EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Large-Block Multi-rate Streaming Sort

van Valenberg, Damy F.B.

Award date:
2022

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Department of Mathematics and Computer Science
Architecture of Information Systems Research Group

Large-Block Multi-rate Streaming Sort

Master Thesis

Damy van Valenberg
(2087728)

Supervisors:
Prof.dr.ir. C.H. van Berkel
Dr. R.H. Mak
Dr.ir. R. Jordans

Version 1.0

Vught, July 2022

Preface

Before you lies the thesis ”Large-Block Multi-rate Streaming Sort”. The purpose of this research is to improve upon or find a new solution for streaming sorting. The thesis has been written to fulfil the graduation requirements for the Master’s degree in Embedded Systems at the Eindhoven University of Technology. I started the research in July 2021 and should be finished in July 2022.

This project was chosen in consultation with Kees van Berkel. The research was conducted solely for the Eindhoven University of Technology. For me, the research was a lot more difficult than initially expected. Fortunately, I was still able to provide an answer to the research question with a design and partial implementation in StaccatoLab. Also, a novel design for parallel sorting with higher-throughput was found.

I would like to thank my supervisor Kees van Berkel for the excellent guidance, pushing me out of my comfort zone and for keeping me critical at my own work. I would also like to thank my family and friends for keeping me motivated and all advice.

I hope you enjoy reading my thesis.

Damy van Valenberg

Vught, June 2022

Abstract

Sorting is an important problem in computer science. It is a very fundamental concept used in many other algorithms, but is also critical for the performance of databases, for example. If large data sets or high-performance sorting is required, sorting using only the CPU is often infeasible, and often only hardware sorting can satisfy the requirements. The simplest solution is sorting on the GPU, but an FPGA or ASIC can also be used. In this research, we find a solution for the following problem:

How can a streaming sort
- of large blocks (of variable size) of key-value pairs be achieved
- with a fixed rate greater or equal to two
- while minimizing memory resources and traffic
- given it will be modelled in StaccatoLab
- with possibly an FPGA with external memory implementation?

The research is split up in three phases. In the first phase, some optimizations are made for the baseline sorter, such as minimizing the memory usage and solving the variable block sizes. During the second phase, the single rate sorter is parallelized, such that it becomes a multi-rate sorter. This increases the throughput of the sorter. The parallelization uses a novel method. The design is pipeline-friendly, which allows very high clock rates. The example provided in this research uses a parallelization of $P = 4$, and a clock frequency of 200 MHz, with a key-value pair size of 128 bits, this results in a throughput of 12.8 GB/s. This throughput is enough to fully saturate the bandwidth of the DDR3-1600 DRAM memory onboard the Xilinx VC707. But higher parallelization and clock frequencies are also possible as long as the input and output mediums can handle the throughput.

In the final phase of the research, the focus lies on sorting large blocks/data sets. The data is so large, that it must be stored in external memory or even external storage. It is also found that sorting is limited by the bandwidth of the external memory, which means that the reads and writes to the external memory must be kept as low as possible. Moreover, it is important to guarantee 100% bandwidth utilization of the external memory/storage, such that there is no loss of throughput. The result is a very efficient sorter, which is compared to another existing work, and reduced the sorting time of sorting 0.5 GB from 2390 seconds to 1650 seconds on the same hardware. This is a reduction of approximately 31% of sorting time.

Contents

Contents iv

List of Figures vi

List of Tables vii

1 Introduction 1

2 Background information 2
2.1 Dataflow . . . 2
2.2 Sorting definition . . . 3
2.2.1 Batch sorting . . . 4
2.2.2 Streaming sort . . . 4
2.2.3 Dataflow rate . . . 4
2.2.4 Throughput . . . 4
2.2.5 Latency . . . 5
2.2.6 Sorting algorithms . . . 5
2.3 Merge sort . . . 7

3 Existing research 9

4 Problem description 10

5 Single pass sorting 12
5.1 Baseline . . . 12
5.2 Caching . . . 13
5.3 Memory usage . . . 13
5.3.1 Removing slack . . . 13
5.3.2 Adding indexing . . . 14
5.4 Variable block sizes . . . 16
5.5 Implementation . . . 17

6 Multi-rate sorting 19
6.1 Sorting networks . . . 19
6.2 Parallel sorting . . . 21
6.3 Design . . . 21
6.3.1 Design 1 (Feedback loop) . . . 21
6.3.2 Design 2 (Min select) . . . 22
6.4 Implementation . . . 23
6.4.1 Comparison . . . 25
6.4.2 Results . . . 25

7 Multi pass sorting **26**
7.1 K-way merge 28
7.2 Design 28
7.3 Buffer sizing 29
7.4 Implementation 31
7.4.1 StaccatoLab 31
7.4.2 Larger implementations 32
7.4.3 Results 33

8 Improvements **35**

9 Conclusions **36**

Bibliography **37**

Appendix **39**

A Flag derivation **39**

B Single pass sorting code **42**

C Multi-rate sorting code **48**

D Multi pass sorting code **55**

List of Figures

2.1 Dataflow graph . . . 2
2.2 FIR filter as a dataflow model . . . 3
2.3 Radix sort double buffering implementation . . . 6
2.4 Pseudocode for the merge function of the merge sort . . . 7
2.5 Merge sort, divide and conquer strategy . . . 8
2.6 Pseudocode for the complete merge sort . . . 8

4.1 System overview . . . 10

5.1 Merge sort implementation for $N = 8$ . . . 13
5.2 Memory usage trace for the merge stage $n = 8$ . . . 15
5.3 Memory usage comparison for a single stage with key size, k=128 . . . 16
5.4 Split (spl) node sending all blocks of size 1 to edge A (flow only view) . . . 16
5.5 Split (spl) node code . . . 17

6.1 Bitonic merge sort . . . 20
6.2 Batcher Odd-Even Mergesort . . . 21
6.3 Terabyte sort merger . . . 22
6.4 Minimum select design for $P = 4$ . . . 23
6.5 Logarithmic plot of the comparator count for designs 1 and 2. . . . 23
6.6 Multi-rate first two stages . . . 24
6.7 Multi-rate last stages . . . 24
6.8 Memory usage multi-rate sorter . . . 25

7.1 Difference in arithmetic intensity for several levels of parallelism (P) . . . 27
7.2 Roofline for a block size of 128 MB. The operational intensity is half the arithmetic intensity for two passes. . . . 27
7.3 4-way merge tree variants . . . 28
7.4 Multi pass design abstracted overview (for $K = 4$) . . . 29

List of Tables

2.1 List of comparison sort algorithms 6

5.1 Merge stage input/output data 17

6.1 Table of the comparator count for designs 1 and 2. 23

7.1 Arithmetic intensity with a block size of $N = 1024$ pairs (16 KB) 26
7.2 Optimal solutions for two passes. N is in number of pairs (128-bit per pair) . . . 32
7.3 Optimal solutions for three passes. $N * K^2 = 1$ GB for all solutions. 33
7.4 Required N, K and passes to sort 0.5 GB for P 1 up to 32. 33
7.5 512 GB sorting performance from the Terabyte sort paper 34
7.6 512 GB expected sorting performance with the proposed design 34

Chapter 1

Introduction

Sorting is a very well-known problem in computer science. It is often at the core of almost every application that processes data. For example, databases depend heavily on sorting [17], selecting and sorting millions of records. Efficiency of the sorting process is critical for these systems. Some database systems even use hardware acceleration using FPGA's to improve sorting performance[4]. Besides direct applications of sorting, there are also numerous algorithms that depend on sorting. For example, searching, checking for uniqueness, min/max selection and many more. Any minor improvement in sorting performance could potentially have a significant impact on many future systems.

The research focuses on finding a solution for a streaming sort implementation in hardware, specifically an FPGA. The research is split up in three parts. In the first part, a single-rate streaming sorter is designed and implemented. A single-rate streaming sorter takes in a stream of unsorted data and produces a stream of sorted data. The streams have a rate of one, so one value enters, and one value exits each clock cycle (hence, the single-rate). In the second part, the single-rate solution is parallelized, such that the sorter can process more than one value in one clock cycle. As a result, the input, and output streams also transfer multiple values each cycle. This sorter is called a multi-rate streaming sort. In the last part of this research, a multi pass sorter is designed. In which case, data is temporarily stored in external memory. This is needed when the unsorted data set is so large that it no longer fits in the internal memory of the FPGA.

There is a lot of existing work on sorting in hardware. But most research focuses on batch sorting, not streaming sorting. There is one very detailed paper, 'Terabyte Sort on FPGA-Accelerated Flash Storage'[9], by Sang-Woo Jun, Shuotao Xu and Arvind. The paper was published in 2017 and contains a lot of design, implementation, and performance details. The performance seems good and is realistic. This paper was mainly used as a benchmark to compare to this research.

Chapter 2

Background information

2.1 Dataflow

For this research, dataflow is used to solve the sorting problem. There are many variants of dataflow. Most relevant are static dataflow, cyclo-static and dynamic dataflow. Every dataflow graph contains nodes, edges and tokens. A basic (flow only) dataflow graph is shown in figure 2.1. It contains three nodes (A, B and C) and the nodes are connected by (directional) edges. The edge between node A and B contains a token, indicated by the black dot on the edge. A token can contain data, which can be a single value or a complex structure (such as tuples). An edge can also store more than one token. The number of tokens it can store is also called slack. The tokens on the edges always have FIFO behaviour. In static dataflow, if a token is present on every input edge of a node, the node will fire, consuming the input tokens and producing a token on every output edge using its output function(s). In cyclo-static dataflow, the firing is controlled by a finite state machine (FSM). The state machine uses firing rules that indicate which input tokens will be consumed and which output tokens will be produced. In dynamic dataflow, the FSM of a node is dependent on the data in a token.

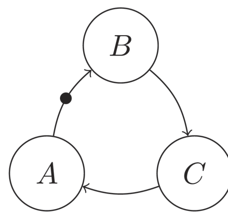


Figure 2.1: Dataflow graph

StaccatoLab[1] is a tool developed by Kees van Berkel. This tool is used to model and simulate dataflow graphs. There are many problems that can modelled using a dataflow graph. One simple example is digital audio filters. In figure 2.2 a FIR (finite impulse response) filter is shown that was modelled in StaccatoLab. Models in StaccatoLab can be thoroughly analysed, since the input and output can be fully simulated. StaccatoLab helps to calculate the throughput and latency of the model (and many other statistics). This also makes it simple to calculate the required clock frequency if the model is to be implemented in hardware.

Besides simple filters, there are also many other algorithms that can be modelled. For example, a data stream can be inverted, selecting minimum or maximum values from a stream, or even sorting, which is the main focus of this research project. The dataflow graphs can also be converted to hardware implementations. It should be possible to do this conversion automatically, which is a feature that is still in development in StaccatoLab.

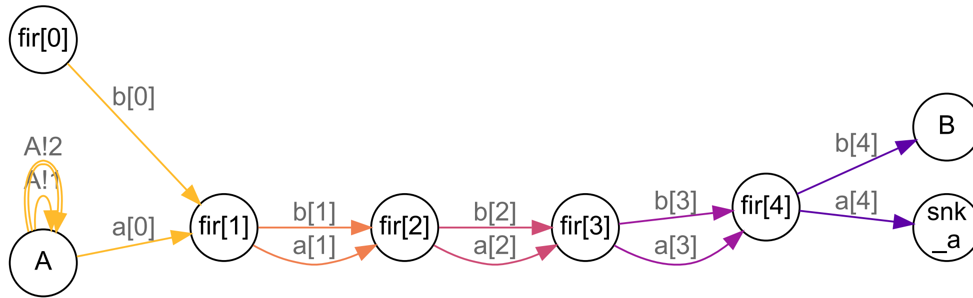


Figure 2.2: FIR filter as a dataflow model

Traditionally, most implementations of algorithms are designed with memory-based computing in mind. In which data is read/written from/to a shared memory to achieve the desired result. The memory bandwidth is often limiting the throughput. Caching or shared memory modules used in GPU's can increase performance significantly. In dataflow, the design and implementation is based on the data and how this moves through the graph. The edges in dataflow graphs are often just registers or a small block of ram (BRAM) when it is transformed to an FPGA implementation. The registers and BRAM can be read/written independently, which allows many reads and writes each cycle. Sorting is also often limited by the memory bandwidth. This is why sorting is an interesting subject to be modelled with dataflow. Moreover, dataflow simplifies designing for a known throughput. For example, it is often easy to ensure that a graph will fire every cycle (although not always possible). In which case, the throughput is mainly dependent on the clock frequency. This also makes for a very efficient design, since the used hardware will often have a high utilization.

All the nodes of a dataflow graph can fire independently. This makes parallelism easy to implement. Without going into too much detail, the FIR filter implementation from figure 2.2 uses pipelining behaviour. The FIR filter must do four multiplications with a coefficient and the input data and sum the results for the correct output. These multiplications and additions are done in each $fir[1]..fir[4]$ stage. By using pipeline parallelism, it is possible to stream this FIR filter, so each cycle one output is produced. An example of another form of parallelism, that is not pipeline behaviour, would be to duplicate this FIR filter implementation one or more times. This allows processing of multiple independent streams, essentially raising the throughput. This can be useful when filtering audio with both a left and right channel, for example, since these streams are independent.

2.2 Sorting definition

As said before, sorting plays an important part in computer science. But what is sorting, actually? Let's first define a sorted list. Assume there is a list A, and a_i is the element in list A at index i. Then the list A is sorted when it satisfies the following condition $\forall\,[0 \leq i < |A| - 1 : a_i \geq a_{i+1}]$. More specifically, this defines the descending order. The ascending order is defined as follows, $\forall\,[0 \leq i < |A| - 1 : a_i \leq a_{i+1}]$. Note that a list can contain any type of data (numbers, strings, etc.) as long as the relation '$\leq$' and '$\geq$' can be defined for all the elements in the list. A list is considered unsorted when it satisfies neither the descending nor ascending condition. Sorting is the process that converts an unsorted list (or sorted if the state of the list is unknown) into a sorted list, without removing or adding values. Sorting can be done using a sorting algorithm.

Sorting can be done on the CPU, but is also regularly done on the GPU or even specialized hardware, such as an FPGA. The idea of hardware sorting is to improve the performance compared to sorting on the CPU, since an FPGA can achieve a very level of parallelism. This also frees the CPU to do other tasks. In most cases, a block of data is provided to the sorting hardware via

some kind of shared memory. When the hardware has finished sorting, the data should be sorted in the same or another block of shared memory.

There are different implementations of sorting algorithms in hardware. Two classifications are important. Firstly, an implementation can be either a batch sorter or a streaming sorter. Secondly, an implementation can be either sequential or parallel. These two classifications are independent of each other. What they exactly mean will be described in the next sections.

2.2.1 Batch sorting

One of the most common ways to implement a hardware sorter is to connect the hardware to the CPU using shared memory. The shared memory is the bridge between the CPU and sorting hardware. The CPU signals the sorting hardware when it has finished copying the unsorted data to the shared memory. The sorting hardware will then start sorting using its algorithm. The hardware will signal the CPU once it has finished sorting. The CPU can then read the sorted output from shared memory.

This is an easy way to implement a sorter. However, there are some issues with this approach. Often the processing time is not constant (this depends on the sorting algorithm), which makes performance analysis more difficult. Another issue is that the next block of data cannot be copied to the shared memory until the hardware has finished sorting. This issue could be resolved by using double buffering. But, as a consequence, this would also mean you need twice the amount of memory.

2.2.2 Streaming sort

In the case of a streaming sort, a constant stream of data is provided to the hardware. The samples from the stream arrive at a fixed rate. For example, every cycle, one sample is provided to the sorter. Interestingly, the hardware sorter can already start sorting when the first sample arrives. And actually the sorter should start the sorting immediately because otherwise the input values must be buffered, which would increase the memory required. A good streaming sorter implementation will often have a low latency and a fixed throughput. However, since the sorter needs to start sorting when not all data is available yet, only a few sorting algorithms can be used. Most sorting algorithms require all data before they can start the sorting process.

