hybrid Memory Fabric [12, 8]. However, the fabric itself consumes significant power thereby undermining the power efficiency benefits using an FPGA.

The Memory bottleneck is caused due to the Random Access nature of the memory accesses. DRAM (or any other memory) have a fixed row buffer size (say b) which is the lowest granularity at which data and be read/written to the memory. Whenever a memory address is accessed, the DRAM retrieves and returns the entire line of size b. However, because DRAM address space is typically in the order of GiBs, complete random accesses patterns means that there is negligible chance for adjacent memory accesses to belong to the same DRAM access line (or row buffer). Therefore, every random memory access requires the DRAM to retrieve a new access line and wasting the rest of the data that is retrieved apart from the one requested.

To solve this problem, we propose the **BunchBloomer**, an FPGA accelerator that removes the computation overhead (as previously described) by performing the all Hashing on the chip. In addition, we **address the memory bottleneck problem** by using **Hardware Radix-Sorters**, the part on which I worked on, to efficiently remove the random access patterns. The sorters bucket the incoming stream of addresses using a fixed set of 8 (level-1)/7 (level-2) bits and send out the requests to the DRAM in bursts which are directed to the same 8KiB region in memory. The burst size (8KiB) chosen matches the size of the row buffer on our target DDR3 DRAM and, therefore, by making use of good temporal locality, it dramatically reduces the performance impact of the random accesses. We implemented the Radix-Sorter in two layers, to optimize the design due to chip constraints, by connecting the two sorters using an array bucket buffers in DRAM.

We implemented our prototype on the Xilinx KC705 chip that uses a Kintex-7 FPGA and is equipped with 1 GiB of DDR3 DRAM memory. Results have shown that our BunchBloomer is able to outperform a costlier 12-core server based software implementation by over 2x in terms of Throughput achieved and consuming much less power, thereby demonstrating 10x better power efficiency. BunchBloomer also demonstrates competitive performance compared to a published FGPA Bloom filter accelerator equipped with power-hungry Hybrid Memory Cubes, resulting in a 4x better power efficiency.

The rest of the report will proceed as follows:

- Chapter 2 motivates the problem by introducing a target application, and provides necessary background regarding the memory bottleneck and Bloom filters in general.
- Chapter 3 introduces the BunchBloomer and briefly outline the various components used on the chip.
- Chapter 4 details the Radix-Sorter's architecture on which I worked on and outlines the various design tradeoffs that helps us in reaching the final optimized implementation.
- Chapter 5 presents our results and throughput evaluations, both Radix-Sorter specific and overall. Comparisons to software and other implementations are also presented.
- Chapter 6 concludes the report.

Chapter 2

Motivation & Background

In this chapter, we introduce one such target application, genome sequencing, and reason as to why it is computationally expensive and why Bloom filters would be useful in such a case.

2.1 What is Genome sequencing?

Genome sequencing is the process of figuring out the order of the DNA Nucleotides (or bases) such as Adenine (A), Guanine (G), Thymine (T), etc in a human genome. The data obtained from the sequencing is useful to the scientists in a lot of ways. Some of these include:

- Obtaining clues about where the genes are
- Understanding how the genes work together and help in the growth and development of the organism.
- Identify the regulatory part of the genome (outside the genes) which control how the genes are turned on and off.

The following diagram (Fig 2.1) shows an example of a human RNA sequence, consisting of the Nucleotides Guanine (G), Cytosine (C), Uracil (U), Adenine (A), and so on in that order.

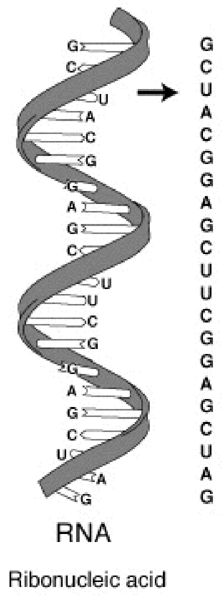

Figure 2.1: An example RNA strand showing the Nucleotides present.

Source: https://en.wikipedia.org/wiki/Nucleic_acid_sequence

2.2 Why is sequencing difficult?

A whole genome is very large to be sequenced all at once and the available DNA sequencing methods can only handle short stretches at a time. Therefore, the large genomes are first broken into small pieces which can be handled by the existing sequencers. The whole process can be divided into two steps:

- Sequencing: The small pieces obtained after breaking the long original genome are sequenced individually to obtain the order of the Nucleotides within them. Several techniques, such as Electrophoresis, Capillary sequencing, etc are used to obtain these individual sequences. Earlier, this process was done manually by hand, however, automatic sequencers are now available can now process more than 50,000 Nucletodies in a few hours.

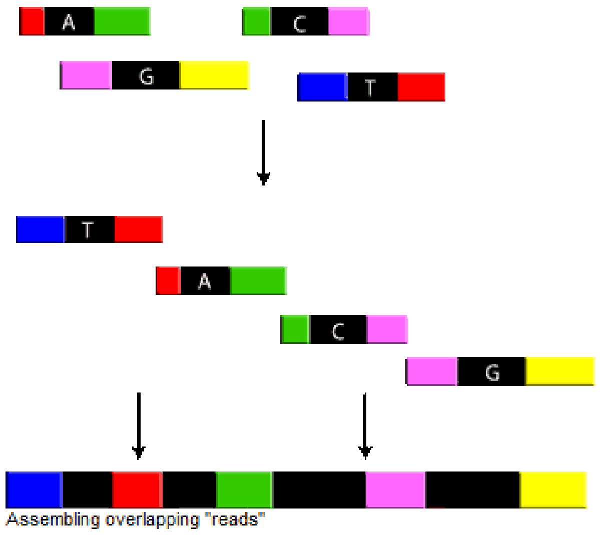

Figure 2.2: The Assembling finds overlapping regions and requires millions of comparisons.

Source: http://www.genomenewsnetwork.org/resources/whats_a_genome/Chp2_3.shtml

- Assembly: This compute heavy step consists of putting all the pieces in the proper order by comparing the overlaps at the end (as shown in Fig 2.2). For this, the assembler compares every read with every other read and methodically place the pieces next to each other to reconstruct back the genome. For a human genome, the comparisons required run into the order of trillions, thereby requiring requiring high performance machines having a lot of memory and consuming a lot of power.

Apart from the use cases mentioned above, genome sequencing is becoming more and more important in this today's world, for example, to identify the new variants of COVID-19. As a result, it is increasingly important to make cheaper and accessible high throughput genomics analysis available to everyone.

2.3 Background

In order to overcome the massive computation requirements and to bring predictive medicine [2] and personalized healthcare [18] within the reach of the general public, performance improvements are being sought in many fronts. These include hardware acceleration using GPUs [5], by creating a software implemented k-mer counting tools and using CUDA GPU programming and algorithm engineering to accelerate the counting step. as well as more effective algorithms and data structures [17] [20]. We will discuss the Bloom filter approach below.

The Bloom filter: A more efficient data structure

The Bloom filter is a space efficient probabilistic data structure that can quickly test whether an element is (possibly) present in a large set. It is extremely useful in many high-throughput analytics applications which would require practically infinite amount of memory if we were to use the deterministic techniques to search for elements.

Bloom filter uses a fixed size bitmap and a set of hash functions to map every element in the set to some particular locations in the bitmaps. The following diagram (Fig 2.3) illustrates the working of a bloom filter.