Such an input stream could be from a sensor, which periodically measures something, or an audio source, for example. Or if the FPGA is integrated in a PC, this could be a PCIe data stream. This makes streaming sorters very useful for real-time applications. The main advantage of a streaming sorter is that the values do not have to be stored in external memory, which reduces hardware resources and power consumption.

2.2.3 Dataflow rate

In dataflow, a common characteristic of any design is the dataflow rate. It indicates how many tokens are consumed (and produced) every clock cycle. A graph with rate 1 consumes 1 token each cycle. A graph with a rate of 0.5 consumes 1 token every 2 cycles. However, a graph can also have a rate larger than 1. A graph with rate 2 consumes 2 tokens each cycle.

Note that a multi-rate (rate > 1) streaming sort has a higher throughput than a streaming sort with a rate of 1 assuming the clock frequency is the same, which will be explained now.

2.2.4 Throughput

Throughput is one of the metrics which can be used to measure the performance of a system. Throughput can be measured (or calculated) at the input or output of a system. In the case of sorting, the input and output throughput should be equal. This is because the number of tokens in the system should remain constant, as there is a finite amount of memory to hold the tokens.

For a streaming sorter, the throughput is directly related to the dataflow rate. The data consumed each cycle is determined by the token size and dataflow rate, which can be multiplied to get the total input size to the system. If the input size is multiplied again by the frequency of the system, the result shall be the throughput, as shown below.

$$Throughput = Clock\ frequency \times rate \times token\ size$$

Throughput can be expressed in different units. The most common units are bits per second (also known as bitrate) or bytes per second, and are often combined with metric prefixes. For example, 1,000 bit/s = 1 kbit/s or another example, 1,000,000 bytes/s = 1 MB/s.

2.2.5 Latency

Latency is defined as the time delay between the input and the output of a system. It can be measured in different ways. Consider the following case, a large block of data is consumed element by element on the input and later produced element by element on the output. Then the latency can be measured from the first input to the first output, or from the first input to the last output, etc. So, for latency, it is always important to explicitly indicate or check what is meant.

2.2.6 Sorting algorithms

There are many sorting algorithms. One of the main characteristics used to compare these algorithms is the performance. For this, the big O notation is used. This indicates how the performance will be affected if the number of items to be sorted increases. For example, the insertion sort has a worst-case performance of $\mathcal{O}(n^2)$. So, as the number of items to be sorted increases linearly, the sorting time increases quadratic.

Sorting algorithms are also distinguished by being a comparison sort or not. Comparison sorts, sort a list of data only by comparing elements. Comparison sorts have been proven to be limited to $\mathcal{O}(n \log n)$ as the average case performance [8]. Non-comparison sorts are not bound to this limit. However, they often have other constraints, such as limited key size or range. The most well-known comparison sorting algorithms with their performance is shown in table 2.1.

The most important non-comparison sorts are the counting and radix sort. They are used in many sorters, since they perform extremely well. Counting sort has an average case performance of $\mathcal{O}(n + r)$, where r is the range of the keys. Radix sort has an average case performance of $\mathcal{O}(n \cdot \frac{k}{d})$, where k is the size of the key and d is the digit size. [19]

The radix sort is often implemented in software running on the CPU or the GPU. Its performance can be better than a simple merge sort. The "Designing Efficient Sorting Algorithms for Manycore GPUs" [18] paper shows an implementation which is faster than a merge sort on the GPU. Although the radix sort is not limited to a software implementation (as shown in the paper "FastRadix: A Scalable Hardware Accelerator for Parallel Radix Sort" [12]), it becomes unpractical to implement standalone on an FPGA as a streaming sort. Radix sort requires all data of the block before it can start the sorting process. This has several unwanted consequences for a streaming sort implementation. Most importantly, the hardware implementation would not be able to accept new incoming values (which will arrive every cycle, since a streaming sort is required) until the sorting is complete. This means the incoming values need to be stored/buffered, before they can enter the sorter. This would most likely be done using double buffering, which will significantly increase the memory usage. An example of what such an implementation would look like is shown in figure 2.3. Sorting algorithms that behave like batch sorters often require this double buffering solution to turn them into streaming sorters. Another result is that all data needs to be stored in the same BRAM. Every BRAM block in an FPGA often has two read/write ports. As a result, a very high memory throughput could be reached if these BRAMs are used independently in parallel. But if all the data needs to be stored in 'joined' BRAM blocks, this limits the read/write operations to two per cycle.

Name	**Best** ($\mathcal{O}$)	**Average** ($\mathcal{O}$)	**Worst** ($\mathcal{O}$)	**Method**
Quicksort	$n \log n$	$n \log n$	n^2	Partitioning
Merge sort	$n \log n$	$n \log n$	$n \log n$	Merging
Introsort	$n \log n$	$n \log n$	$n \log n$	Partitioning & Selection
Heapsort	$n \log n$	$n \log n$	$n \log n$	Selection
Insertion sort	n	n^2	n^2	Insertion
Block sort	n	$n \log n$	$n \log n$	Insertion & Merging
Timsort	n	$n \log n$	$n \log n$	Insertion & Merging
Selection sort	n^2	n^2	n^2	Selection
Cubesort	n	$n \log n$	$n \log n$	Insertion
Shellsort	$n \log n$	$n^{4/3}$	$n^{3/2}$	Insertion
Bubble sort	n	n^2	n^2	Exchanging
Exchange sort	n^2	n^2	n^2	Exchanging
Tree sort	$n \log n$	$n \log n$	$n \log n$	Insertion
Cycle sort	n^2	n^2	n^2	Selection
Library sort	$n \log n$	$n \log n$	n^2	Insertion
Patience sorting	n	$n \log n$	$n \log n$	Insertion & Selection
Smoothsort	n	$n \log n$	$n \log n$	Selection
Strand sort	n	n^2	n^2	Selection
Tournament sort	$n \log n$	$n \log n$	$n \log n$	Selection
Cocktail shaker sort	n	n^2	n^2	Exchanging
Comb sort	$n \log n$	n^2	n^2	Exchanging
Gnome sort	n	n^2	n^2	Exchanging
Odd–even sort	n	n^2	n^2	Exchanging

Table 2.1: List of comparison sort algorithms with their corresponding complexity in big-$\mathcal{O}$ notation. [19]

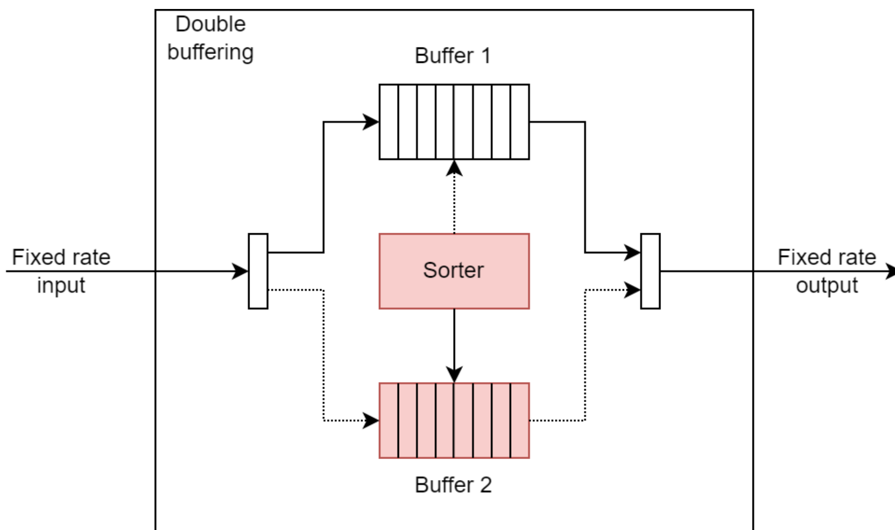


Figure 2.3: Radix sort double buffering implementation

2.3 Merge sort

The merge sort algorithm lends itself extremely well to hardware implementations, which will be shown later (in chapter 5.1). The merge sort is a simple sorting algorithm, and has a complexity of $\mathcal{O}(n \log n)$, which means it scales optimal for a comparison sorting algorithm.

The core of the merge sort is the merge function. It takes in two sorted lists, which it merges to produce one sorted output list. The merging is done by only looking at the front of each list, since they contain the lowest elements. The lowest of the two elements is removed from the corresponding list and is appended to the back of the output list. This process is repeated until both lists are empty. Pseudocode for the merge algorithm is shown in figure 2.4.

```
function merge(left, right) is
    var result := empty list

    while left is not empty and right is not empty do
        if first(left) <= first(right) then
            append first(left) to result
            left := rest(left)
        else
            append first(right) to result
            right := rest(right)

    // Either left or right may have elements left; consume them.
    // (Only one of the following loops will actually be entered.)
    while left is not empty do
        append first(left) to result
        left := rest(left)
    while right is not empty do
        append first(right) to result
        right := rest(right)
    return result
```

Figure 2.4: Pseudocode for the merge function of the merge sort[20]

The merge sort algorithm uses the divide and conquer strategy to sort an input list with random data. This is well illustrated in figure 2.5. The unsorted list on top in red is broken up into smaller parts, until only one element remains in all lists. The merge function is then used to build increasingly larger lists (in green), until the final output list is produced. The pseudocode for the complete merge sort algorithm can be seen in figure 2.6.

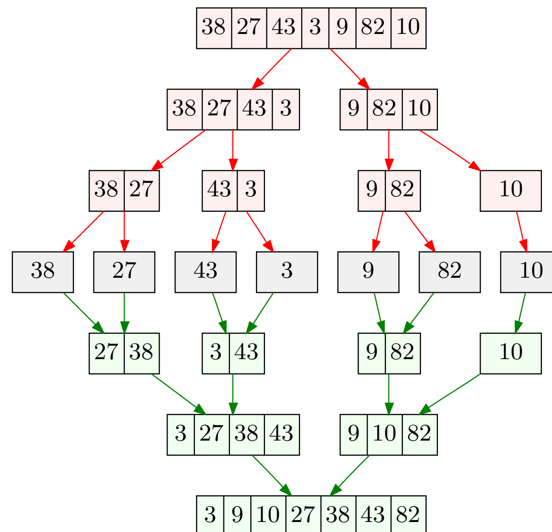


Figure 2.5: Merge sort, divide and conquer strategy[7]

```
function merge_sort(list m) is
    // Base case. A list of zero or one elements is sorted, by definition.
    if length of m <= 1 then
        return m

    // Recursive case. First, divide the list into equal-sized sublists
    // consisting of the first half and second half of the list.
    // This assumes lists start at index 0.
    var left := empty list
    var right := empty list
    for each x with index i in m do
        if i < (length of m)/2 then
            add x to left
        else
            add x to right

    // Recursively sort both sublists.
    left := merge_sort(left)
    right := merge_sort(right)

    // Then merge the now-sorted sublists.
    return merge(left, right)
```

Figure 2.6: Pseudocode for the complete merge sort[20]

Chapter 3

Existing research

Finding applicable, high-quality papers of existing work also makes up a large part of the research. Many papers exist about dataflow, but none that combine dataflow with sorting. But, papers about sorting algorithms are plentiful. Most FPGA implementations use some kind of merge sorter or a combination with another sorter, such as sorting networks. Sorting networks also seems to be an important part of many implementations. There are also many papers that use the parallelism of the GPU with different algorithms. Most of the GPU papers seem irrelevant, since there is a too much of a difference between a GPU and an FPGA, but these could be used to compare performance. Some papers are discussed in the next paragraphs.

The paper "Terabyte Sort on FPGA-Accelerated Flash Storage" by Sang-Woo Jun, Shoutao Xu and Arvind[9] was by far the most important for this research. This paper gives an in-depth design and implementation of a parallel hardware sorter. The smallest part or core component of their sorter is actually a streaming sort. Not only that, it is also a multi-rate sorter, which is helpful for the second phase of the research. This component is reused in other components that are required to create a larger sorting network. The streaming sorter uses merge sorting as a sorting algorithm. The streaming sorter has a throughput of 4 GB/s at 125 MHz. The paper also gives a solution for sorting large data-sets (up to 1 TB, hence the name of the paper). They utilize a 1 TB PCIe SSD to load and store the data. All things combined make the paper very valuable to this research.

Another paper called "Hardware Acceleration of Sorting Algorithms Using Reconfiguration Technics" by Pawel Russek and Kazimierz Wiatr ([17]) contains in depth information about the implementation of sorting nets for FPGA's. This paper connects closely with the previous paper by Sang-Woo Jun. It includes half cleaners, but also full sorting nets, which will be useful again for implementing a multi-rate design. Moreover, it also includes some practical limits for the sorting nets.

NVIDIA published a paper in 2009 called "Designing Efficient Sorting Algorithms for Manycore GPUs" written by Nadathur Satish, Mark Harris and Michael Garland ([18]). The paper claimed to have the fastest sorter at that time. It combines radix sort, counting sort and merge sort into one sorting algorithm. The sorting is done in parallel on the streaming processors of a GPU. The NVIDIA GPU's contain many layers of memory[13]. It has external memory, L2 cache and L1 cache. This is not available in an FPGA, since that only has BRAM. The bandwidth of the external memory (GDDR5X) of a GTX 1080 is around 320GB/s, which is much higher than that compared of the DDR3 dram of 12.8GB/s used in this research. The architecture differs too much to adapt the implementation for hardware/dataflow. But, it shows how fast the radix sort can be.

Finally, in the paper "FastRadix: A Scalable Hardware Accelerator for Parallel Radix Sort" by Xingyu Liu and Yangdong Deng ([12]) they also implemented a radix sort in hardware. They propose the "FastRadix" hardware accelerator, which differs slightly from the standard radix sort. The sorter was implemented on an FPGA and works together with the CPU. The paper includes some implementation details about their radix sort, which can be very valuable if, at a certain point, a radix sorter will be designed in StaccatoLab for the research project.

Chapter 4

Problem description

As said in the introduction, sorting is a basic but also crucial problem. In software, the problem has been very well researched, and most recent improvements come from using the GPU. But when sorting in hardware, there are many ways to implement a single algorithm as there is a lot more freedom in the architecture. This makes hardware sorting so interesting. The most common hardware implementations use an FPGA or ASIC. In this research, the main focus lies on FPGA's, but is also applicable to an ASIC. A hardware sorter is often used as a form of hardware acceleration as part of a server, but could also be used standalone. Both the FPGA and ASIC are capable of achieving a very high level of parallelism, which can increase the performance of such algorithms drastically. As a consequence of using hardware acceleration, the CPU will also be freed to do other work while the hardware is sorting.

Existing research focuses mainly on batch sorting. There are only few existing streaming sort solutions. This research project will take a step even further. The focus will be on creating an efficient streaming sort implementation in StaccatoLab with a data-rate larger than 1 for any data block size. This research will explore some new area's and will hopefully be able to improve the existing implementations. A minor improvement to existing sorting solutions could already have a significant impact because sorting is so essential to many other (aforementioned) applications.

For the final design, the target will be to sort large data sets that will not fit inside the memory of an FPGA. As a result, the FPGA must fall back to external memory, such as DRAM or even other (external) storage, such as an SSD or HDD. The data sets could even be so large, that the FPGA would require multiple passes over the data, which means the same data must be read and written multiple times from/to external memory.

A very basic diagram of the system is shown in figure 4.1. This is not a design to solve the problem, but is shows the environment in which the problem exists. The hardware which will be used for the research is the same as the hardware used in the Terabyte sort paper. Namely, the Xilinx VC707 Evaluation Kit. It contains a Virtex-7 (XC7VX485T-2FFG1761C) as the FPGA and the board has 1 GB of DDR3-1600 memory.