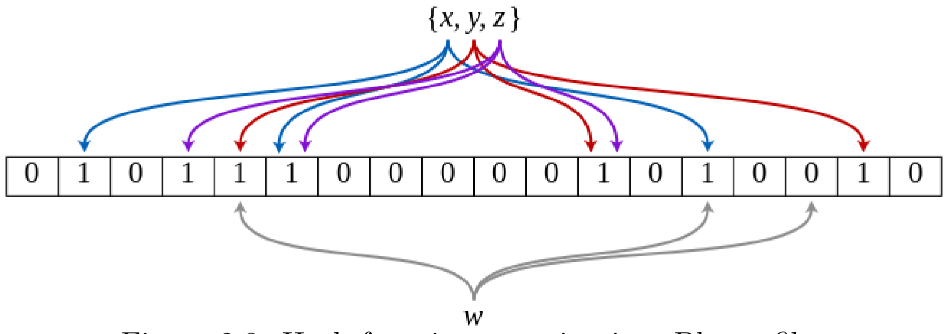

Figure 2.3: Hash function mapping in a Bloom filter

Source: https://www.kdnuggets.com/2016/08/gentle-introduction-bloom-filter.html

In the above example, there are three elements x, y, z in the set, and the bitmap is of 18-bits. The hash functions map each of the elements in the set to 3 out of the possible 18 positions, with x being mapped to bits 2, 6, and 14, y mapped to bits 5, 12, and 17, and z mapped to bits 4, 6, and 12. To search if the element w is present in the set or not, it is passed through the same hash functions and the positions it gets mapped to are checked to see if they are set to 1.

The probabilistic nature of the filter arises from the fact that although we can definitely say that an element is not present in the set if one of the bits is not set to 1, we cannot definitely say if the element is present even if all the bits it maps to are set to 1. Therefore, the results may have a lot of false positives, but cannot have any false negatives.

The bits in the bitmap can be accessed in $O(1)$ time. As a result, Bloom filters are widely used to represent complicated data structures in genomics analytics such as sparse histograms for genome sequencing, k-mer counting [15] or de Bruijn graph construction [16]. In such a use case, Bloom filters for a genome dataset is constructed, and the constructed table is given to downstream analytics software as an analytics tool.

Although Bloom filters can speed up the lookup process significantly, they themselves are a very computationally and memory-intensive data structure during both construction and querying. In software implementations, computing good, non-trivial hash functions is a prominent bottleneck [10, 8].

The computation overhead of Hashing can be easily removed by using high-performance accelerators such as FPGAs. However, memory bottleneck still exists and therefore, although such FPGA accelerators can typically achieve sufficient hashing performance, they become memory-bound [4].

The above diagram (Fig 2.4) illustrates how even a high speed memory such as a DRAM cannot catch up with the speed of processing thereby undermining the processing capability

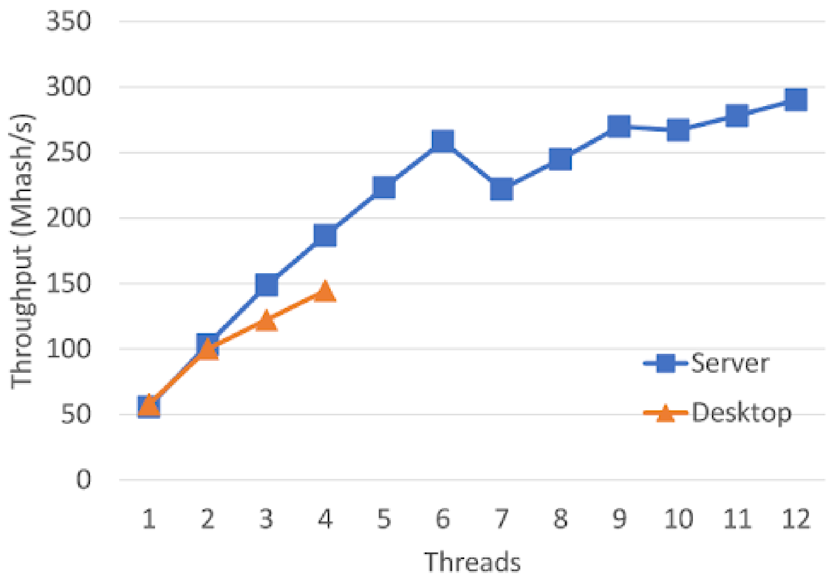

Figure 2.4: Variation of Throughput with the Performance Acceleration (number of threads) demonstrates the Memory bottleneck

of the accelerator. To illustrate, the variation of the throughput is observed as the number of threads is increased using a software implementation on a Desktop machine and a high performance Server. It is observed that initially when the thread count is low, the throughput is limited because of the computation limitation of the system. As the thread count is increased, the system how can achieve sufficient hashing performance. However, after a certain point (thread count of 6), the throughput doesn't scale anymore because memory becomes a bottleneck. To extract useful performance from an accelerator, addressing the memory bottleneck problem is extremely critical.

As a result, FPGA Bloom filter accelerators have typically focused on small tables which can fit in fast on-chip memory [7, 11, 4], or provision fast and power-hungry off-chip memory fabric [12, 8]. For genomics applications such as de Bruijn graph construction which require large tables which are multiple GiB in size [3], the only way to avoid under-utilizing accelerator performance is to couple it with fast memory, potentially undermining the power efficiency benefits of FPGA acceleration. We will address this problem in this project.

Chapter 3

Overview of BunchBloomer

3.1 Approach

We address the memory performance problem of Bloom filter construction with BunchBloomer, which uses hardware radix sorters to efficiently remove random accesses. The bursting radix sorter accelerator design collects bit update requests into bursts directed to the same 8 KiB memory region, which is same the size of the row buffer on our target DDR3 DRAM. Each 8 KiB region can be cached in on-chip BRAM with good temporal locality, dramatically reducing the costly performance impact of random access into DRAM.

The following diagram (Fig 3.1) illustrates the significant performance improvement that can be achieved as the memory accesses become more and more localized. 64B is the complete random access case, and the 8KB is the maximum line size that can be accessed at once from DDR3 memory. There is almost a 10x improvement in the DRAM bandwidth by increasing the burst size from 64B to 8KiB. That certainly indicates that Sorting would help, but of course at the cost of increased computation overhead. So, our solution as we stated is to FPGA sorting. We experiment by varying the burst size and observing the bandwidth of the DRAM.

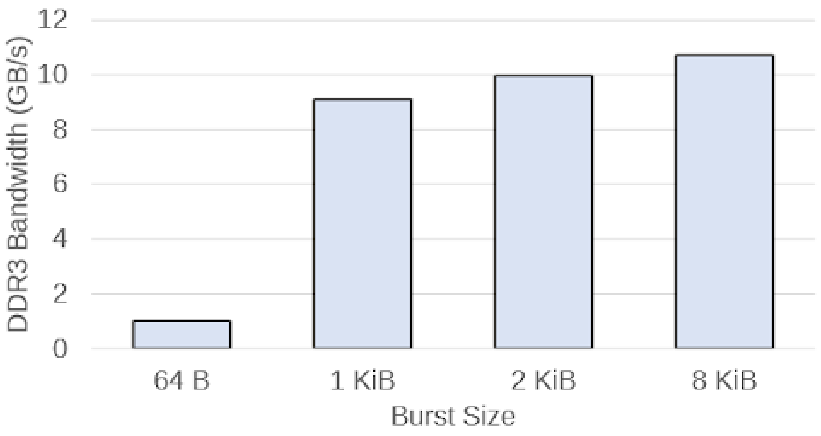

Figure 3.1: Variation of the DDR3 DRAM Bandwidth with the Burst size.

Due to chip resource constraints, the radix sorter is implemented in two layers, connected over an array of bucket buffers in DRAM. The second layer sorter emits bursts of up to two thousands update requests targeting the same memory region. The maximum burst length is the result of available on-chip BRAM resources, and BunchBloomer can create longer bursts if instantiated on a larger FPGA chip.

3.2 Architecture

The overall architecture of the BunchBloomer system is shown in the diagram below (Fig 3.2). There are roughly four sub-parts that the whole system can be divided into. All the components in the system are connected via a 128-bit datapath.

- The Hashing: The input to the system is a stream of genome encodings, which are passed through a set of hash functions (for the reasons described in the Background & Motivation section. There are 8 hash functions implemented on the chip each with a fully pipelined implementation. These hash functions emit four 32-bit hash values in

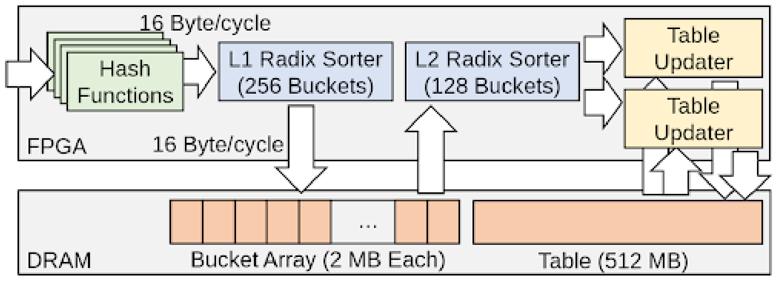

Figure 3.2: Architecture of the Bunch Bloomer.

one cycle thereby fully utilizing the available bandwidth.

- The Radix Sorting: The radix sorters on which I mainly work on are an important intermediary in the pipeline to bucket and store the incoming requests so that they processed together in bursts. The system consists of 2 radix sorters. The level 1 radix sorter is an 8-bit sorter, which uses the top 8 bits to sort/bucket the stream. Since the range of an 8-bit mapping is from 0 to 255, the sorter uses 256 buckets to store the addresses.

  Likewise, the second radix sorter uses the next 7-bits to sort the stream further on the data received from the level-1 sorter. This sorter uses 128 buckets (because the 7-bit maps to 0-127). Once the top 16-bits have been sorted (the reason why the remaining 1 bit is not used is explained later), bursts emitted from the level-2 sorter are sent to the table updater.

- Table Updates: The table updater receives the buckets of adjacent memory addresses and updates them in the table present in the DRAM. There are two table updaters present on the chip to speed up the process and the bursts are diverted to one of the table updaters based on the value of the 1 unused bit.

- The Memory: The target chip (KC705 FPGA) has 1 GB of DDR-3 memory available. The entire memory is divided into two parts, using half of it for storing buffers, and the remaining 512MB memory to store the table. As discussed later, the buffers act as the communication between the two radix sorters.

Chapter 4

Radix Sorter Architecture

The radix sorter buckets the incoming stream of 32-bit addresses by separating them based on the value of certain 7/8 bits. The first radix sorter uses bits 31 to 24, bucketing them into 256 buckets whereas the second radix sorter uses bits 23 to 17 and buckets them further into 128 buckets. The architecture and the design choices that went into each of the radix-sorters are designed below.

4.1 How does Radix Sorting work?

As discussed earlier, the input to a radix-sorter is an incoming stream of 32-bits hashed memory addresses. Although there is an occasional skew in the data, these memory addresses can be technically completely random in nature. The following diagram shows a 32-bit address map.

Lets consider the unsorted stream of such addresses. Since the number of bits in the address is 32, the value variation that the bits can map to is from 0 to $2^{32} - 1$. Therefore the value corresponds to an address space variation of 4 GB.

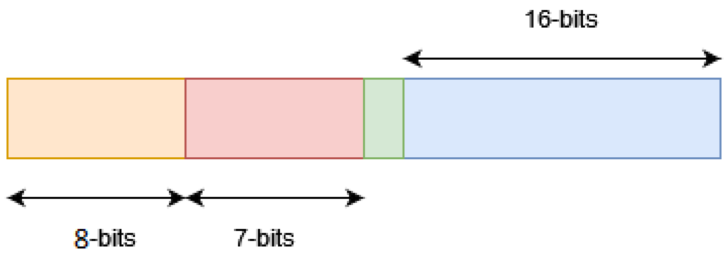

Figure 4.1: Bitwise breakdown for sorting of the 32-bit Memory Address

DRAM Row Buffer: The DDR3 memory present on the chip has row buffer size of 8KB. That means that whenever an element on a line is accessed, the entire 8KB row is read from the DRAM memory into the DRAM row buffer in the controller.

However, because of the random access nature of these addresses and the range of the address space it maps to, it can be safely assumed that no two adjacent memory accesses will belong to the same line in the DRAM. That effectively means that every such memory access will pull out a new line, of which only 1 bit data is used and the rest of data is discarded (eventually).

However, consider grouping these accesses based on the bits shown in the above diagram (Fig 4.1). The level-1 radix sorter uses the top eight bits to store the memory addresses into 8 buckets. The level-2 radix sorter further sorts them using the next 7 bits and stores them into 128 buckets.

What does the extra bit do? There are two table updaters on the chip, and the bit shown in green is used to direct the the bursts coming from the radix sorters to one of the two updaters. Therefore, the bit won't be counted as a random access bit (because it doesn't contribute to the address space variation from the point of view of one updater).

Now let us consider the sorted address stream. Since the top 8 bits are sorted by level-1 sorter, the next 7 bits are sorted by level-2 sorter, and the extra bit is used to decide the table updater, the address space variation now arises only due to the lower 16-bits. This maps to a value of 2^{16} which equals 64Kb or 8KB.

This perfectly matches the line size of the DRAM. Hence, if the sorted addresses are processed in bursts, each of these bursts will map to the same 8KB region and there is a very high chance of using only entire access line, and without the need to access one line for every memory access as in the previous unsorted case. Hence the throughput of the system will be significantly improved.

4.2 Architecture

There are several design details that had to be addressed and decisions that had to be taken. They are outlined one by one in this report.

Design Tradeoff 1: First was the data width, which is the number of incoming bits that could be processed at once. Processing every 32-bit number sequentially would lead to a very low throughput. Although the peak memory bandwidth for the chip is 512-bit, the data pipeline bandwidth of 512-bit isn't feasible because the logic for such a radix sorter was too large to fit on the chip we are targeting. Also, the peak bandwidth would be available only in the ideal case. In reality, multiple components would be accessing the memory simultaneously and the memory bandwidth is split between them. Therefore, after some experimentation and to be in line with the requirements of the rest of the architecture, we found a middle ground of 128-bit bandwidth so that each of the radix sorter occupies decent amount of space.

Design Tradeoff 2: The second design choice was about the number of sub-sorters within in the radix sorter. This logic determines the appropriate queue or bucket (numbered from 0 to 128) into which the 32-bit element has to be inserted. The simplest choice was to independently implement the circuit for each of the buckets, each of which just checking whether the given element belongs to the particular queue or not. However, again, the amount of chip space required to implement 128 such units was too large to fit onto the chip. Therefore, we decided to use only 8 such units, each of which handles data going into 16 of the queues

Also, since the input arrives in a sequential manner as a stream of 128-bits (with 4 hash addresses), we have implemented a network of input queues with a single point of ingestion. Each of these queues then propagate the data element forward until it reaches the the last buffer.

The following diagram (Fig 4.2) shows the entire internal architecture of a single radix sorter.