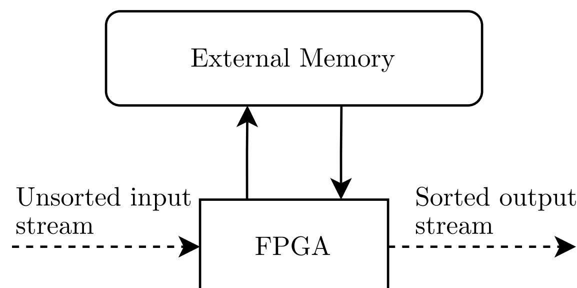


Figure 4.1: System overview

Sorting is almost always done using a key-value pair. The keys are what is looked at for the sorting. The value is not used for the sorting itself, but it contains the reference to the actual data. Using a key-value pair avoids having to move a large amount of unused data around. For this research, both the key and value are assumed to be 64-bit, so one key-value pair is 128-bits in total.

The research question is formulated as follows:

How can a streaming sort
- of large blocks (of variable size) of key-value pairs be achieved
- with a fixed rate greater or equal to two
- while minimizing memory resources and traffic
- given it will be modelled in StaccatoLab
- with possibly an FPGA with external memory implementation?

The research question is divided up into three parts. In the first part, a baseline solution is analysed and improved to sort small (but variable) block sizes at rate 1, using minimal internal memory. The second part will use the knowledge of the first part to parallelize the solution and make it multi-rate. In the final part, multi pass sorting is researched to be able to sort the large block sizes.

There could be many solutions for each sub problem, which makes it extra important to properly compare the solutions. Each solution will be measured/analysed by the following statistics:

- Throughput (bytes per second) or sorting time (total seconds)
- Comparators used
- BRAM (internal memory) used
- DRAM (external memory) used

Chapter 5

Single pass sorting

5.1 Baseline

The merge sort algorithm has multiple phases. Each phase merges two sorted lists from the previous phase to create a sorted list that has the length of both input lists combined. So, each phase produces a sorted list twice as large as the previous phase. If the phase is defined as $1 \leq p$, then $n = 2^p$ is the size of the sorted output list. For an unsorted list of size N (for now N is assumed to be a power of two), there are $log_2(N)$ phases. In the last phase $n = 2^{log_2(N)} = N$. In a first phase, the inputs are two lists containing one element and the output will be a list containing $n = 2$ elements in sorted order. But there will be $N/2$ sorted lists (of size 2) produced. In any phase, to produce a list of size n, at least $n - 1$ comparisons are required in the worst case. So for a list of size N, a phase requires $\frac{N}{n} * (n - 1)$ comparisons. But, each phase will of course process exactly N elements. The $\frac{N}{n}$ number of lists produced in each phase are completely independent, which actually means that once two lists are available, the next phase of the merge sort can already start. This allows a pipelining behaviour. In hardware, each phase is modelled as a stage. Each stage has a rate of 1 and processes N elements. As a result, in the long run, each stage has a throughput of 1 and a utilization of 100%.

In 1991 Edward Ashford Lee published a paper which presents a streaming merge node for two monotonically increasing input streams in dataflow[10]. This is essentially one stage of a complete merge sorter. The book 'Mutli-Processor System-on-Chip: Vol. 2'[1], gives a full implementation of a merge sorter in StaccatoLab, which were based on the findings of Lee. Figure 5.1 shows the implementation of the merge sorter in StaccatoLab. The implementation sorts blocks of size $N = 8$. The `src` node produces a random data stream with values between 0 and 100. The `srt` node is the first stage that does a merge sort. It takes two tokens, sorts them, and produces them back on its output edge (producing $n = 2$ sorted list). The next stage of the merge sorter actually includes two nodes, namely the `spl` and `mrg` nodes. The `spl` node is basically a switch to move the sorted sub-lists from the previous stage to the correct input edge for the `mrg` node. Once there are tokens on both the `spl!0` and `spl!1` edges, then the `mrg` node will start merging these two sub-lists to produce a sorted list of length $N = 4$. The next stage again consists of the `spl` and `mrg` nodes. Note that this `spl` node switches lists of size $N = 4$. The last `mrg` node produces a list of size $N = 8$. The number of stages the merge sorter has is equal to $\log_2(N)$.

This implementation has a rate of 1. Which means that each cycle, one token is consumed, and one token is produced. That makes this implementation a streaming sort. The average latency of a token is also relatively low and equal to exactly $N + 2 \cdot (\log_2(N) - 1)$.

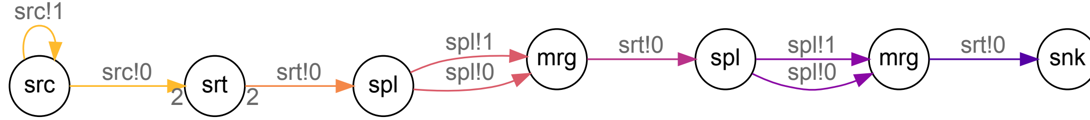


Figure 5.1: Merge sort implementation for $N = 8$

5.2 Caching

The `spl!0` and `spl!1` edges from figure 5.1 quickly become too large when N is increased for the edges to be stored in registers. For these edges, BRAM would be used instead of registers. The `mrg` node compares the front of the `spl!0` and `spl!1` edges, so it reads these from the BRAM every cycle. The BRAM in an FPGA are dual port, so it is possible to do one write and one read simultaneously each cycle. And since the `spl!0` and `spl!1` do not share the same BRAM module, it is possible to read from both edges in the same cycle as well. But when merging two lists, only the smallest value is sent to the output, the larger value will be used next cycle in the next comparison. This essentially results in two or more reads of the same value. By caching the front of both lists, it is possible to remove one of the reads. Essentially, this means that each cycle only one value is read from the `spl!0` or `spl!1` edge, but never both. This is useful later on, when the `spl!0` or `spl!1` will be combined into one edge / one BRAM.

5.3 Memory usage

Using Little's Law, it is possible to determine how many tokens will be in the system as a long-term average. Little's Law is defined as follows[11]:

$$L = \lambda W$$

- L - the average number of tokens in the system.
- λ - the average number of tokens arriving in the system per cycle.
- W - the average processing time / latency per token.

Since the system is a streaming sort, λ will be equal to 1, which means the average number of tokens in the system is equal to the latency. However, another important note is that because the system is a streaming sort, the number of tokens in the system does not change after the startup period. This means that the number of tokens will always be equal to or less than our latency $(N + 2 \cdot (\log_2(N) - 1))$. When deriving the current memory usage from the graph (which is equal to the sum of the slack of all the edges), this is equal to $2N - 2 + 2 \cdot (\log_2(N) - 1)$. Hence, $N - 2$ registers are always empty. If these registers could be removed, this would be a reduction of almost a factor of 2.

5.3.1 Removing slack

Figure 5.2(a) illustrates the problem in the current design. The `spl!0` edge (from figure 5.1) contains list A, and edge `spl!1` will contain list B as it arrives. At $t = 4$, the A list has already fully arrived and has filled up the buffer. The buffer is the slack/capacity of each edge in this case. At $t = 5$ the first token from the B list arrives at the edge. Now the merger will take the lowest valued token (in this case from the B list) and produce it on the output edge. So, at $t = 6$, the value 1 is produced and the next value of B arrived. At $t = 7$, again the B token was produced,

since it had the lowest value. At $t = 8$, the token in the A list is produced. Observe that now the B edge also contains 2 tokens, but the A edge contains 3 tokens.

After the initialization phase, on average only $\frac{5}{8}$ of the slack of the `mrg` node edges is utilized, which can also be seen in figure 5.2(a). Note that neither the slack of the A nor B edges can be reduced. The A edge will always fill up when a new sorted list arrives. And the B edge will be filled in the extreme case when all values of A are lower than all values of B. One of the ways to reduce the memory usage, is first to make the realization that the A and B edges combined use exactly $\frac{5}{8}$ places of the buffers. More specific $n/2 + 1$ per merge node. So, what can be done is that the A and B lists can be combined into one buffer. Since this can no longer be stored as slack on edges, the implementation becomes slightly more complicated, but the idea remains the same. Figure 5.2(b) shows how this works. At $t = 4$, the buffer is still being filled from the initialization phase. At $t = 5$, the buffer is filled with both the A list and the first value of B. At $t = 6$, the first value of B was produced on the output and the next value of B was placed in the buffer. At $t = 8$, the value of A list was removed, since it had the lowest value. Note that B now fills more of the buffer, and the A list fills less of the buffer.

In this implementation, all places in the buffer would always be filled. So, the memory utilization is 100%. The rate and latency remain unchanged from the baseline implementation.

5.3.2 Adding indexing

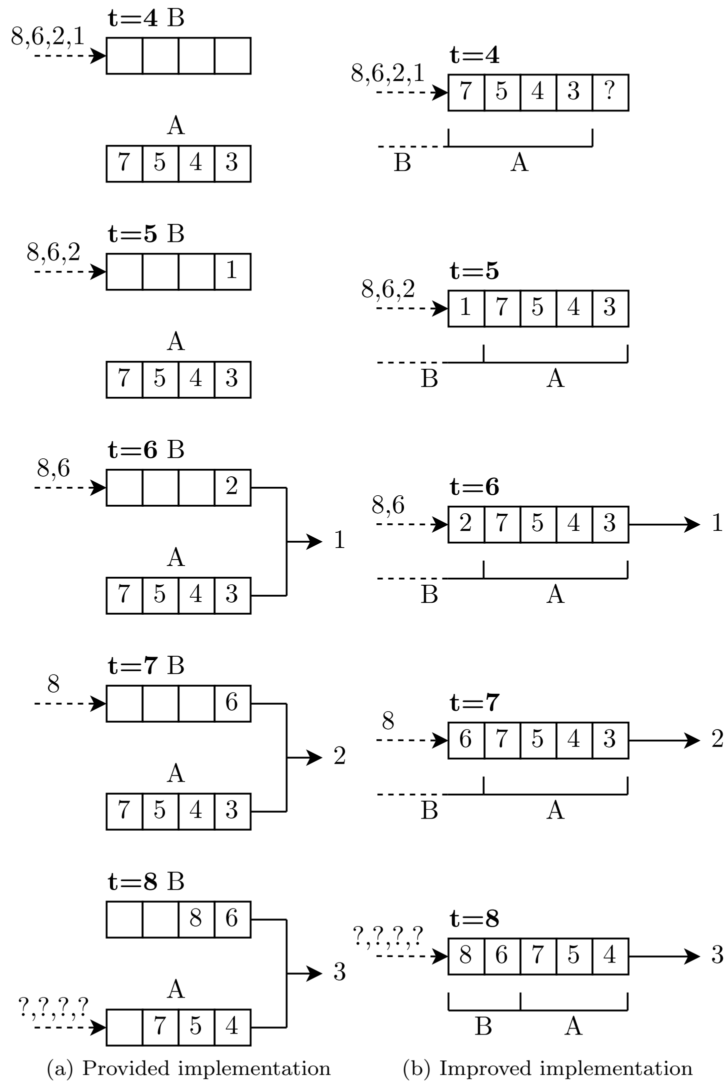
Another problem arises with the proposed solution, however. The buffer stores two FIFO queues in a shared space. This results in four cases of where tokens are placed and removed, listed below.

1. Take head from queue A, store new element at end of A
2. Take head from queue A, store new element at end of B
3. Take head from queue B, store new element at end of A
4. Take head from queue B, store new element at end of B

This prevents the implementation of a single ring buffer, since there are multiple heads and tails. As a result, this would require shifting (of parts) of the buffer, which is not feasible for large buffers and would certainly not be possible to do in a single cycle. If shifting is not allowed, a new element must be stored at the place where an element will be removed. Since, there are no guarantees if this will be (at the tail) in queue A or B, it is required to keep track where elements are stored. This adds a layer of indirection.

Because of the indirection, it is now also required to store these references to the elements, which in turn adds more memory. Zooming in again on a single merge stage, the input is two blocks of length $n/2$ and the output will be one block of length n. The buffer size will be $n/2 + 1$ (as shown in 5.3.1). The length of queue A and B will both be $n/2$ at maximum, so n places are required for the indices in total. The indices require at least $\lceil \log_2(n/2 + 1) \rceil$ bits to store the highest index. The maximum block size will always be equal to a power of two, so $\log_2(n)$ can be used instead, which has the same value when n is a power of two. Thus, the memory usage is increased for each stage with $n * \log_2(n)$ bits.

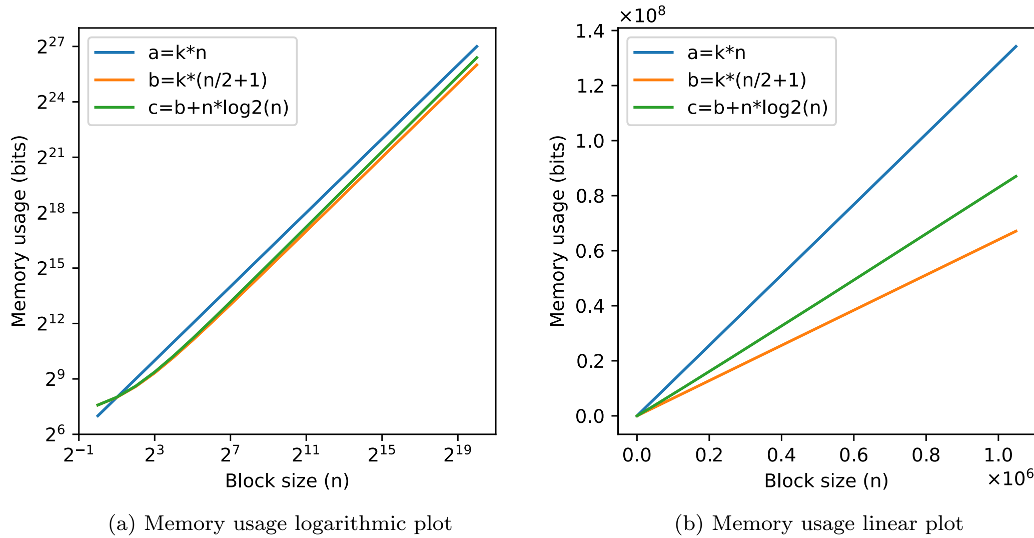
Note that the key-value pair size is also critical to determine if this indirection is beneficial for the memory usage. A larger key-value size will profit more than a small size. If the key-value size is small enough, it will even have the opposite effect and cost more memory to have this layer of indirect. For this research problem, the key-value pair uses 128-bits (64-bit key, 64-bit value), so here it will still be beneficial. Figure 5.3 shows how much memory each approach uses for a key-value size of 128-bits up to $n = 2^{20}$. The total memory usage of this solution still grows faster than the baseline solution in relation to n. However, the reference solution would only be worse after around $n > 2^{85}$, which will never be a problem (the FPGA used only has 2^{25} bits of memory, which already exceeded at $n > 2^{21}$). With an element size of 32-bits, this referencing solution



(a) Provided implementation (b) Improved implementation

Figure 5.2: Memory usage trace for the merge stage $n = 8$

would only be beneficial up to around $n = 2^{21}$, which could have been a problem if such a small element size were used.



(a) Memory usage logarithmic plot

(b) Memory usage linear plot

Figure 5.3: Memory usage comparison for a single stage with key size, k=128

5.4 Variable block sizes

The baseline solution can only sort block sizes of exactly N. Also, N must be a power of two. As stated in the problem description, the sorter should be able to sort any block size as long as the block size is smaller than the designed N. For example, if the sorter is configured for maximum size N, that sorter should be able to sort any block of size (S) as long as $1 \leq S \leq N$.

Consider the case where $N = 2$ and all blocks have a length of 1. Then it is important to alternate each block over the A and B edges, since if all blocks would enter the A edge, the select node would never fire, as illustrated in figure 5.4. The figure is a flow only view, so the numbers on each edge indicate the number of tokens, not values.