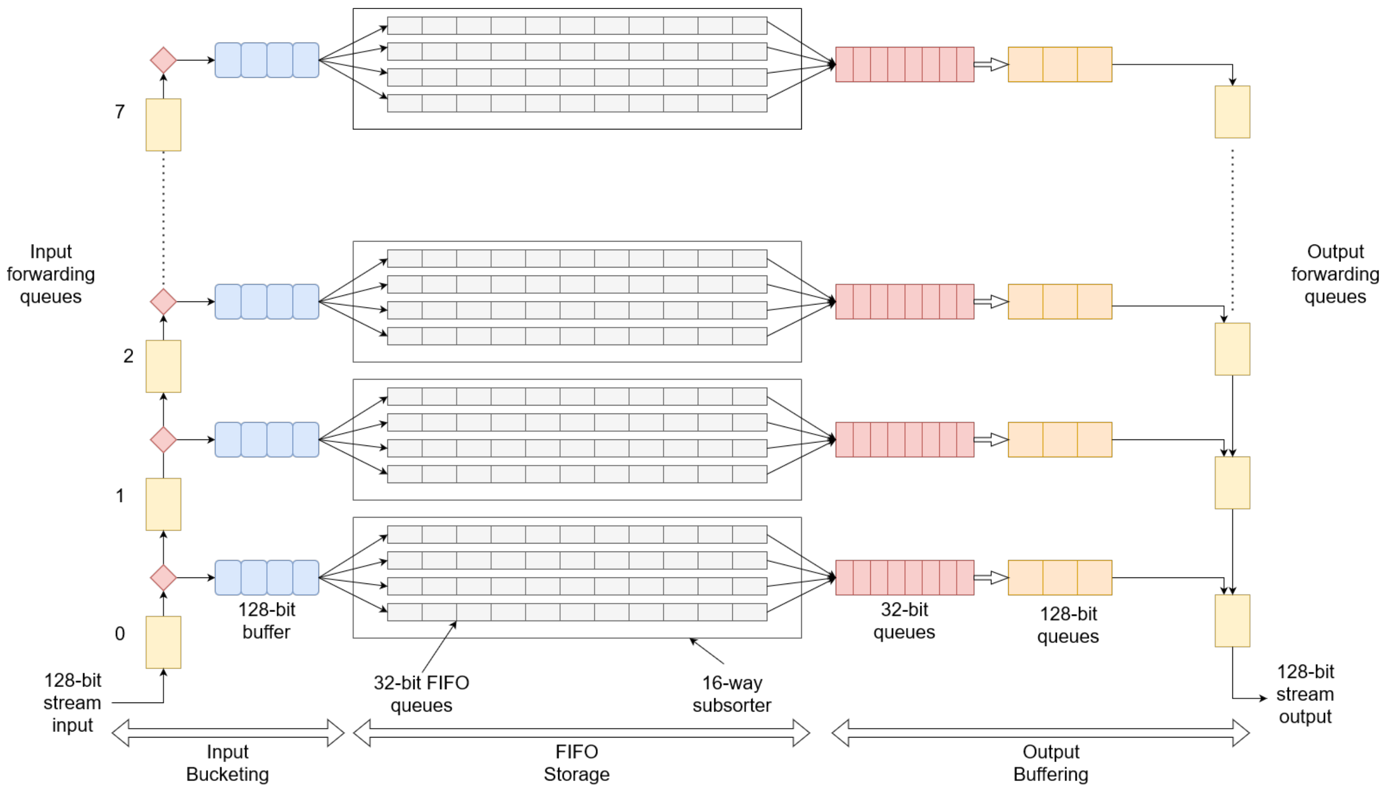

Figure 4.2: Internal Structure of a 7-bit Radix Sorter.

The structure can be divided into 3 different parts, the input part where the sorting happens, the storage part where addresses in the same bucket are stored together, and the output part which merges all the outgoing streams together. As shown in the diagram, the unsorted incoming 128-bit stream enters the lowest-ordered input forwarding queue, and the outgoing sorted 128-bit stream exits from the lowest-ordered output forwarding queue.

Each of these parts are discussed in detail in the chapter.

Step 1: Input Bucketing

The components used in the input sorting part are the input forwarding queues, and the 128-bit buffers. The following diagram (Fig 4.3) shows this part of the circuit. We discuss the utility of these parts below:

- **Input Forwarding Queues:** The sorter consists of 8 input forwarding queues into stream the incoming 128-bit elements. Once a 128-bit element is inserted into the lowest forwarding queue, the element is checked if any of its 4 32-bit addresses belong to this particular sub-sorter or it. For instance, in the first forwarding queue it is checked if any of the 4 elements belong to the range 0-15 (indices handled by the first sub-sorter). If yes, a copy of the entire 128-bit is sent into the 128-bit buffer for further processing. Additionally, the element is also forwarded to the next forwarding queue. This process is repeated until the element reaches the last input forwarding queue.
- **128-bit Buffers:** There are 8 128-bit buffers present in the radix sorter, each corresponding to a particular Whenever an element is inserted into a particular buffer, it means that atleast one out of the four 32-bit addresses present in the element belong to that particular sub-sorter. For example, an element (128-bit) in the buffer of the first sub-sorter contains atleast one (32-bit) address whose bits map to 0 to 15. Each of the four addresses are processed sequentially, one per cycle, and if matched are sent to the appropriate FIFO storage queues. Although processed sequentially, these buffers op-

erate separately from the input forwarding queues, and therefore it is ensured that the original pipeline is not stopped (unless there is a local skew in data, which is discussed later in the results section).

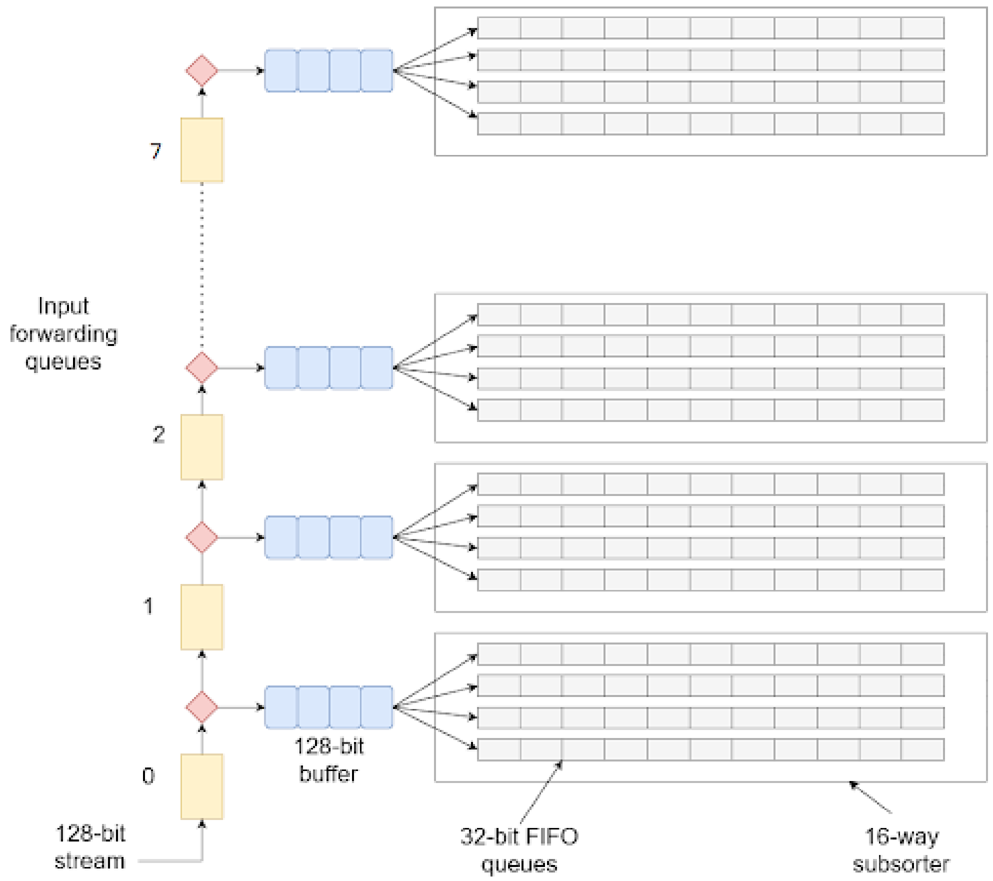

Figure 4.3: Input circuit showing the Sorting and Buckets

Step 2: Storage and Flushing

The components used in the storage and the output stream handling are the 32-bit buffer queues, 128-bit buffer queues, and the output forwarding queues. The following diagram (Fig 4.4) shows this part of the circuit. We discuss the utility of these parts below:

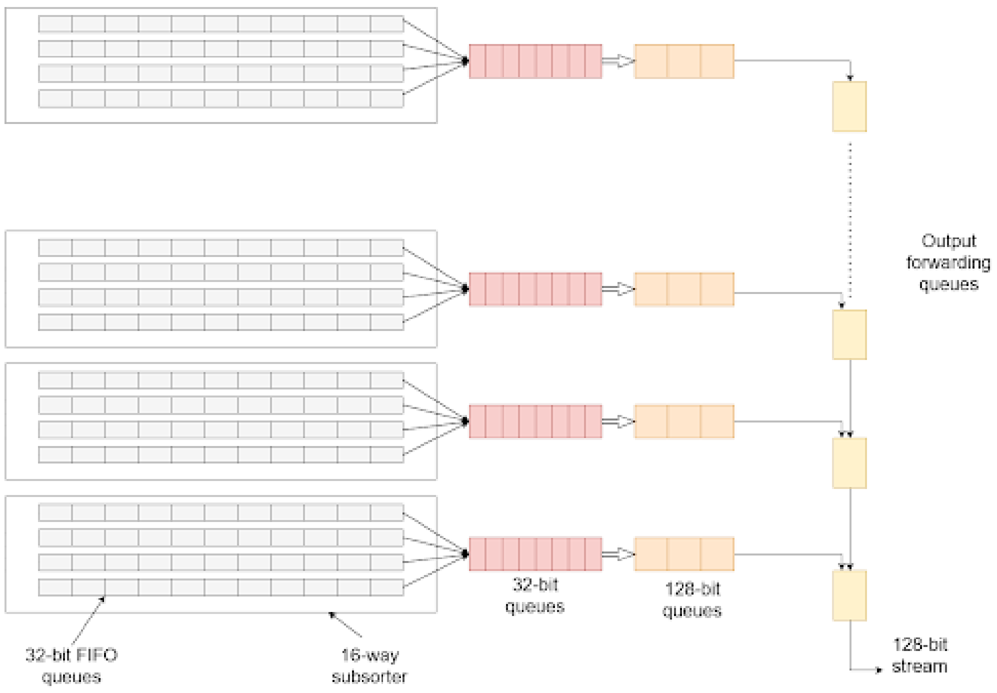

Figure 4.4: Output Circuit for merging the Bursts emitted

- **Storage FIFO Queues:** The entire sorter has a total of 128 32-bit FIFO storage queues to store the addresses and process them in bursts. For the efficient usage of the chip, the the design had been modelled as a 2-Dimensional structure, consisting of 8 sub-sorters each of which contain 16 storage queues. Addresses are stored in each queue until the size of the queue reaches 1024B or 1KB. Once the size is reached, the addresses are burst out at once into the 32-bit buffer queues.

- **32-bit and 128-bit Buffer Queues:** There are 8 32-bit buffer queues and 8 128-bit buffer queues present in the output. Each 32-bit buffer queue receives bursts of size

1KB from the 16 storage queues that are present in that particular sub-sorter. These addresses are de-serialized back into 128-bit elements using and stored in the 128-bit buffer queues so that the output doesn't get interleaved.

- **Output Forwarding Queues:** Output queues use locking mechanism for synchrony between the different sub-sorters. During idle state, every forwarding queue (say O_i) checks if the Buffer (B_i) or the next forwarding queue (O_{i+1}) to see if there is any data bursts are present. If any of them contains data to be cleared, lock is applied on O_i so that all the data in B_i is first cleared into O_i. Likewise, if O_{i+1} contains data, then the lock on O_i is held by O_{i+1} to drain all its data.

  This process is repeated until all the data reaches the lowest forwarding queue (O_0) which drains out the data from the Radix Sorter.

4.3 2D vs 1D Architecture

Design Tradeoff 3: A second major design choice was to implement a 1-Dimensional system of Storage queues as opposed to the Hierarchical 2D structure described before. The following diagram (Fig 4.5) illustrates such an architecture. In this design, every storage queue is processed independently and, although it eliminates the need for a 128-bit Buffer on the input side, it requires a separate Input Forwarding Queue on the input side and separate 32-bit and 128-bit Buffer Queues and Output Forwarding queues on the output side.

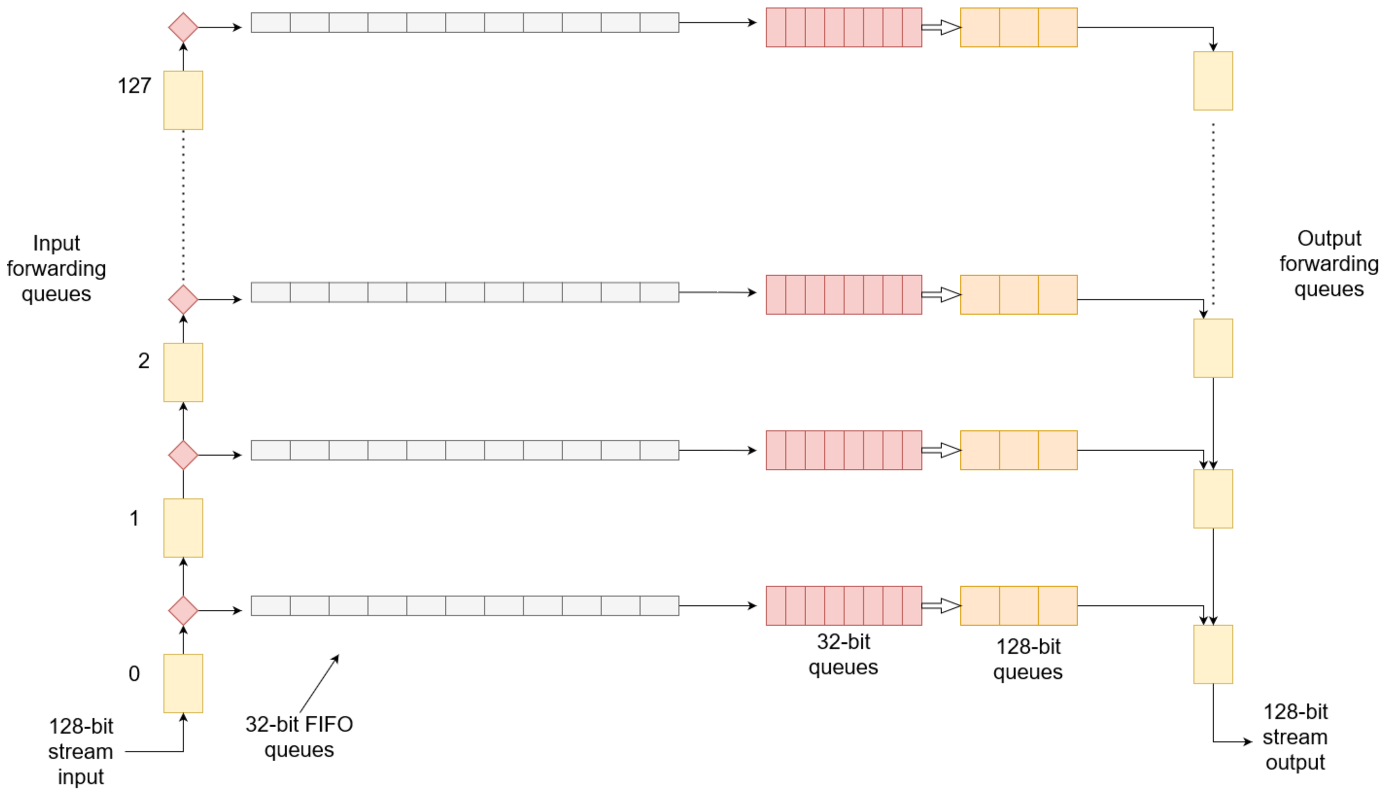

Figure 4.5: The one-dimensional 7-bit Radix Sorter architecture

Although the sorting/bucketing logic is simpler to implement in this case, such an architecture will be take up more space because of the presence of 128 meta-queues (Forwarding Queues and Buffering Queues) where as in the 2D architecture, they are just 8 in number.