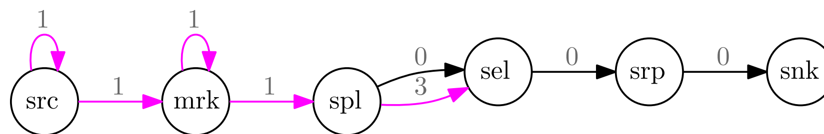


Figure 5.4: Split (spl) node sending all blocks of size 1 to edge A (flow only view)

Instead of sending blocks to a single edge, blocks will be equally distributed over the two edges. But currently there is the issue that blocks cannot be kept apart. For an input stream where the block size for every block is 1, then the output stream should be equal to the input stream (FIFO behaviour). However, if there is a block on edge A and edge B, it cannot be known which block must be selected. The solution used is to add a block number to each block. This way, it is also possible to sort on the 'oldness' of elements. The block number is combined with the value in a tuple, and will look like this: (blocknr., key). Now it is possible to do a lexicographical comparison on the elements, this way older blocks will always be considered to have 'a lower value', thus being output first. Blocks with the same block number will still be sorted by their key.

The current firing finite state machine (FSM) still expects elements of the same block on the A and B edges, which can also no longer be guaranteed with variable block sizes. Just adding the block number is not enough to make this sorter functional. The FSM would require a new design,

or, another approach is to simplify the FSM and add another value to the lexicographical ordering. Let's call this value rank, and insert it between the block number and the key: (blocknr., rank, key).

Each stage of the merge sorter sorts an increasing number of elements. E.g. the first stage sorts a block of two elements. The second stage sorts four elements, the third stage eight elements, etc. Using the rank, these elements are grouped together. So, elements that should be in the same block as a merge stage output will have the same rank. Generating the ranks is done by simply adding the count / index of the element in the block. So, the first element gets a rank of 0, the second element gets a rank of 1, the third element a rank of 2, etc. In each merge stage, the rank is divided by 2 (a simple bit shift), such that the group of each rank get larger at each merge stage as explained above. Table 5.1 shows an example of a small block of $N = 4$ with the inputs of each stage and the final output. Note that elements with the same block number and rank are always in sorted order.

	Stage	1	2	Output
Inputs	Element 0	(0, 0, 79)	(0, 0, 62)	(0, 0, 17)
	Element 1	(0, 1, 62)	(0, 0, 79)	(0, 0, 62)
	Element 2	(0, 2, 17)	(0, 1, 17)	(0, 0, 74)
	Element 3	(0, 3, 74)	(0, 1, 74)	(0, 0, 79)

Table 5.1: Merge stage input/output data

Having the rank and block number is now also sufficient to ensure that the A and B edges shall always be filled. The elements in the same block can be alternated between the A and B edges by checking if the rank is odd or even (the last bit). However, for a stream of blocks with a length of one, all elements would still be sent to the same edge, so the same can be done with the block number. The blocks are alternated over the A and B edges by checking the oddness of the blocknumber. This is combined with the rank check by a simple XOR. Figure 5.5 shows the implementation of the split node.

```
class Split(Node):
    def __init__(self, I=[]):
        super(Split, self).__init__(I=I)
        pass0 = Rule(I=(1,), O=(1,0)) # Output to edge A
        pass1 = Rule(I=(1,), O=(0,1)) # Output to edge B
        self.set_fsm([Select([pass0, pass1])])
        self.set_fs([lambda x: (x[0] & 1) ^ (x[1] & 1)]) # pass0 if 0, pass1 if 1
        self.set_fo([lambda x: (x[0],x[1]//2,x[2]), lambda x: (x[0],x[1]//2,x[2])])
```

Figure 5.5: Split (spl) node code

5.5 Implementation

All the previous discussed improvements are combined into one implementation. The code for the single-rate sorter is attached as appendix B.

A significant drawback of the block number and rank becomes visible when calculating the overhead. To guarantee uniqueness for the block and rank numbers, both need at least $\lceil \log_2((N+1)/2) \rceil = log_2(N)$ bits (since N is a power of two). Although, the rank does lose a bit after every stage. Adding the block and rank numbers increases the memory requirement for the FPGA when implemented in hardware. For small values of N, the extra memory is not significant, but at $N = 2^{20}$, it adds 40 bits to each 128 bit key-value pair, which is an increment of $\approx 31\%$. But an even more important concern is how this affects the comparators. The Xilinx VC707 uses DSP slices as a comparator. Only, each DSP slice has a bit-width of 48-bits. Since the key size is

64-bits, this requires at least 2 DSP slices. Meaning, 32-bits are left for the block number and rank. It is important to stay at or below this number when choosing the maximum block size. The maximum block size (N) is limited to 2^{16}, otherwise it would use more DSP slices. This translates to a block size of 1 MB. These overheads become significantly smaller for multi-rate sorting, as will be shown next chapter.

Formula 5.1 can be used as an approximation for the total memory usage of all stages combined for any block size N and key-value pair size k.

$$\sum_{p=2}^{\lceil \log_2(N) \rceil} (k + 2 * \log_2(N)) * (2^{p-1} + 9) + 2^p * p \tag{5.1}$$

Chapter 6

Multi-rate sorting

Multi-rate sorting will require multiple inputs and outputs to be consumed and produced each cycle. So, for example, a multi-rate sorter of rate four, consumes four inputs and produces four outputs in one cycle. StaccatoLab does not provide a way to consume multiple tokens in one cycle, so instead inputs and outputs will be bundled in one token as a tuple, e.g., `(a,b,c,d)`, where a, b, c and d are all independent key-value pairs. Let us call the sorting rate P. The tuple token width is then equal to P, such a token will also often be called an element. If $P > 1$ is used in a sorter, this is then considered a parallel sorter.

6.1 Sorting networks

A regular merge sort has no way of sorting multiple values in a single cycle. A component must be created that can take in P inputs and produce P outputs in a single cycle to create a multi-rate sorter. The most common way to do this is utilizing the well-researched sorting networks. Sorting networks are based on one component with two inputs and outputs. The inputs are compared and swapped if necessary such that the outputs are in a non-decreasing order. Using this 'swap' component, it is possible to construct a network with multiple inputs and outputs, such that all outputs are in sorted order. Sorting networks are often illustrated with horizontal and vertical lines. The data from an input follows the horizontal line from left to right and can be swapped with another horizontal line via a vertical line. The vertical line represents such a 'swap' component. Figure 6.1 shows such a large sorting network. If the direction of the 'swap' components are all the same, then the arrowheads are often omitted. There are two important properties of any sorting network. The first is the total number of comparators needed (which is equal to the amount of 'swap' components). The second is the depth of the sorting network. The depth is equal to the maximum number of comparators that can be encountered on any path. There are many variants of the sorting networks. Each variant having a different purpose. For example, minimizing these two factors, total comparator count and depth. Some sorting networks are known to be optimal[5][6]. For example, the minimal depths are known until $P = 18$ and the minimal size until $P = 12$. There are many ways to construct larger sorting networks, but these are not optimal. Sorting networks scale poorly, known (practical) implementations often have $O(n \log^2 n)$ space complexity, which is why P should be minimized if possible.

Ken Batcher published two sorting networks in 1968[2]. These are both very commonly used to implement such a network. The first network is the bitonic merge sort. Figure 6.1 shows the implementation of the network for $P = 16$. The input of each stage (each blue and green block in the figure) is a bitonic sequence. A bitonic sequence is defined as follows.

$$x_0 \leq ... \leq x_k \geq ... \geq x_{n-1} \text{ for some } k, 0 \leq k \leq n$$

In each stage the two previous stages are merged, so it should be obvious that $k = n/2$ at the input of each stage. However, each stage can actually sort any bitonic sequence for any k[15]. This

is an important fact for the next section. What can already be noticed is that the merger of each stage in the single-rate sorter has two sorted inputs. So, if one of those inputs would be 'flipped', this is a bitonic sequence. Thus, only requiring the very last stage of the bitonic merge sort (for size P). This saves many comparators. Equation 6.1 can be used to calculate the comparators of the last stage of the bitonic sorter (thus requiring a bitonic sequence as input). k is used to denote the degree of the sorting network. A sorting network of degree k has 2^k inputs.

$$\begin{cases} s(1) = 1 \\ s(k) = 2s(k-1) + 2^{k-1} \end{cases} \tag{6.1}$$

This recursive formula can also be simplified to a non-recursive variant, given by equation 6.2 below.

$$s(k) = k * 2^{k-1} \tag{6.2}$$

Observe that the depth of the last stage is simply equal to the degree (k) of the sorting network.

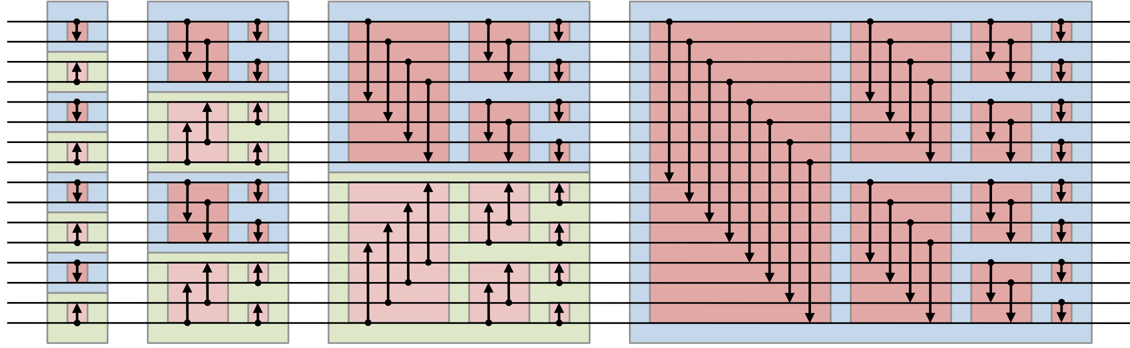


Figure 6.1: Bitonic merge sorter, source: Wikipedia[3]

The other sorting network is Batcher's odd-even merge sort. This method performs well to minimize the comparator count of the sorting network compared to other methods. However, it does not have the same property as the bitonic sort, namely that the last stage can sort a bitonic sequence[15]. So, Batcher's odd-even merge sort has a lower comparator count only if the input is not guaranteed to be a bitonic sequence. A visualization for $P = 8$ is shown in figure 6.2. It is very similar to the bitonic merge sort. The number of comparators for the last stage is reduced by $2^{k-1} + 1$ compared to the bitonic merge sort, as shown by equation 6.3.

$$o(k) = k * 2^{k-1} - 2^{k-1} + 1 \tag{6.3}$$

The total number of comparators is then shown in equation 6.4.

$$\begin{cases} h(1) = o(1) = 1 \\ h(k) = 2 * h(k-1) + o(k) \end{cases} \tag{6.4}$$

And this can also be written as the non-recursive variant shown in equation 6.5

$$h(k) = 2^{k-2} * ((k-1) * k + 4) - 1 \tag{6.5}$$

The depth of the output stage is the same as that of the bitonic variant, so for all stages this is equal to equation 6.6. And in the non-recursive variant, equation 6.7.

$$\begin{cases} d(1) = 1 \\ d(k) = d(k-1) + k \end{cases} \tag{6.6}$$

$$d(k) = (k+1) * k/2 \tag{6.7}$$

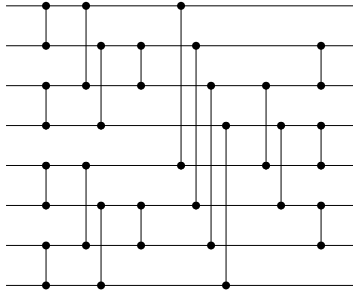


Figure 6.2: Batcher Odd-Even Mergesort, source: Wikipedia[14]

6.2 Parallel sorting

The first idea to create a parallel sorter might be to duplicate the single rate solution P times. Using some kind of parallel merge, these P input streams could be merged. However, there is a problem with this idea. Let's call these sorters $S_0..S_P$. Then for example, if all output elements from S_0 are less than the elements of $S_1..S_P$, then this would require P elements from S_0 every cycle (until all elements from S_0 are depleted). Thus, the sorters must also have a rate of P, which means the 'single-rate' sorters cannot be used. Also note that the utilization over the long term will be equal to $1/P$ for each sorter S_i (and even lower for preceding stages). So, this is not a very efficient design (since it leads to a very low utilization and high resource cost).

Another idea could be to parallelize the merge stages themselves. Currently, for the single-rate sorter, the input stream is split into two inputs at the split node. The merge node then sorts these two input streams. The split node can easily divide the inputs at rate P, so the only thing required to make this work is to find a solution for a merge node that can sort two input streams with a rate of P.

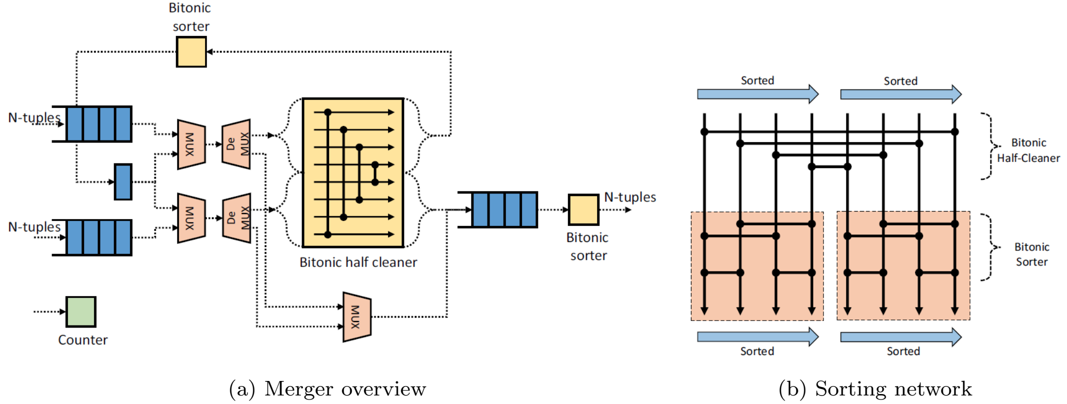
6.3 Design

A parallel merge node would have two inputs. Again, let's call these A and B. And let's call the elements a_i with a_0 being the front of the A input. And do the same for b_i, with b_0 being the front of the B input. Then since it is a merge sort, we assume that $\forall_{i,j}[0 \leq i < j < N : a_i \leq a_j \wedge b_i \leq b_j]$, in other words, the inputs of A and B will already be sorted. The problem to solve for the parallel merge node is that the P lowest value elements should be output. However, these could be all in A or all in B or a combination. This means there are $2 * P$ candidates that must be considered for only P outputs. Another factor that must be considered is, what happens with the P elements that were not the lowest?

6.3.1 Design 1 (Feedback loop)

Two papers that implemented the merge sort and solved this problem used some form of a feedback loop[9][16]. One such design is shown in figure 6.3(a). This is a merger with $P = 4$. However, it can be seen, that a sorting network of size $2 * P$ is required. The sorting network is shown in figure 6.3(b). The bitonic half-cleaner isolates the lower half values from the higher values. This is why the bitonic sorters can also be separated from the half-cleaner and the sorting of the smaller halves can be done at a later stage. Note that this is not done in the feedback loop, since there is no register between the half cleaner and the bitonic sorter. There is only one register present, the single blue box after the bitonic sorter, in the feedback loop. So, this could be considered as one edge with a slack of one in StaccatoLab.

There is one important drawback in this design. Since $2 * P$ values are sorted each cycle, the P highest values must be fed back into the sorter, hence the feedback loop. However, these must be immediately available, since they could contain the lowest values that must be output in the next



(a) Merger overview (b) Sorting network

Figure 6.3: Terabyte sort merger (where $P = 4$), from [9]

cycle. For example, consider the following inputs with $P = 2$: $A = [0, 0, 2, 2], B = [1, 1, 3, 3]$. In the first cycle the front of A is compared with the front of B, the lowest values will be the output ($[0, 0]$), the highest values ($[1, 1]$) will be fed back into the sorter. The second cycle the $[1, 1]$ is expected as the output, so this loop must already be present at the input of the bitonic sorter. Although this is possible (as shown in the existing paper), this creates a critical path that cannot be pipelined. So, this could limit the possible clock rate when implemented in hardware. Also note that this would get worse as P grows.