The following are some of the parameters that were measured, which deemed this design impractical for our target application.

- Chip space: Because of the presence of independent meta-queues for every Storage Queue (Bucket), this Radix Sorter Design takes up more than 60% of the chip space. Therefore, it wouldn't be possible to fit two Radix-Sorters on the chip along with the other components which were described in the Overall Architecture section.

- Latency: Since the forwarding queues forward elements one by one every cycle, the maximum latency of an to reach a Bucket or Storage queue is 128 cycles, whereas the 2D Hierarchical architecture would only need 8 cycles. Considering a similar delay due to the output forwarding, the worst case ejection latency would be 256 cycles as compared to just 16 indicating that 2D approach would be much better.

- Rule conflicts: Although the 1D architecture can be optimized by using only 8 forwarding and buffer queues, this leads to rule conflicts in BlueSpec because multiple rules will be trying to access the same Storage queue simultaneously.

4.4 Connecting the Radix Sorters

The two Radix Sorters are connected back to back using buffers in the memory as illustrated in the diagram below (Fig 4.6).

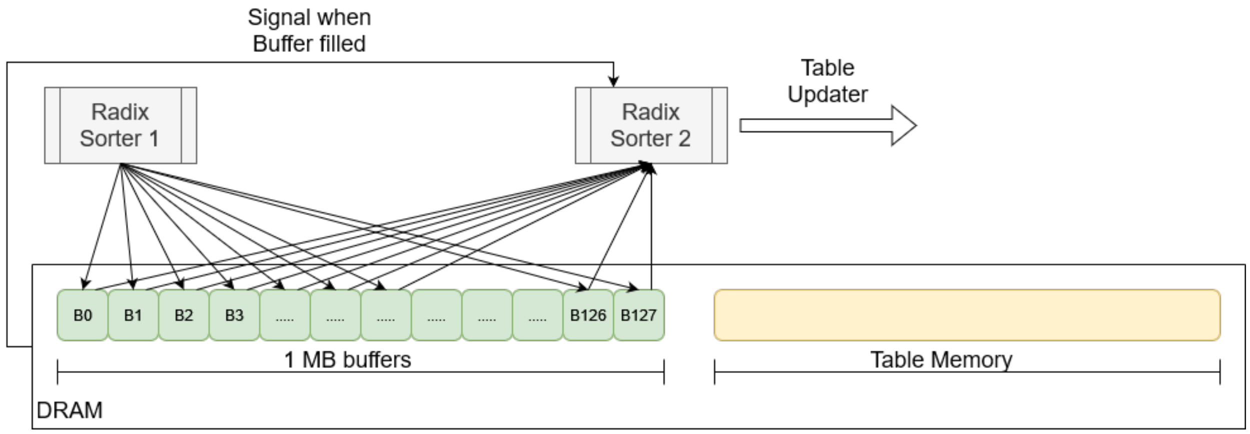

Figure 4.6: Bucket Buffers in the memory for connecting the two Radix Sorters

The available 1GB DDR3 DRAM memory is partitioned into two parts, and the first part is used to create 128 identical buffers of 1 MB each. The remaining part of the memory is used for storing the actual table. These buffers are used to store the sorted burst of addresses that are received from the first radix sorter temporarily before they are ingested by the second Radix-Sorter.

The first radix sorter maintains an offset for each of the memory buffers. As the bursts are sent out from the storage queues, they are stored in the corresponding Memory buffers and the offset is updated accordingly. The second radix sorter is signalled periodically so as to ensure that the buffers don't overflow.

Once the offset of a buffer reaches the midway point (512KB), the second radix sorter is signalled to ingest the first half of the buffer simultaneously as the first radix sorter writes to the later half of the buffer. Likewise, when the offset reaches the end of the buffer, it wraps around to the start of the buffer again and the second radix sorter is now signalled to clear the later part of the buffer. Signalling at the half way point ensures that the data is not overwritten by the first radix sorter before it can be ingested by the second sorter, which otherwise would require the first sorter to wait until the second radix sorter ingests the entire data effectively stopping the entire pipeline.

4.5 Single Level vs Two level Sorting

Consider an architecture design in which a single radix-sorter is capable of sorting all the 16-bits at once, instead of the 2-level approach that was described earlier. In such a case, the number of buckets that would be required be $2^{16} = 65$k. Since the number of FIFOs and the extra meta-buffers (on the input side and the output side) proportionally increase with the number of buckets, this design would drastically increase the chip occupancy making it impossible for other components to fit on the chip. Further, as explained earlier, such a design would also have a very large latency (in the worst case, an element would require 65k cycles to reach its corresponding storage queue).

Considering all the design tradeoffs that very were described so far, the final decision was to split the sorting part into two-levels, each store sorting half of the bits, and having a Hierarchical 2D structure of sub-sorters to optimize it further.

4.6 DRAM Arbiter

The DRAM present on the chip exposes a common interface to read and write data into the memory. This is achieved by inserting read/write requests into the request queue of the DRAM, which are then processed in order by the DRAM controller.

However, when multiple components use the DRAM simultaneously, the requests from them get interleaved in the DRAM request queue and therefore every component ends up with incorrect data. To resolve this issue, we use a DRAM Arbiter which segregates the requests per user by buffering their requests. The following diagram (Fig 4.7) illustrates the functioning of the Arbiter when three users, the Sorter, Updater 1, and Updater 2 are accessing the DRAM simultaneously.

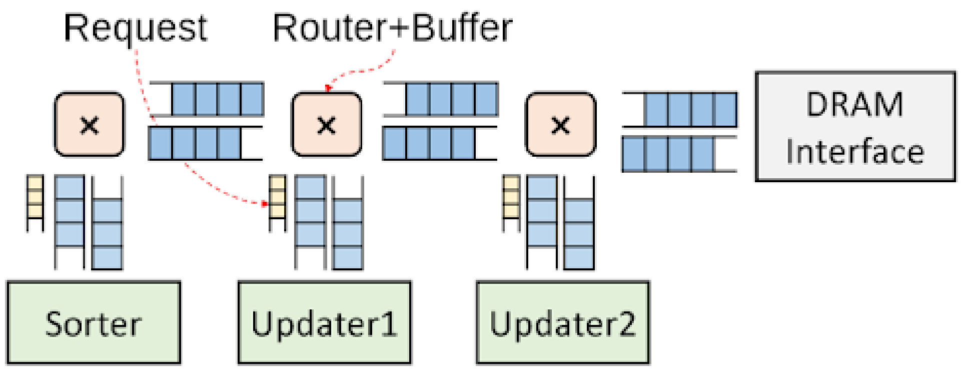

Figure 4.7: Internal structure of the DRAM Arbiter

The requests from each user are buffered in queues and are burst/sent together to the DRAM interface through a series of Buffers and Routers. Once the requests are processed, they are redirected back through the same set of Router + Buffer chains to the user. The bursting operation can be visualized very similar to what happens inside the radix sorter. This operation ensures that the requests of different users don't get interleaved anymore.