6.3.2 Design 2 (Min select)

For the second design, the idea was based on the fact that only the P lowest values are required to be sorted each cycle. So, if these P values can be selected from A and B, then there is no need to have such a feedback loop as in the previous design. Actually, selecting the P lowest values is not that difficult. Let us define a new variable k as the number of elements that should be picked from the (front of the) A input. Then, the number of elements picked from the (front of the) B input will be equal to $P - k$ elements, such that we have a total of P elements. This k will be picked in such a way that these elements $\{a_0..a_k\} \cup \{b_0..b_{P-k}\}$ is the subset containing the smallest values from the A and B inputs. Note that such a k does exist, as shown by the formula below, for which a full proof is provided in Appendix A. In the special cases, $a_0 \geq b_{P-1}$ and $b_0 > a_{P-1}$, $k = 0$ and $k = P$ can be used respectively.

$$\forall_{i,j}[0 \leq i < j < P : a_i \leq a_j \wedge b_i \leq b_j] \implies \\ (\exists_k[1 \leq k < P : a_{k-1} < b_{P-k} \wedge b_{P-k-1} \leq a_k] \vee a_0 \geq b_{P-1} \vee b_0 > a_{P-1})$$

Consider the following example, where $P = 4$, $A = [0, 2, 4, 6]$ and $B = [1, 3, 5, 7]$. Then $k = 2$ should be selected such that the minimum set is equal to $[0, 2, 1, 3]$. Also note that $k = 2$ satisfies the $\exists$-formula, since $a_1 < b_2 \wedge b_1 \leq a_2$ is indeed satisfied.

In hardware, this k can be found by testing all combinations, but combining the comparisons will result in exactly P comparisons total. This is because the second comparison in the $\exists$-formula is the negation of the first formula for $k-1$: $\neg(b_{P-(k-1)-1} \leq a_{k-1}) \equiv b_{p-(k-1)-1} > a_{k-1} \equiv a_{k-1} < b_{P-k}$.

When the P smallest elements are selected, the values can be arranged in the following sequence $(a_0..a_k, b_{P-k}..b_0)$, which is a bitonic sequence. This sequence can then be sorted by a bitonic sorting network as described in chapter 6.1. Note that this design can easily be pipelined, since there is no feedback loop. Figure 6.4 illustrates a simplified version of the design.

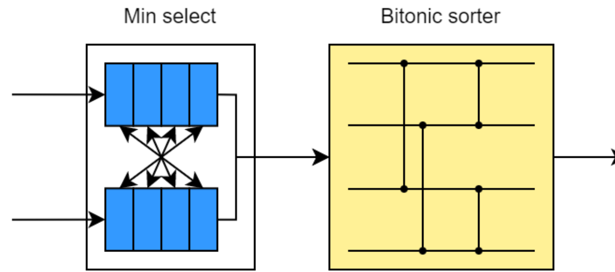


Figure 6.4: Minimum select design for $P = 4$

6.4 Implementation

To fairly compare the designs, the comparator count per stage must also be calculated. For design 1, it is assumed that only 1 comparator is needed to make a selection between the A and B queues (this is possible if the multiplexers use the same select signal, which would come from 1 comparator). The total comparator count is then equal to $1+s(\log_2(2P)) = 1+\log_2(2P)*P$ (using equation 6.2). The comparator count for design 2 is equal to $P + s(\log_2(P)) = P + \log_2(P) * P/2$. All practical values for P until $P = 64$ are shown in figure 6.5 and in table 6.1. Note that any P higher than 64 would probably be unrealistic, since this would create very long (critical) paths.

The maximum depth will be calculated from register to register. For design 1 this is simply $\log_2(P) + 1$ (on the loop). For design 2, it depends on if there is a register between the minimum selection and bitonic sorter stage. In the final implementation this is the case, so then the maximum depth is equal $\log_2(P)$ (otherwise it would be the same as that of design 1).

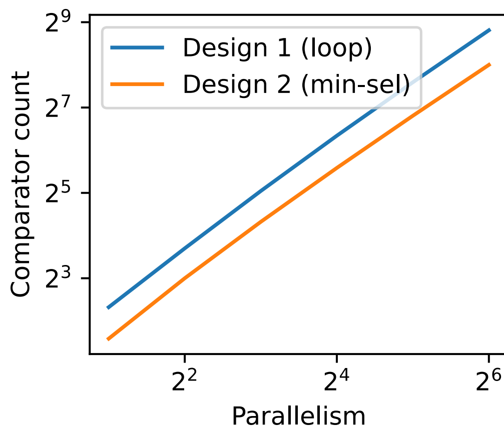


Figure 6.5: Logarithmic plot of the comparator count for designs 1 and 2.

P	2	4	8	16	32	64
Design 1	5	13	33	81	193	449
Design 2	3	8	20	48	112	256

Table 6.1: Table of the comparator count for designs 1 and 2.

From all the discussed differences between design 1 and 2, it can already be seen that design 2 is better. This is why design 2 was implemented in StaccatoLab and will be compared to the implementation of the Terabyte sort paper as a reference. The sorting is still very similar to the single rate sorting. Figure 6.6 shows the first two stages. The first stage is simply the 'net' node. This is an odd-even merging network to sort the unsorted tuples from the input. A bitonic sorter cannot be used for this, since the input is not a bitonic sequence. Fortunately, this node will only be used once and is always the first stage. The second stage is the 'spl', 'pss' and 'bnet' nodes. The 'spl' node is functionally the same as that of the single rate sorter. It moves the tuples to the correct edge based on the rank and block number. The 'pss' node does the minimum value

selection and outputs a tuple which values form a bitonic sequence. The 'bnet' finally sorts this bitonic sequence (using Batcher's bitonic sorter), such that the output tuple is fully sorted.

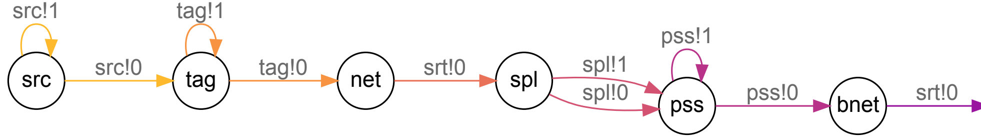


Figure 6.6: Multi-rate first two stages

In chapter 5.3.1 it was explained that the memory requirement can be halved by storing the A and B inputs in block memory (on a self loop). For the single rate implementation, it was possible to combine all the required logic in one node. For the multi-rate design, it would also be possible to implement all of this in the 'pss' node. However, this node was already very complex and adding this functionality (also with the references etc.) would make this unmanageable. So instead, this functionality was kept separate. Storing the A and B inputs is now managed by the 'buf' node, which can be seen in figure 6.7. From the 'buf' node, there are two edges to the 'pss' node. These edges represent the A and B inputs and have a small fixed slack. The 'buf' node does not know whether it needs to produce an A or B input for the 'pss' node. So, there is a signal back from the 'pss' node which signals if the last consumption was from the A or B input. Using this signal, the 'buf' knows which input/edge it should refill.

Because of this feedback loop, some latency is present. As a result, the slack of the inputs of the 'pss' node must be sufficiently large such that it always has and A and B input (otherwise the node cannot fire). This extra slack adds a small (fixed) amount of extra memory for each stage. This is the reason that for stages two and three, the 'spl' node is still used (figure 6.6). Stages four and beyond use the 'buf' node.

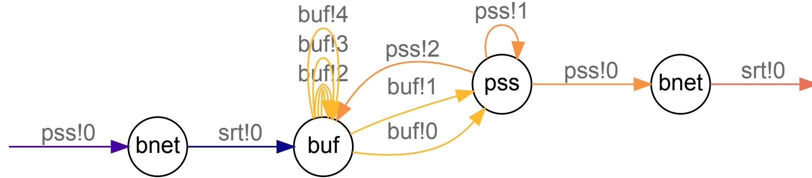


Figure 6.7: Multi-rate last stages

As a final implementation detail, the rank, and block number are attached to the tuple and not for every value in the tuple. This essentially reduces the relative memory overhead of these additional values. Unfortunately, this does constrain the variable block size to be a multiple of P. It is still possible to sort blocks that are not a multiple of P by adding padding values. However, this is not automatically managed by the sorter. The total memory consumption calculation is shown as formula 6.8. The formula has not changed much from the single rate formula, but the element size is now $k * P$ and there are few stages to sum.

$$\sum_{r=2}^{\lceil \log_2(N/P) \rceil} (P * k + 2 * \log_2(N)) * (2^{r-1} + 9) + 2^r * r \tag{6.8}$$

Again, a full implementation of a multi-rate sorter is attached in appendix C.

6.4.1 Comparison

The Terabyte sort paper is mainly constrained by reading the input from flash memory. The flash memory has a throughput of 2.4 GB/s. The paper has designed a page sorter with a throughput of 0.5 GB/s. To fully saturate the flash memory, 5 instances of the page sorter are used. Each page sorter outputs a block of 8 KB in size, and uses a key-value pair size of 64-bits (one block contains 1024 elements). In total 65 comparators are used for this sorter. The sorter operates at a clock frequency of 125MHz.

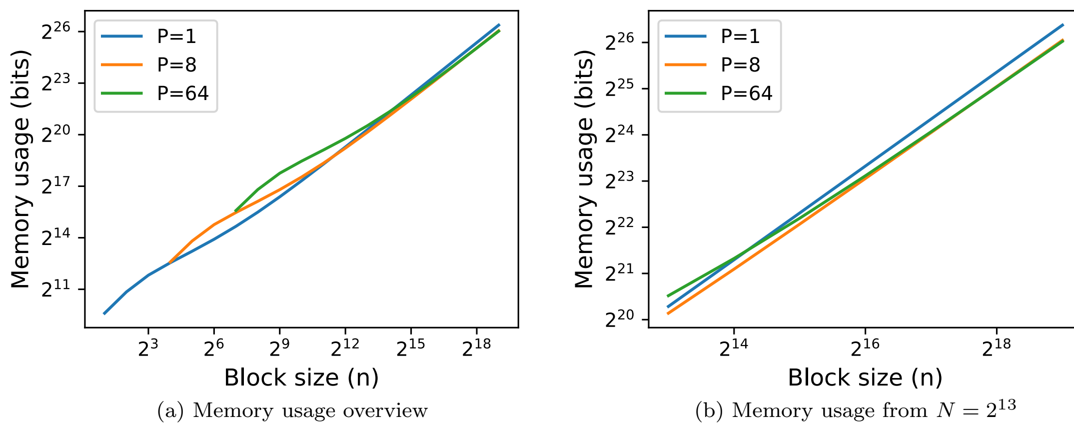
Comparing this fairly with the proposed minimum select design is a bit difficult. To get the same throughput of 2.5 GB/s, an element throughput of 312.5 Melements/second is required. This is equivalent to 2.5 elements per cycle. The elements produced per cycle should be equal to the parallelism; however, this must be a power of 2. So, the option is $P = 2$ or $P = 4$, which results in a throughput of 2 GB/s with 28 comparators or 4 GB/s with 69 comparators respectively. Achieving a throughput of 2.5 GB/s cannot be done without changing the clock frequency. However, the amount of page sorters used in the terabyte sort paper can be scaled to match the 2 GB/s and 4 GB/s throughputs, by using 4 and 8 instances of their page sorter. This would result in 52 and 104 comparators, respectively, which is significantly higher than the newly designed streaming sorter.

6.4.2 Results

Since the proposed design can also be pipelined, a higher frequency of 200MHz and parallelism of $P = 4$ is assumed to be achievable. Furthermore, a key-value pair size of 128-bits is used instead, as specified by the problem description. This would result in a throughput of 12.8 GB/s.

The memory usage of the sorter depends mainly on the block size. However, as said before, a larger level of parallelism decreases the overhead of the additional rank and block number parameters. This effect is very noticeable until around $P = 8$. Figure 6.8(a) shows the memory usage for three different levels of parallelism. Each sorter starts at $N = P$, which is why the starting points are not the same. Moreover, while the block size is still small, there is a high overhead from the caching and self loop edges. This is why they all start off high, but come closer together for larger block sizes. Figure 6.8(b) highlights the overhead effect of the additional rank and block size values. E.g., more parallelism results in lower memory usage. Moreover, the memory usage of $P = 8$ and $P = 64$ is almost the same, even for larger block sizes.

The maximum block size is limited by the on-chip memory of the FPGA it would be implemented on. The Xilinx VC707 has 4635 KB of BRAM. So, for $P = 4$ with 128-bits for key-value pairs, the maximum block size is equal 2^{18} elements, which is almost 4.2 MB.



(a) Memory usage overview

(b) Memory usage from $N = 2^{13}$

Figure 6.8: Memory usage multi-rate sorter

Chapter 7

Multi pass sorting

Currently, the maximum block size is 4.2 MB as shown in chapter 6.4.2. The limiting factor was the amount of on-board BRAM. When blocks of even larger size need to be sorted, external memory will be required. To sort larger blocks, it is then possible to make multiple passes over the data, which intermediate results can be stored in the external memory. For example, in a first pass 4 blocks of 4.2 MB can be sorted using the multi-rate sorter. Each block can be stored in the external memory. For a second pass, these blocks can be read from memory and merged with a merge sorter, such that there are now two sorted blocks of 8.4 MB. And then a third pass could do the same to create one sorted block of 16.8 MB, which is then sent to the output.

The problem with the method above is that every read and write (or pass) decreases the sorting throughput. This is because the DRAM performance limits the sorting throughput. This relation becomes clearly visible when a roofline analysis is constructed. For simplicity, when constructing the roofline, it is assumed that the input and output will also be stored in the DRAM. That means, in the first pass, each element would be read and written once. In a second pass twice, etc. To make the roofline, first, the memory performance must be calculated. The Xilinx VC707 has 1 GB of DDR3 memory. The clock rate of the memory is 1600 MHz and has a bit width of 64. This translates to a peak read/write performance of 12.8 GB/s. Secondly, the arithmetic intensity must be calculated. The arithmetic intensity is defined as the number of operations divided by the input plus output size in bytes. In this case, one comparison is considered to be one operation. The VC707 has 2800 DSP slices, which each can run at up to 741 MHz. But since one comparison requires two DSP slices, the computational roof is equal to $2800 * 741/2 \approx 1037$ Gops/s. The number of comparisons required to sort a whole block depends on the block size, the amount of parallelism (displayed in table 7.1) and the number of passes. Figure 7.1 shows how the arithmetic intensity scales as the block size increases for several values of P. From the arithmetic intensity (I_a), the operational intensity (I_o) can be calculated. Since for each pass the number of reads and writes are the same as the block size, the operational intensity is simply equal to $I_o = I_a/passes$. So, for two passes, the operational intensity is half that of the arithmetic intensity.

P	1	4	16	64
Comparators	10	69	351	1567
Cycles	1024	256	64	16
Total comparisons	10240	17664	22464	25072
Arithmetic intensity	0.625	1.08	1.37	1.53

Table 7.1: Arithmetic intensity with a block size of $N = 1024$ pairs (16 KB)

It is important to note that a higher arithmetic intensity does not necessarily mean better performance. For example, using $P = 4$ for sorting, resulted in a throughput of 12.6 GB/s. Using $P = 4$ from table 7.1, shows $I_a = 1.08$ (for sorting 16 KB). Now assuming the input and output bandwidth is also limited by 12.6 GB/s, then raising P to $P = 64$ would result in a higher

arithmetic intensity ($I_a = 1.53$). One could argue the performance is better since a higher number of operations per second is achieved, but this does not increase the actual performance. The higher P solution needs more comparisons for the same amount of work (sorting 16 KB in this case). Both $P = 4$ and $P = 64$ sort blocks of 16 KB at 12.6 GB/s. So, raising P when the sorting is already bandwidth limited (on the input or output) does not provide any benefits.

There is now enough information to construct a roofline. For the roofline, assume $P = 4$ and a block size of $N = 2^{23}$ (128 MB) needs to be sorted. The roofline for these parameters is shown in figure 7.2. It includes the operational for both single and two passes. From this figure, it is apparent that sorting will always be memory bounded. Since, to be computation bounded, an operational intensity of approximately 100 would be needed, which is not achievable. As a result, the passes should be minimized, since each extra pass significantly decreases the operational intensity.

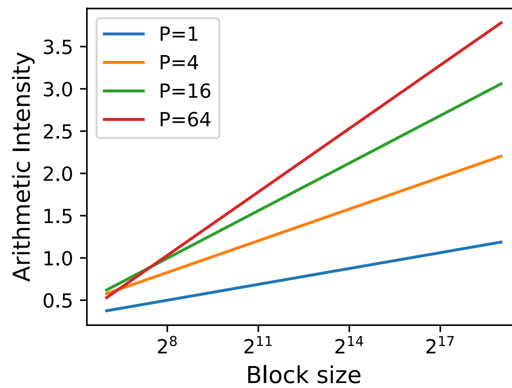


Figure 7.1: Difference in arithmetic intensity for several levels of parallelism (P)

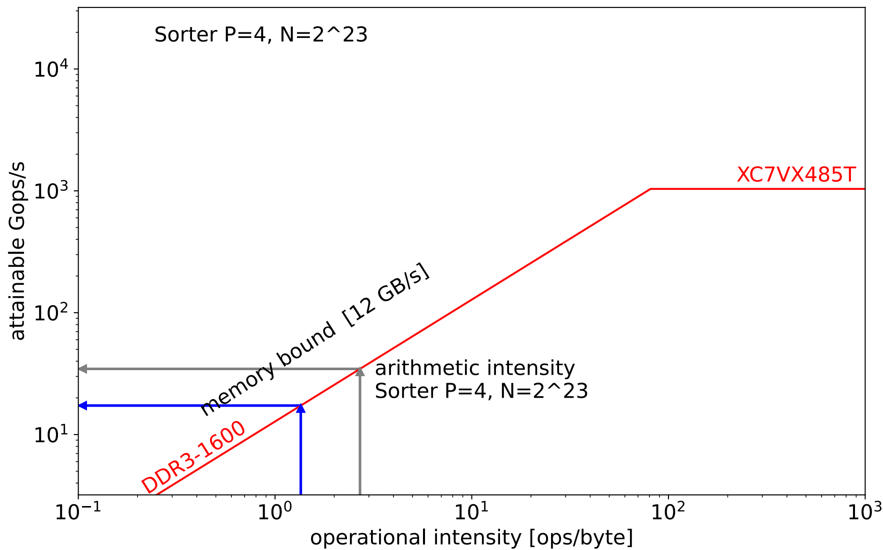
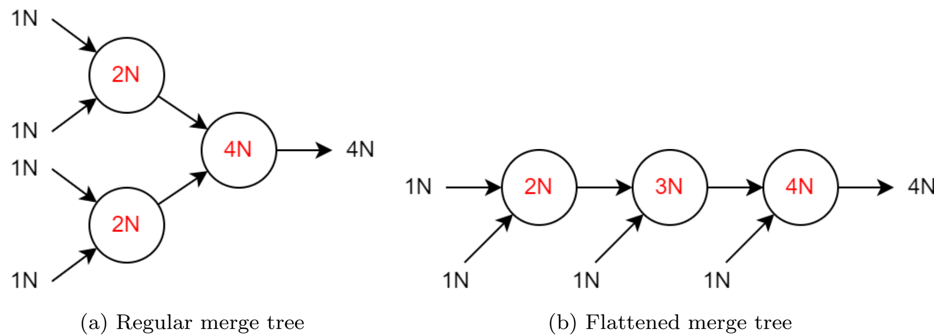


Figure 7.2: Roofline for a block size of 128 MB. The operational intensity is half the arithmetic intensity for two passes.

7.1 K-way merge

A way to reduce the number of passes could be to merge multiple blocks instead of only two per pass. This is called a K-way merge. K is used to indicate the number of inputs or blocks in this case. There are multiple ways to do this, but the most common is the K-way merge tree. Another method is the flattened variant of the merge tree. Figure 7.3 shows both variants. The main difference between these mergers is the number of comparisons that are done. For the regular merge tree, each stage uses twice the number of comparisons done in the previous stage. So, for the 4-way merge tree, this is results in $2N + 2N + 4N = 8N$ comparisons. In the flattened merge tree, each stage requires the same number of comparisons as the previous stage, plus the size of the extra block. Which for a 4-way, results in $2N + 3N + 4N = 9N$ comparisons. So, the flattened merge tree needs slightly more comparisons. However, the number of comparators is still the same, since both variants have the same amount of merge nodes. The main difference here is the utilization. The regular merge tree has a lower average utilization. This would translate to a lower energy consumption when implemented on an FPGA. Another difference is the latency. The regular merge tree has a latency of $\log_2(K)$, whereas the flattened variant has a latency of K. The advantage of the flattened merge tree is that it most likely maps better to an FPGA. However, because of all the other factors, the regular merge tree seems like the most logical choice.



(a) Regular merge tree (b) Flattened merge tree

Figure 7.3: 4-way merge tree variants

7.2 Design

The multi pass sorter will exist out of three main components:

1. Multi-rate (single pass) sorter
2. K-way merge tree
3. External memory (DRAM)

The first pass will be done by a single pass sorter to create a sorted block of length N. K blocks are then stored in DRAM. The K-way merge tree will then merge these K blocks. Because DRAM is slow when reading small amounts of data from random locations. To solve this, a 'read' or 'write' request will always be a page of size C. In the Terabyte sort paper, a page size of 8 KB was used, with which they achieved 10 GB/s writing/reading random pages to/from DRAM. So, also here $C \geq 8KB$ is assumed to get similar or better performance. Each input of the K-way merge tree should buffer a certain number of values, such that these pages can be immediately stored. The buffers will generate an event when they are almost empty, such that they are refilled in time. The required size of the buffers will be explained next chapter.

Figure 7.4 shows an overview of the design. The 'Srt' node handles the first sorting pass of the data. Via a switch, 'R/W ctrl', the read and write requests are sent to the DRAM. When it is

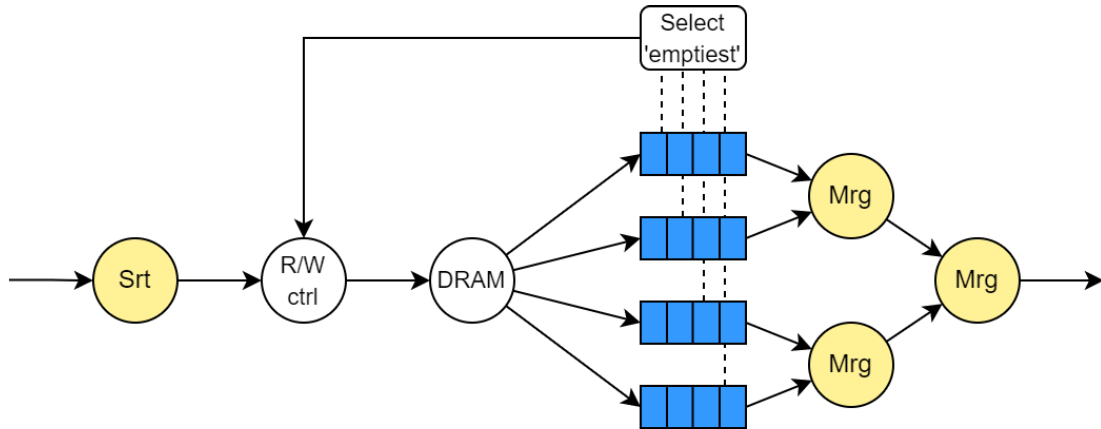


Figure 7.4: Multi pass design abstracted overview (for $K = 4$)

time to load a new page, the occupations of the buffers is read and the 'empties' buffer is selected for refilling. Note that it is a simplified design, since there is a great deal of bookkeeping logic to keep track of and load the correct pages. Furthermore, when a page is loaded, it should be sent to the correct input buffer of the merge tree.

There are two options how the writing and reading can be handled. In the first method, the reading and writing are divided into two phases. During the writing phase, $K * N$ tuples are written to DRAM. This allows for the maximum K, where $K * N$ could potentially fill the entire DRAM. During the reading phase, pages of size C are read until the DRAM is depleted, after which the writing phase will be start again. A problem with this method is that it would not be possible to do more than two passes, since 1) it would not have any space in DRAM to store the sorted output block. And 2) even if there would be space available, there is no time allocated to write the sorted output block to DRAM.

The second method uses round-robin scheduling for the reads and writes to DRAM. So, one period exists of two slices. In the first slice a page will be written and in the second slice a page will be read. In this case, it must be prevented that the writing of a page must not overwrite a page that is still pending to be read. The best way to prevent this would be to use double buffering, such that the reading and writing addresses are always separated. This solves all the problems from the first method, but the double buffering would restrict the maximum size of $K * N$ to half the DRAM size. However, as will be shown later, the first method cannot utilize the full DRAM size anyway as $K * N = 1$ GB is unreachable because of the limited available resources on the FPGA.

7.3 Buffer sizing

Choosing the correct buffer size is not a trivial task. First, it is important to realize that because the external memory is limiting the throughput, a utilization of 100% of the external memory bandwidth allows for maximum performance. So, it must be avoided at all cost that the external memory is idle. The external memory can be idle if there is no read or write request. Of course, a write request will always be available, but this cannot be ensured for the read requests yet. It turns out that the buffer sizing plays a critical role to ensure an optimal bandwidth utilization of the external memory.

For all the examples, calculations and proofs, the following assumptions were made. Firstly, for a fixed period (T), exactly one page of size C is read, and a new page is written to one input buffer. Which buffer will be written to, is decided at the start of the period and cannot change during this period T. Furthermore, for simplicity it is assumed that in the sub-period T/C, one element is read, and a new element is placed in the input buffer. All calculations are also done in

'number of elements', where an element is a tuple that contains P key-value pairs.

As an important reminder, if any of the inputs of a node is empty, the node cannot fire. So, if one input buffer is empty, the merge tree will not fire. A cycle where the merge tree cannot fire is called a stall. A consequence of the stall is that (assuming that a page is being loaded) the total number of elements in the buffers will increase.

Next, an example is given to show why the buffer sizing is critical. The first idea might be to give the K buffers a size of C. During the startup period, all K buffers are still empty. So, each period T a buffer is filled, and they will stay filled at level C until the last buffer has an element in the buffer. The first time the merge tree can fire is at time $(K-1)*T+T/C$. Now the (worst) case is considered where there is an equal draw from all the buffers (this depends on the data, but is possible since each input has an independent set of keys). At time $K*T$, all buffers will be filled up to $C-C/K$. Now a buffer should be picked to start reading a new page, but none of the buffers have space for a new page. So, clearly for any K, the buffer size must be larger than C.

From the example, it can be seen that, to ensure a read request can always be done, there must always be one buffer that has a free space of size C. Moreover, this property must be maintained when a stall occurs, since that will increase the total number of elements in the buffers.

So, to ensure 100% utilization of the DRAM, it must first be proven that there is a limit to the number of stalls that can occur, since otherwise the buffer would need to have an infinite size.

Let $q_{i,t}$ denote the number of elements in buffer i, where $1 \leq i \leq K$ at time t, where $t \in \mathbb{N}$. If at the start of each period ($p*T$, where $p \in \mathbb{N}$) the buffer chosen to be refilled will be buffer $min\{q_{1,p*T},..,q_{K,p*T}\}$. Then we can prove with strong induction that for every value of K where $K \geq 2$, there exists an s such that there cannot occur a stall ($\forall_{i,t}[i \in \mathbb{N}^+ \wedge i \leq K, t \in \mathbb{N} : q_{i,t} \geq 0]$) if the sum of all buffers is equal to or greater than this s ($\forall_t[t \in \mathbb{N} : \sum_{i=1}^{K} q_{i,t} \geq s]$).

First, the base case ($K=2$) must be proven. Then let $s = 2*C$. If $0 \leq q_{1,p*T} < q_{2,p*T}$ for some $p \in \mathbb{N}$ ($p*T$ is the start of a period), then $min\{q_{1,p*T}, q_{2,p*T}\} = q_{1,p*T}$ is selected to be refilled with a new page. The smallest $q_{1,(p+1)*T}$ can get is equal to $q_{1,p*T}$, which happens if all elements are taken from buffer 1 ($q_{2,t}$). So, we have $q_{1,(p+1)*T} \geq q_{1,p*T} \geq 0$. The smallest $q_{2,(p+1)*T}$ can get is then equal to $q_{2,p*T} - C$, which happens if all elements are taken from buffer 2 ($q_{2,t}$). Then, since $q_{1,p*T} + q_{2,p*T} = 2*C$ and $q_{1,p*T} < q_{2,p*T}$, we know that $q_{2,p*T} > C$. Also, then $q_{2,p*T} - C > 0 \implies q_{2,(p+1)*T} > 0$. Because of symmetry, it is also obvious that $q_{1,(p+1)*T} > 0$ and $q_{2,(p+1)*T} \geq 0$ holds if $0 \leq q_{2,p*T} < q_{1,p*T}$. Then for the most critical case, when $q_{1,p*T} = q_{2,p*T} = C$. Either buffer could be selected to be refilled. So, let's assume that buffer 1 ($q_{1,p*T}$) is selected. Then the lowest value of $min\{q_{1,(p+1)*T}, q_{2,(p+1)*T}\}$ happens when all elements are taken from buffer 2 ($q_{2,t}$). So, then $q_{2,(p+1)*T} = q_{2,p*T} - C = 0$. Thus also $q_{2,(p+1)*T} \geq 0$ holds. Then $s = 2*C$ is the witness for $K=2$, and thus the base case holds.

Now we must prove the same for $K > 2$. To simplify the proof, we will first prove the lemma that, at any time $t \geq 0$, if $\forall_i[1 \leq i \leq K : 0 \leq q_{i,t} \leq (\sum_{i=1}^{K} q_{i,t})/K + C]$ holds, then $max\{q_{1,t+T},..,q_{K,t+T}\} \leq (\sum_{i=1}^{K} q_{i,t})/K + C$. So, in other words, $(\sum_{i=1}^{K} q_{i,t})/K + C$ is an upper bound for the maximum number of elements in any buffer at time $t+T$. For any buffer, the only way to increase the number of elements it contains happens when a page is read from DRAM and placed in that buffer. However, the buffer that will be refilled is chosen at the start of a refill period (so, at time $p*T$ for $p \in \mathbb{N}$) and the buffer chosen to refill is $min\{q_{1,p*T},..,q_{K,p*T}\}$. The maximum amount that the number of elements in a buffer can increase in one period is C (the size of a page), and that happens when a buffer is selected to be refilled and no element is removed from this buffer throughout the refill period (so, until $(p+1)*T$). As a result, the maximum number of elements in a buffer occurs when $min\{q_{1,p*T},..,q_{K,p*T}\}$ is maximized, since then, this buffer will contain $min\{q_{1,p*T},..,q_{K,p*T}\} + C$ elements at time $(p+1)*T$. To maximize this minimum, the elements should be equally divided over all the buffers, so all buffers should contain $(\sum_{i=1}^{K} q_{i,t})/K$ elements. Hence, the maximum number of elements is equal to $(\sum_{i=1}^{K} q_{i,t})/K + C$ at time $t+T$.

Now we prove the original statement for $K > 2$. Let m be the witness from $K-1$, such that no stall could occur. Then let's choose $s = K * \frac{m+C}{K-1}$ and assume that $\forall_t[t \in \mathbb{N} : \sum_{i=1}^{K} q_{i,t} \geq s]$. We should then prove that $\sum(\{q_{1,t},..,q_{K,t}\} \cap \{max\{q_{1,t},..,q_{K,t}\}\}^c) \geq m$ for every value of $t \in \mathbb{N}$,

or equivalently, we should prove that $max\{q_{1,t},..,q_{K,t}\} \leq s - m$ for every value of $t \in \mathbb{N}$. Then from the lemma we know that maximum value of $max\{q_{1,t},..,q_{K,t}\} = s/K + C$. So, we must prove that $s/K + C \leq s - m$.

$$
\begin{aligned}
& s/K + C \leq s - m \\
\implies & (K * \frac{m+C}{K-1})/K + \frac{C*(K-1)}{K-1} \leq K * \frac{m+C}{K-1} - \frac{m*(K-1)}{K-1} \\
\implies & \frac{m+C}{K-1} + \frac{KC-C}{K-1} \leq \frac{Km+KC}{K-1} - \frac{Km-m}{K-1} \\
\implies & \frac{KC+m}{K-1} \leq \frac{KC+m}{K-1}
\end{aligned}
$$

Hence, s is a witness such that no stall can occur. Thus, the proposition also holds for $K > 2$. As a result, we have proven via strong induction that the proposition holds for all $K \geq 2$.