Chapter 5

Evaluation & Results

The sorted data coming out of the radix sorters arrives in bursts of size 8KB each. The burst size of 8KB matches the access line size of the DRAM and therefore ensures that most of the adjacent data is accessed from the same DRAM access line.

The following diagram (Fig 5.1) illustrates the overall sorting process.

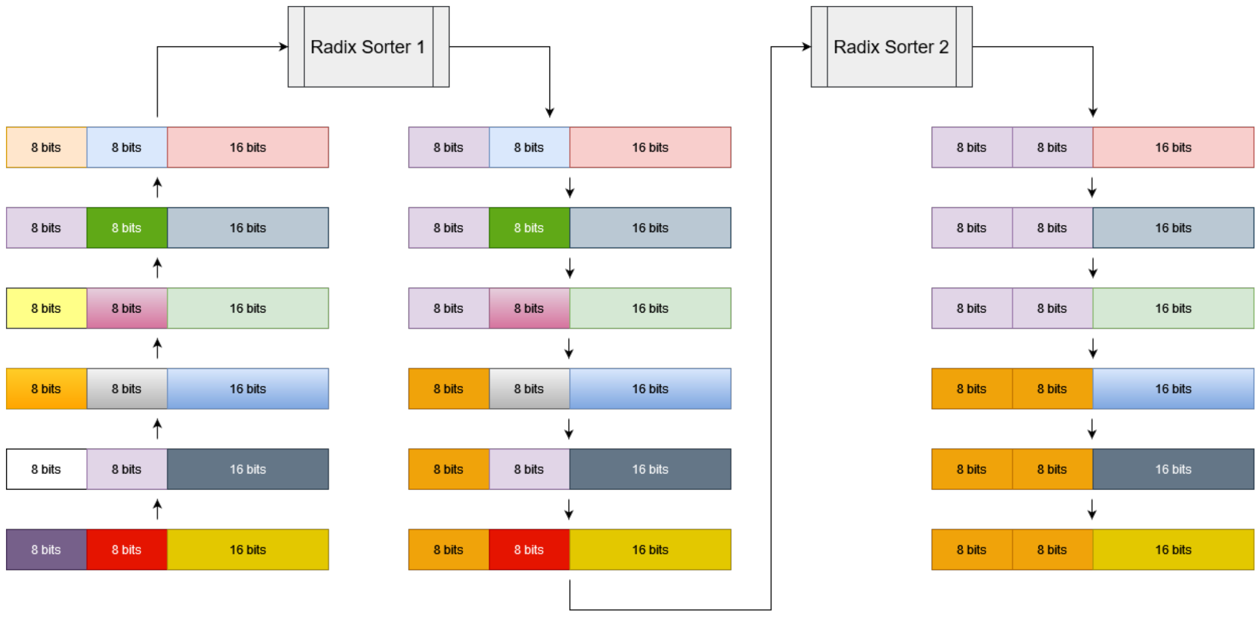

Figure 5.1: The overall Sorting processed in Bursts

The initial input stream of entering the Radix Sorter 1 is completely unsorted and therefore map to an address range of 4GB. The Radix sorter 1 sorts the top 8 bits and the stream coming out of it contains bursts within which all the addresses have the value at the top 8 bits. These bursts now have a partially reduced address space variation of $2^{24} = 2$ MB. The stream is sorted further by passing it through the second radix sorter that further sorts the stream, and all the 8KB bursts coming out contain addresses contain the same value of the top 16 bits. These bursts now have a reduced address space variation of just $2^{16} = 8$KB.

The two crucial parameters that we optimized the design for the radix sorter part are the Chip utilization and the Throughput. These results are described below. Further, the overall throughput of the entire chip and the power consumption are also described.

5.1 Radix Sorter Results

The design was tested on the **Xilinx KC705** chip using a **Kintex-7 FPGA**. Although the chip is relatively small and old, we target it for its low cost and high power efficiency.

5.1.1 Chip Utilization

Overall both the sorters combined more than 60% of the available chip space. The following table (Table 5.1) shows the percentage utilization of the chip in terms of the number of LUTs and the RAMB36 usage.

We can see that the LUT utilization of the first sorter is 43% whereas the second sorter uses only 16.2%. The main reason as to why this might be happening is because the first sorter sorts using 8 bits, and therefore has 256 buckets, whereas the second radix sorter uses only 7 bits to sort and thus has 128 buckets. Therefore, compared to the first sorter, the second

$Module$	LUTs	RAMB36	RAMB18
$Sorter1$	130,412 (43.0%)	256 (24.7%)	0 (0%)
$Sorter2$	49,083 (16.2%)	544 (52.8%)	0 (0%)
$Total$	275,739 (90.8%)	986 (95.7%)	28 (1.7%)

Table 5.1: Chip and Memory utilization of Radix Sorters

sorter has a lot less number of forwarding, buffer and storage queues resulting in the lesser LUT units utilization.

However, in contrast, the BRAM usage of first sorter is just 24.7% whereas the second sorter uses 52% of it. This happens because of the sizes of the bursts which are processed. Since the first sorters processes bursts of 1KB each, the amount of time (and the size of data) for which addresses are stored in the FIFO queues is very less and they are cleared very often. However, since the second sorter processes bursts of 8KB, data has to be stored for a longer time before it sufficient quantity is accumulated for bursting. As a result, the second sorter ends up using more BRAM than the first.

5.1.2 Throughput of the Radix Sorters

- Ideal case throughput: Considering that 128-bits are processed every cycle, the ideal throughput of the radix sorters would be the bit processed per cycle multiplied by the clock frequency which equals **4 GiB/s**. However, this theoretical maximum can't be achieved in reality due to local skews in the data.

- Ideal case 32-bit throughput: If instead only 32-bits are processed every cycle, then the theoretical maximum throughput is **1 GiB/s**.

- Actual performance: The actual throughput of the Radix sorters turned out to be approximately **2.91 GiB/s**. The reason the theoretical maximum cannot be reached is because the data often contains local skews with all the data to be directed to a single radix sub-sorter. Since elements in the buffers of a sub-sorter are processed sequentially, this effectively reduces the throughput. However, a throughput of 2.91 GiB/s is still very impressive, and as discussed next, the radix sorters are not the bottleneck in the overall pipeline and thus we are satisfied with the performance.

5.2 Some Overall Results

In this section, we present some overall performance values (including the rest of the components in the architecture, such as the Hash Functions and the Table Updaters) that were demonstrated that were achieved.

5.2.1 Overall Throughput

- Ideal case throughput: The main bottleneck in the overall chip is the memory, which has a peak bandwidth of **10 GiB/s**. Considering all the components, this bandwidth will be shared between them. We estimated that there would be 3 read-write pairs happening simultaneously (1 pair each from each of the table updaters, one read action from the second radix sorter, and one write action from the first radix sorter). Hence, the ideal peak bandwidth that would be available to each component is 10/6 = **1.66 GiB/s**.

- Actual performance: The actual end-to-end pipeline performance achieved was **1.49 GiB/s**. Although there is a slight reduction in performance due to the inefficiencies in the components, the performance is still impressive.

5.2.2 Power Consumption & Efficiency

The following plot (Fig 5.2) shows the throughput (in terms of Mhash/s) of the BunchBloomer and how it compares other implementations. It compares the Software implementations using the libraries Guava [6], bloom (Bl) [1], libbloom (Lbl) [13, 9], and custom single-thread (S1) and multi-thread (S4) implementations with Direct update (DU) and BunchBloomer FPGA implementations.