The value of the witness (s) can also be caught in a formula:

$$
b(k) = \begin{cases} 2C & \text{for } k = 2 \\ k * \frac{b(k-1)+C}{k-1} & \text{for } k > 2 \end{cases} \tag{7.1}
$$

So for each $K > 2$, $s = b(K)$ was used as a witness. Finally, the buffer sizes should then be equal to $b(K)/K + C$, which is derived directly from the lemma. This also means that there is always space in at least one buffer to fit a new page. So, this also ensures 100% utilization of the DRAM.

When latency is introduced in the system, then this would also influence the buffer sizes. There are two forms of latency. The first form we will call L_r, which is the read latency. So, this is the latency from the time a buffer is chosen to be refilled (at time $p * T$) until the first element from this page reaches the buffer ($p*T+L_r$). The second form of latency we will call L_s, and this is the selection latency for finding $min\{q_{1,p*T},..,q_{K,p*T}\}$. The total latency is defined as $L = L_r + L_s$. Both latencies have an impact on the lemma. If at time $p * T - L_s$, $q_{1,p*T-L_s}$ is being refilled and $\forall_i[1 \leq i \leq K : qi, p*T - L_s = \sum_{i=1}^{K} q_{i,t}$, then at time $p * T$, $\min\{q_{1,p*T-L_s},..,q_{K,p*T-L_s}\}$ is used and could mean buffer 1 is selected. As a result, there are still L_s extra elements that can be placed in the time from $p*T-L_s$ until $p*T$. The same reasoning can be used for the read latency, such that at time $(p+1)*T$ until $(p+1)*T+L_r$, there are L_r extra elements that can be placed in the buffer. As a result $max\{q_{1,p*T+L_r},..,q_{K,p*T+L_r}\} \leq (\sum_{i=1}^{K} q_{i,t})/K+C+L_s+L_r = (\sum_{i=1}^{K} q_{i,t})/K+C+L$. This changes the witness (s) in the proof. Now $max\{q_{1,t},..,q_{K,t}\} \leq s - m \implies s/K + L + C \leq s - m \implies s \geq \frac{K*(L+C+m)}{K-1}$. The definition of $b(k)$ then changes to:

$$
b(k) = \begin{cases} 2(C+L) & \text{for } k = 2 \\ k * \frac{b(k-1)+C+L}{k-1} & \text{for } k > 2 \end{cases} \tag{7.2}
$$

And then the buffers sizes should be equal to $b(K)/K + C + L$ elements. Note that the size of each buffer is increased by $2L$ at $K = 2$, but increases as K grows. Even $2L$ is already significant. For example, consider a total latency of $L = 10$ and parallelism of $P = 4$, and a key-value pair of size 128 bits. Then the increment of $2 * L * P * (128/8) = 1280$ bytes per buffer. So, it is important to limit the latency.

7.4 Implementation

7.4.1 StaccatoLab

An earlier version of the design was implemented in StaccatoLab. It uses the first method, where the reading and writing are split into two separate phases. Moreover, at the time of development, it was not yet clear that the minimum selection of the buffer occupations would be required.

Instead, the implementation works solely via events that are sent when a buffer has space for a page. The problem when 'minimum selection' is not used to prioritized buffer refilling is that there could occur a long queue of events. Each event would request a page to be loaded, but if they are not prioritized, one buffer could be empty (and at the back of the queue) and must wait for all the other read requests to finish before the empty buffer is refilled. To compensate for this, gigantic buffers would be required. The latest implementation is again provided as appendix D.

Another problem with the StaccatoLab implementation is that the simulation times are very long. Even for a single pass with $K = 4$, $P = 4$ and $N = 2048$, a simulation of 40,000 cycles takes more than 3 minutes to complete. Fortunately, there is sufficient information to create an accurate analysis for larger implementations.

7.4.2 Larger implementations

In short, multi pass sorting exists out of two stages. The first stage, 'Srt' in figure 7.4, sorts blocks until size N, which is considered the first pass. From the second pass onward, the second stage is used, which merges K blocks. If m indicates the number of passes, then $N * K^{m-1}$ indicates the final output block size. So, for an optimal implementation, the goal is to maximize the output block size in as few passes as possible and also ensuring the hardware resource limits are not exceeded. To summarize, the resource limits for the Xilinx VC707 are:

- 4746240 bytes of BRAM
- 1 GB of DRAM
- 2800 DSP slices

To create a fair final comparison with the Terabyte sort paper, the same page size is used of $C = 8$ KB in the calculations. For now, a parallelism of $P = 4$ is used. As latency $L = 10$ is assumed. When setting $K = 1$ (which results in no hardware usage for the merge tree), then $N = 262144$ key-value pairs (equivalent to 4 MB) is the maximum value of N. The resource utilization is 99.3% for the BRAM and 9.5% for the DSP slices for this N. Finding the maximum K can be done by setting $N = C$. Which finds $K = 64$ as a maximum with 68.3% memory and 36.0% DSP slice utilization. From this is it already clear that both stages are limited by the available amount of BRAM.

Finding the optimal N and K to maximize $N * K$ for a two pass solution is simply done by testing all combinations. This returns $N * K = 4194304$ pairs (64 MB) as a maximum and has two possible solutions for N and K. These solutions are shown in table 7.2. Note that, apart from the difference in utilization, the solutions are equivalent in sorting time/throughput. This is because for both solutions, the number of bytes written to and read from DRAM is identical. So for only two passes, $K = 32$ is clearly better, since it requires less resource. But, if more than two passes are done, then $K = 64$ will be better, since $N * K^{m-1}$ grows faster if K is larger.

N	K	Memory utilization	DSP slice utilization
65536	64	92.9%	44.4%
131072	32	79.3%	26.6%

Table 7.2: Optimal solutions for two passes. N is in number of pairs (128-bit per pair)

To find the optimal solution for a three pass sorter, the same strategy is used to solve for N and K, but now $N * K^2$ is maximized instead. This yields one solution, which is, $N = 65536$ pairs and $K = 64$. Then $N * K^2 = 4$ GB. Unfortunately, this exceeds the DRAM size. Since double buffering is required for more than two passes, $N * K^2 \leq 512$ MB is another restriction. When solving N and K with this extra constraint, this yields three solutions shown in table 7.3. Again, all solutions are equivalent apart from their resource usage. Both $K = 32$ and $K = 16$ seem like good solutions.

Because there is still memory available, it is probably a good idea to try to maximize the page size, as this will slightly increase the DRAM performance. The page size can be increased to 32 KB. Then only the $K = 16$ solution is still viable, but the memory utilization changes to 98.0%. However, as said before, for the comparison with the Terabyte sort paper later on, $C = 8KB$ is used instead for all calculations.

N	K	Memory utilization	DSP slice utilization
8192	64	71.4%	42.6%
32768	32	42.2%	25.5%
131072	16	62.2%	17.5%

Table 7.3: Optimal solutions for three passes. $N * K^2 = 1$ GB for all solutions.

These calculations can be made for different levels of parallelism. Of course, each level of parallelism requires a different frequency to ensure the DRAM bandwidth is maximized. Table 7.4 shows these solutions. The indicated minimum frequency is required to achieve at least 6.4 GB/s, which is half the DRAM frequency. This way, if the read and writes have a 50% duty cycle, the DRAM bandwidth is maximized. For low levels of P (until $P = 4$), there were again multiple solutions (the solutions displayed have the lowest amount of memory utilization). Solutions where $P \geq 8$, are limited by the amount of DSP slices available. And there are no solutions for $K = 64$ if $P \geq 8$ because of this. This indicates that for more than 3 passes (or even if the DRAM would be larger), $P \leq 4$ would be optimal.

P	N	K	Passes (m)	$N * K^{m-1}$ (MB)	Memory utilization	DSP slice utilization	Minimum frequency (MHz)
1	32768	32	3	512	44.4 %	3.3 %	400
2	32768	32	3	512	42.3 %	9.7 %	200
4	32768	32	3	512	42.2 %	25.5 %	100
8	32768	32	3	512	44.0 %	62.8 %	50
16	8192	16	4	512	18.9 %	86.8 %	25
32	2048	4	8	512	4.4 %	85.7 %	12.5

Table 7.4: Required N, K and passes to sort 0.5 GB for P 1 up to 32.

7.4.3 Results

The Terabyte sort paper presents a detailed performance overview of their sorter for sorting 512 GB of data. This overview can be seen in table 7.5. The performance is split per pass and there are a total of 8 passes to sort 512 GB. In the terabyte sort paper, they were also limited by the bandwidth of DRAM, but also flash, which they used as the input and output medium. Since the sorter that is proposed in this report is also bandwidth limited, the same sorting times can be assumed (note that the bandwidth is directly related to the sorting time of one pass, e.g., $\frac{2^{39} \text{ bytes}}{5e9 \text{ B/s}} \approx 110$ s).

Since 512 GB needs to be sorted, K is maximized to 64 inputs to minimize the passes. Furthermore, N is adjusted to $N = 2^{13}$ (131 KB) such that the 3rd pass outputs blocks of exactly 0.5 GB to fill half the DRAM. Then only two more passes are needed to complete the sorting. Using this configuration with $P = 4$, the memory utilization is 71.4% and the DSP slice utilization is 42.6%. So, in total, only 5 passes are required to sort 512 GB of data. This is significantly lower than the 8 passes required in the Terabyte sort paper.

Why fewer passes are required might not be clear yet. In the Terabyte sorter, a page sorter was used to sort the input stream into sorted blocks of the page size (2^{13} bytes). But the page sorter did not have enough bandwidth to handle the bandwidth of the input stream. As a result, they duplicated the page sorter 5 times, to saturate the input bandwidth. But this gives many small,

Pass	Sorted Block Size $\log_2(bytes)$	Medium	Bandwidth (GB/s)	Time (s)
1	13	flash-DRAM	2,4	220
2	17	DRAM-DRAM	5	110
3	21	DRAM-DRAM	5	110
4	25	DRAM-DRAM	5	110
5	29	DRAM-flash	2	280
6	33	flash-flash	1	520
7	37	flash-flash	1	520
8	41	flash-flash	1	520
			Total	2390

Table 7.5: 512 GB sorting performance from the Terabyte sort paper[9]

sorted blocks. In the multi-rate sorter proposed in this design, the sorting stages are sequential. Since the stages are sequential, the resulting block size is one large block (2^{17} bytes). So, the proposed design needs only 1 pass instead of 2 to sort up to 2^{17} bytes.

After the initial pass, the multi pass sorter with the merge tree is used. The Terabyte sort paper uses a 16-way merge tree (this is also clear from the 2^4 increment each pass from table 7.5). But because a feedback loop was used in their sorter, the sorter has only a bandwidth of 4 GB/s. As a result also two merge trees were used to saturate the DRAM bandwidth (5 GB/s for writing 5 GB/s for reading). In the newly proposed design, the mergers have a much higher throughput, which means only one merge tree is required. So, instead of having two separate merge trees, one and twice as large merge tree is used. However, the proposed merge tree has $K = 64$, not $K = 32$. So, there must be a second reason for this factor of 2 difference. Since, the size of the merge trees are not DSP limited, but memory limited, the only explanation is that the newly proposed input buffers are smaller than the buffers used in the Terabyte sort paper. Unfortunately, the Terabyte sort paper does not provide any information on their buffer sizes. This makes it difficult to verify, but this is the most likely.

Note that the first stage (which contains the multi-rate solution) is severely limited by the bandwidth of the flash memory. Actually, to the point where a single rate solution ($P = 1$) running at 150 MHz is sufficient to still maximize the flash throughput.

Pass	Sorted Block Size $\log_2(bytes)$	Medium	Bandwidth (GB/s)	Time (s)
1	17	flash-DRAM	2,4	220
2	23	DRAM-DRAM	5	110
3	29	DRAM-flash	2	280
4	35	flash-flash	1	520
5	41	flash-flash	1	520
			Total	1650

Table 7.6: 512 GB expected sorting performance with the proposed design

Chapter 8

Improvements

In the last chapter, we have seen that both the multi-rate sorter and the K-way merge tree are bounded by the available BRAM on the FPGA. Minimizing the memory should be a fairly high priority. This also means that, although it works well, the solution for sorting variable block sizes was a bit short-sighted. Adding two extra fields (blocknr. and rank) to each token adds a significant memory overhead on the tokens (especially when P is low). In hindsight, it would have been better to create a more complex finite state machine to handle the variable block sizes.

A problem that has been present from the single rate design has to do with the fact how dataflow works. The merge node has two inputs, but only one of the two inputs is produced to the output. But when one input is empty, the node cannot fire, since it is missing one input. As a result, some tokens get 'stuck' in the system, until new tokens are sent to the input. For the streaming sorters, this is not really an issue, since there is a constant stream of input tokens. But in the multi pass sorting, this really becomes a problem. Once a large-block has finished sorting, the last ~ 10 tokens get stuck in the merge tree. They will not be released until the next block is present in the merge tree. But this can take a long time, since the new inputs must first be stored in DRAM etc. To solve this, some sort of flushing mechanism should be implemented to force the last tokens out at the end of a large-block.

There is a chance that the input buffers of the K-way merge tree could be reduced significantly. As shown in chapter 7.3, if throughput of the DRAM and the merge tree is equal, then once the total number of elements get above a certain s, then stalls can no longer occur and the number of elements will no longer increase. The reason the number of elements increases up to s, is because a stall of the merge tree can occur, which happens if one of the inputs of the merge tree is empty. But if the throughput of the merge tree is increased by some factor, the outflow of tokens could be higher than the inflow of tokens, even when a stall occurs. This could lower the value of s significantly, and as a result, the size of the buffers also reduce. Proving this relation between the throughput and the buffer sizes is most likely a lot more difficult. However, since the K-way merge tree was also memory bounded, this could potentially allow for a higher K.

Chapter 9

Conclusions

In this research, we searched for a solution to the following problem:

How can a streaming sort
- of large blocks (of variable size) of key-value pairs be achieved
- with a fixed rate greater or equal to two
- while minimizing memory resources and traffic
- given it will be modelled in StaccatoLab
- with possibly an FPGA with external memory implementation?

In total, multiple contributions were made with designs and simulations, but there is no fully integrated solution. Because there was no FPGA implementation, the frequencies used in the results had to be estimated. These estimations were based on the existing solutions from the Terabyte sort paper.

After the first and second phase of the research, a multi-rate streaming solution was proposed. The multi-rate solution selects the minimum values in advance, so these can be sorted in parallel using a smaller bitonic sorting network. The consequence was that the feedback loop, required in traditional sorters, does not exist and thus pipelining is possible. This allows a much higher clock frequency, and as a result, a much higher bandwidth. The example that was provided uses parallelism of 4 and runs at a clock frequency of 200 MHz. This results in a bandwidth of 12.8 GB/s which is the same as the DDR3-1600 bandwidth present on the VC707. The maximum block size possible was 2^{18} key-value pairs, which is almost 4.2 MB. The limiting factor of the block size was the amount of available BRAM on the FPGA.

In the last phase of the research, a solution was given for sorting large blocks. From the roofline diagram (figure 7.2), it was clear that the throughput/sorting time of all solutions using a merge sort are always bounded by the external memory/storage bandwidth. It could also be concluded that the number of passes should be as low as possible, to keep the operational intensity high and reduce the sorting time. To achieve this, the largest possible K-way merge tree ($K = 64$ inputs) is used when more than 3 passes are required. Mostly because of the improvements to the multi-rate sorter, the largest K-way merge tree is 4 times larger than the size of the merge tree used in the Terabyte sort paper. As a result, 3 fewer passes are required than the Terabyte sort solution when sorting 0.5 GB of data. This reduces the sorting time from 2390 seconds to 1650 seconds, which is a reduction of approximately 31%. And although some aspects of the research question are incomplete, the multi pass design is a crude, first iteration solution that could solve the research problem.