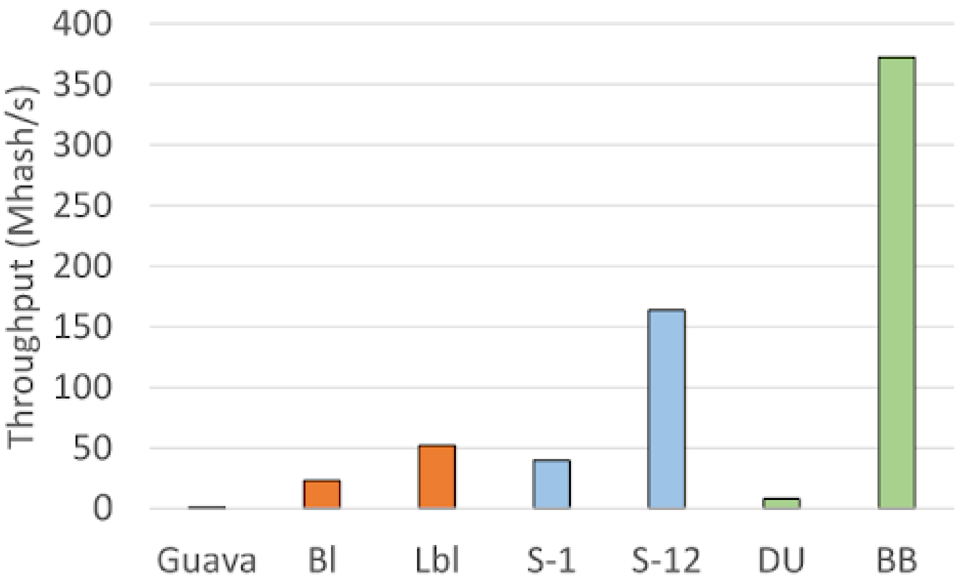

Figure 5.2: Throughput comparison (in MHash/s) of BunchBloomer with other implementations

The most notable comparisons can be made to the Software implementation on a multi-core server, and an FPGA implementation with Hybrid Memory Cubes. The BunchBloomer achieves an impressive throughput of **372 MHash/s** at a very low power consumption of **25W**.

In comparison, a software 12-threaded implementation on a multi-core server achieved a throughput **163 MHash/s**. Although the throughput difference is just 2x and the software implementation can be improved further by optimizing the code more, what makes the BunchBloomer standout is the power consumption. The multi-core server consumed a power of **120W** compared to 25W and therefore, BunchBloomer has 10x better Power Efficiency (Throughput/Power).

Likewise, [19] uses an FPGA Hybrid Memory Cube based implementation that produces a **220 MHash/s** throughput. Although the power consumption was nit explicitly stated, we estimate that the FPGA would consume similar power as our implementation **(25W)** and an **extra 43W** is used by the Hybrid Memory Cube Fabric. Therefore, in comparison, BunchBloomer still has a 4x better Power Efficiency.

Chapter 6

Conclusions

We conclude this report by summarizing some of the important details in the project.

- Bloom filters are an important tool that are used in several important genomics applications such as k-mer counting, genome sequencing, de Brujin graph construction, etc. However, considering their magnitude, Software implementations can easily become performance and memory bound.
- Using Accelerators such as GPUs or FPGAs can significantly speed the Hashing process thereby removing the performance overhead. However, a high Performance memory, either the on-chip memory or high performance power hungry off-chip memory such as HMCs, is required to not undermine the processing capacity of the accelerator.
- The two-level Radix Sorting approach improves the bandwidth of the existing DRAM by decreasing the randomness in the memory accesses. The addresses are bucketed and processed in bursts of 8KiB to match the line buffer size of the DDR3 DRAM of the target chip, thereby ensuring temporal locality in the accesses and efficient usage of the cache.

- The overall BunchBloomer has 10x better Power Efficiency compared to a 12-threaded Software implementation on a multi-core server and over 4x better Power Efficiency compared to an FGPA Hybrid Memory Cube implementation.

Bibliography

[1] ArashPartow. C++ bloom filter library. `https://github.com/ArashPartow/bloom`.

[2] D. L. Cameron, J. Schröder, J. S. Penington, H. Do, R. Molania, A. Dobrovic, T. P. Speed, and A. T. Papenfuss. Gridss: sensitive and specific genomic rearrangement detection using positional de bruijn graph assembly. *Genome research*, 27(12):2050–2060, 2017.

[3] R. Chikhi and G. Rizk. Space-efficient and exact de bruijn graph representation based on a bloom filter. *Algorithms for Molecular Biology*, 8(1):1–9, 2013.

[4] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. In *11th Symposium on High Performance Interconnects, 2003. Proceedings.*, pages 44–51. IEEE, 2003.

[5] M. Erbert, S. Rechner, and M. Müller-Hannemann. Gerbil: a fast and memory-efficient k-mer counter with gpu-support. *Algorithms for Molecular Biology*, 12(1):1–12, 2017.

[6] google. Guava: Google core libraries for java. `https://github.com/google/guava`.

[7] J. Harwayne-Gidansky, D. Stefan, and I. Dalal. Fpga-based soc for real-time network intrusion detection using counting bloom filters. In *IEEE Southeastcon 2009*, pages 452–458. IEEE, 2009.

[8] W. Huangfu, K. T. Malladi, S. Li, P. Gu, and Y. Xie. Nest: Dimm based near-data-processing accelerator for k-mer counting. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[9] jvirkki. A simple and small bloom filter implementation in plain c. `https://github.com/jvirkki/libbloom`.

[10] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better bloom filter. In *European Symposium on Algorithms*, pages 456–467. Springer, 2006.

[11] M. J. Lyons and D. Brooks. The design of a bloom filter hardware accelerator for ultra low power systems. In *Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design*, pages 371–376, 2009.

[12] N. Mcvicar, C.-C. Lin, and S. Hauck. K-mer counting using bloom filters with an fpga-attached hmc. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 203–210. IEEE, 2017.

[13] P. Melsted and J. K. Pritchard. Efficient counting of k-mers in dna sequences using a bloom filter. *BMC bioinformatics*, 12(1):1–7, 2011.

[14] J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J. M. Tiedje, and C. T. Brown. Scaling metagenome sequence assembly with probabilistic de bruijn graphs. *Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012.

[15] D. Pellow, D. Filippova, and C. Kingsford. Improving bloom filter performance on sequence data using k-mer bloom filters. *Journal of Computational Biology*, 24(6):547–557, 2017.

[16] K. Salikhov, G. Sacomoto, and G. Kucherov. Using cascading bloom filters to improve the memory usage for de brujin graphs. *Algorithms for Molecular Biology*, 9(1):1–10, 2014.

[17] Y. Wang, T. Pan, Z. Mi, H. Dai, X. Guo, T. Zhang, B. Liu, and Q. Dong. Namefilter: Achieving fast name lookup with low memory cost via applying two-stage bloom filters. In *2013 Proceedings IEEE INFOCOM*, pages 95–99. IEEE, 2013.

[18] W. Xiao, L. Wu, G. Yavas, V. Simonyan, B. Ning, and H. Hong. Challenges, solutions, and quality metrics of personal genome assembly in advancing precision medicine. *Pharmaceutics*, 8(2):15, 2016.

[19] J. Zhang and J. Li. Degree-aware hybrid graph traversal on fpga-hmc platform. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 229–238, 2018.

[20] Q. Zhang, J. Pell, R. Canino-Koning, A. C. Howe, and C. T. Brown. These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. *PloS one*, 9(7):e101271, 2014.