Bibliography

[1] Liliana Andrade and Frederic Rousseau. *Multi-Processor System-on-Chip 2: Applications*, volume 2020, pages 145–176. Wiley-ISTE, 1 edition, 2020. 2, 12

[2] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery. 19

[3] Bitonic at English Wikipedia, CC0, via Wikimedia Commons. Bitonic Sort, 06 2011. 20

[4] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '14, page 151–160, New York, NY, USA, 2014. Association for Computing Machinery. 1

[5] Michael Codish, Luís Cruz-Filipe, Thorsten Ehlers, Mike Müller, and Peter Schneider-Kamp. Sorting Networks: to the End and Back Again. *arXiv e-prints*, page arXiv:1507.01428, July 2015. 19

[6] Michael Codish, Luís Cruz-Filipe, Michael Frank, and Peter Schneider-Kamp. Twenty-Five Comparators is Optimal when Sorting Nine Inputs (and Twenty-Nine for Ten). *arXiv e-prints*, page arXiv:1405.5754, May 2014. 19

[7] Wikimedia Commons. File:merge sort algorithm diagram.svg — wikimedia commons, the free media repository, 2020. [Online; accessed 13-June-2022]. 8

[8] Thomas H. Cormen, Charles Eric. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*, pages 191–194. The MIT Press, 3rd edition, 2009. 5

[9] Sang-Woo Jun, Shuotao Xu, and Arvind. Terabyte sort on fpga-accelerated flash storage. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–24, 2017. 1, 9, 21, 22, 34

[10] E.A. Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):223–235, 1991. 12

[11] Alberto Leon-Garcia. *Probability, Statistics, and Random Processes for Electrical Engineering*. Prentice Hall, Upper Saddle River, NJ, Verenigde Staten, 2008. 13

[12] Xingyu Liu and Yangdong Deng. Fast radix: A scalable hardware accelerator for parallel radix sort. In *2014 12th International Conference on Frontiers of Information Technology*, pages 214–219, 2014. 5, 9

[13] NVIDIA Corporation. NVIDIA GeForce GTX 1080. `https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf`, 2016. 9

[14] Octotron, CC BY-SA 3.0 ¡https://creativecommons.org/licenses/by-sa/3.0¿, via Wikimedia Commons. Batcher Odd-Even Mergesort for eight inputs, 04 2009. 21

[15] Yehoshua Perl. Better understanding of batcher's merging networks. *Discrete Applied Mathematics*, 25(3):257–271, 1989. 19, 20

[16] Artjom Rjabov. Hardware-based systems for partial sorting of streaming data. In *2016 15th Biennial Baltic Electronics Conference (BEC)*, pages 59–62, 2016. 21

[17] Paweł Russek and Kazimierz Wiatr. A24: Hardware acceleration of sorting algorithms using reconfiguration technics. *IFAC Proceedings Volumes*, 37(20):136–140, 2004. IFAC Workshop on Programmable Devices and Systems - PDS 2004, Cracow, Poland, November 18-19, 2004. 1, 9

[18] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–10, 2009. 5, 9

[19] Wikipedia contributors. Sorting algorithm — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Sorting_algorithm&oldid=1046859530`, 2021. [Online; accessed 29-September-2021]. 5, 6

[20] Wikipedia contributors. Merge sort — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Merge_sort&oldid=1086155116`, 2022. [Online; accessed 13-June-2022]. 7, 8

Appendix A

Flag derivation

(1) $A(k) := a_{k-1} < b_{P-k} \land b_{P-k-1} \leq a_k$, for $1 \leq k \leq P$

(2) $B(k) := a_{k-1} < b_{P-k} \land b_{k-1} \leq a_{P-k}$, for $1 \leq k \leq P$

(3) $\forall_{i,j}[0 \leq i < j < P : a_i \leq a_j \land b_i \leq b_j]$

(4) $\neg(a_0 \geq b_{P-1} \lor b_0 > a_{P-1})$

(5) $\neg(a_0 \geq b_{P-1}) \land \neg(b_0 > a_{P-1})$

(6) $a_0 < b_{P-1} \land b_0 \leq a_{P-1}$

(7) $\forall_k[1 \leq k < P : \neg A(k)]$

(8) **var** $m : 1 \leq m \leq P$

(9) $\forall_j[1 \leq j < m : B(j)]$

{ Case m=1; Definition of B(k) on (6): }

(10) $B(1)$, hence $B(m)$ holds for $m = 1$

{ Case $m > 1$, $\forall$-elim }

(11) $B(m-1)$

{ Definition of B(k) on (11): }

(12) $a_{(m-1)-1} < b_{P-(m-1)} \land b_{(m-1)-1} \leq a_{P-(m-1)}$

{ $\forall$-elim on (7) with $k = m - 1$ and definition of A(k): }

(13) $\neg(a_{(m-1)-1} < b_{P-(m-1)} \land b_{P-(m-1)-1} \leq a_{(m-1)})$

{ De Morgan: }

(14) $\neg(a_{(m-1)-1} < b_{P-(m-1)}) \lor \neg(b_{P-(m-1)-1} \leq a_{(m-1)})$

{ $\lor$-elim on (12) and (14): }

(15) $\neg(b_{P-(m-1)-1} \leq a_{(m-1)})$

(16) $b_{P-(m-1)-1} > a_{(m-1)}$

(17) $a_{m-1} < b_{P-(m-1)-1}$

(18) $a_{m-1} < b_{P-m}$

{ $\forall$-elim on (7) with $k = P - (m-1)$ and definition of A(k): }

(19) $\neg(a_{(P-(m-1))-1} < b_{P-(P-(m-1))} \wedge b_{P-(P-(m-1))-1} \leq a_{P-(m-1)})$

(20) $\neg(a_{P-m} < b_{m-1} \wedge b_{(m-1)-1} \leq a_{P-(m-1)})$

(21) $\neg(a_{P-m} < b_{m-1})$

(22) $a_{P-m} \geq b_{m-1}$

(23) $b_{m-1} \leq a_{P-m}$

{ $\wedge$-intro on (18) and (23): }

(24) $a_{m-1} < b_{P-m} \wedge b_{m-1} \leq a_{P-m}$, hence B(m) holds.

{ For every value of m, we have: }

(25) $B(m)$

{ $\implies$-intro on (9) and (25): }

(26) $\forall_j[1 \leq j < m : B(j)] \implies B(m)$

{ $\forall$-intro on (8) and (26): }

(27) $\forall_m[1 \leq m \leq P : \forall_j[1 \leq j < m : B(j)] \implies B(m)]$

{ Strong induction on (27): }

(28) $\forall_n[1 \leq n \leq P : B(n)]$

{ $\forall$-elim on (28) for $n = P$ and definition of B(k): }

(29) $a_{P-1} < b_0 \wedge b_{P-1} \leq a_0$

(30) $a_{P-1} < b_0$

{ $\forall$-elim on (3) for i=0, j=P-1: }

(31) $a_0 \leq a_{P-1} \wedge b_0 \leq b_{P-1}$

(32) $a_0 \leq a_{P-1} < b_0$

(33) $a_0 < b_0$

(34) $b_{P-1} \leq a_0$

(35) $b_0 \leq b_{P-1} \leq a_0$

(36) $b_0 \leq a_0$

(37) $a_0 \geq b_0$

(38) $\neg(a_0 < b_0)$

{ $False$-intro on (33) and (38): }

(39) $False$

{ ∃-intro on (7) and (39): }

(40) $\exists_k[1 \leq k < P : A(k)]$

{ ∨-intro on (4) and (40): }

(41) $\exists_k[1 \leq k < P : A(k)] \vee a_0 \geq b_{P-1} \vee b_0 > a_{P-1}$

{ $\implies$-intro on (3) and (41): }

(42) $\forall_{i,j}[0 \leq i < j < P : a_i \leq a_j \wedge b_i \leq b_j] \implies$

$(\exists_k[1 \leq k < P : A(k)] \vee a_0 \geq b_{P-1} \vee b_0 > a_{P-1})$

Appendix B

Single pass sorting code

In [ ]:

```
import sys
from    StaccatoLab import *
import  numpy as np
```

In [ ]:

```
class Split(Node):
    def __init__(self, I=[]):
        super(Split, self).__init__(I=I)
        pass0 = lambda s: Rule(I=(1,), O=(1,0), s=s)
        pass1 = lambda s: Rule(I=(1,), O=(0,1), s=s)
        self.set_fsm([Select([pass0(0), pass1(0)])])
        self.set_fs([lambda x: ((x[0] & 1) ^ (x[1] & 1))])
        self.set_fo([lambda x: (x[0],x[1]//2,x[2]),
                     lambda x: (x[0],x[1]//2,x[2])])

class Sel(Node):
    def __init__(self, I=[]):
        super(Sel, self).__init__(I=I)
        self.f= lambda x: x[0] <= x[1]
        pass0 = lambda s: Rule(I=(1,0), O=(1,), s=s)
        pass1 = lambda s: Rule(I=(0,1), O=(1,), s=s)
        self.set_fsm([Select([pass0(0), pass1(0)])])
        self.set_fs([lambda x: 0 if self.f(x) else 1])
        self.set_fo([lambda x: x[0] if self.f(x) else x[1]])

class Merge(Node):
    def __init__(self, I=[], N=2):
        super(Merge, self).__init__(I=[self, self, self, self, I])
        # x[0] - Array token (A/B lists)
        # x[1] - indices a
        # x[2] - indices b
        # x[3] - Cache + len(A) (cache (a,idx), cache (b,idx), len(A))
        # x[4] - Input token

        # len(a) len(b) helper functions
        self.len_a = lambda x: x[3][2]
        self.len_b = lambda x: (N//2+1)-self.len_a(x)

        self.tin = lambda x: (x[4][0], x[4][1]//2, x[4][2])

        # How to do this without concatenate?
        self.replace = lambda a,d,i : (
            np.concatenate([a[:i], [d], a[i+1:]])
        )

        # Decide to take from a or b,
        # make sure len(b) > 0 else take a.
        # Otherwise a if a <= b else b
        # True  - Take A
        # False - Take B
        self.ft = lambda x: (
            True if self.len_b(x) == 0 else
            False if self.len_a(x) == 0 else
            tuple(x[3][0]) <= tuple(x[3][1])
        )

        # Decide to place in a or b
        # True  - Place A
        # False - Place B
        self.fp = lambda x: (x[4][1] % 2) == 0

        # Valid b cache?
```

```python
        self.b_cached = lambda x: self.len_b(x)-(not self.ft(x)) <= 0

        # Get index to replace
        self.idx = lambda x: x[3][0][1] if self.ft(x) else x[3][1][1]

        # FSM to take token from a or b index list
        tapa = lambda s: Rule(I=(1,1,0,1,1), O=(1,1,0,1,1), s=s)
        tapb = lambda s: Rule(I=(1,1,0,1,1), O=(1,0,1,1,1), s=s)
        tbpa = lambda s: Rule(I=(1,0,1,1,1), O=(1,1,0,1,1), s=s)
        tbpb = lambda s: Rule(I=(1,0,1,1,1), O=(1,0,1,1,1), s=s)
        byps = lambda s: Rule(I=(1,0,0,1,1), O=(1,0,0,1,1), s=s)
        empb = lambda s: Rule(I=(1,0,0,1,1), O=(1,1,0,1,1), s=s)
        filb = lambda s: Rule(I=(1,1,0,1,1), O=(1,0,0,1,1), s=s)
        self.set_fsm([Select([tapa(0), tapb(0), tbpa(0), tbpb(0),
                              byps(0), empb(0), filb(0)])])
        self.set_fs([lambda x:
            4 if (self.len_a(x) == 1 and     self.fp(x) and     self.ft(x)) or
                (self.len_b(x) == 1 and not self.fp(x) and not self.ft(x)) else
            5 if (self.len_b(x) == 1 and     self.fp(x) and not self.ft(x)) else
            6 if (self.len_b(x) == 0 and not self.fp(x)) else
            2*(not self.ft(x)) + (not self.fp(x))]
        )

        # Output functions
        self.set_fo([
            # Next array value
            lambda x: self.replace(x[0], self.tin(x), self.idx(x)),
            # Indices a
            lambda x: self.idx(x),
            # Indices b
            lambda x: self.idx(x),
            # Cache, len(a)
            lambda x: (
                x[3][0] if not self.ft(x) else
                (self.tin(x), self.idx(x)) if self.len_a(x) == 1 else
                (tuple(x[0][x[1]]), x[1]),

                (self.tin(x), self.idx(x)) if
                (self.len_b(x) == 0 and not self.fp(x)) or
                (self.len_b(x) == 1 and not self.fp(x) and not self.ft(x)) else
                x[3][1] if self.ft(x) else
                (tuple(x[0][x[2]]), x[2]),

                self.len_a(x) - self.ft(x) + self.fp(x)
            ),
            # Output value
            lambda x: x[3][0][0] if self.ft(x) else x[3][1][0]
            # lambda x: tuple(
            #     x[0][0] if self.ft(x) else
            #     x[0][self.len_a(x)]
            # )
        ])

class Sort(Subgraph):
    def __init__(self, I=[], N=2):
        super(Sort, self).__init__(I=I)
        self.N = N
        self.set_attributes(label='Sort.'+str(N))
        if N&(N-1) != 0:
            print('N must be a power of 2')
        elif N==2:
            self.spl = Split(I=self.I[0])
            self.sel = Sel(I=[self.spl, self.spl])
        else:
```

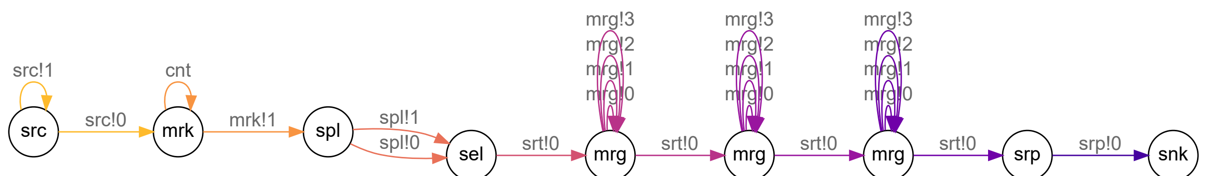
```
            self.srt = Sort (I=self.I[0], N=N//2)
            self.mrg = Merge(I=self.srt, N=N)
        Conn(self.sel if N==2 else self.mrg, self.O[0])
        self.pop()
    def init(self):
        if self.N>2:
            buff_size = self.N//2+1
            self.mrg.O[0].init(D=1, x=np.array([(0,0,0)]*buff_size))
            self.mrg.O[1].init(D=buff_size-1,
                               x=list(range(1,buff_size)))
            self.mrg.O[2].init(D=0, S=buff_size)
            self.mrg.O[3].init(D=1,
                x=(((0,0,0),0), ((0,0,0),-1), buff_size))
        elif self.N==2:
            self.spl.O[0].init(S=3)
            self.spl.O[1].init(S=2)
```

In [ ]:

```
N=16
BS=N

Gsort = Graph ()
G = Gsort
G.src = SRC (f= lambda x: np.int16(np.random.randint(1, 99)))

G.mrk = Node(I=G.src, fo=[lambda x: x[1]+1,
                           lambda x: (x[1]//BS, x[1]%BS, x[0])])
G.cnt = Edge(G.mrk, G.mrk, D=1, x=0)

G.srt = Sort(I=G.mrk, N=N)

G.srp = Node(I=[G.srt], fo=lambda x: x[2])

G.snk = Node(I=[G.srp])

G.build(make_circuit=False)

G.plot_graph(depth=8)
```

```
Gsort (Graph)      : no errors
```

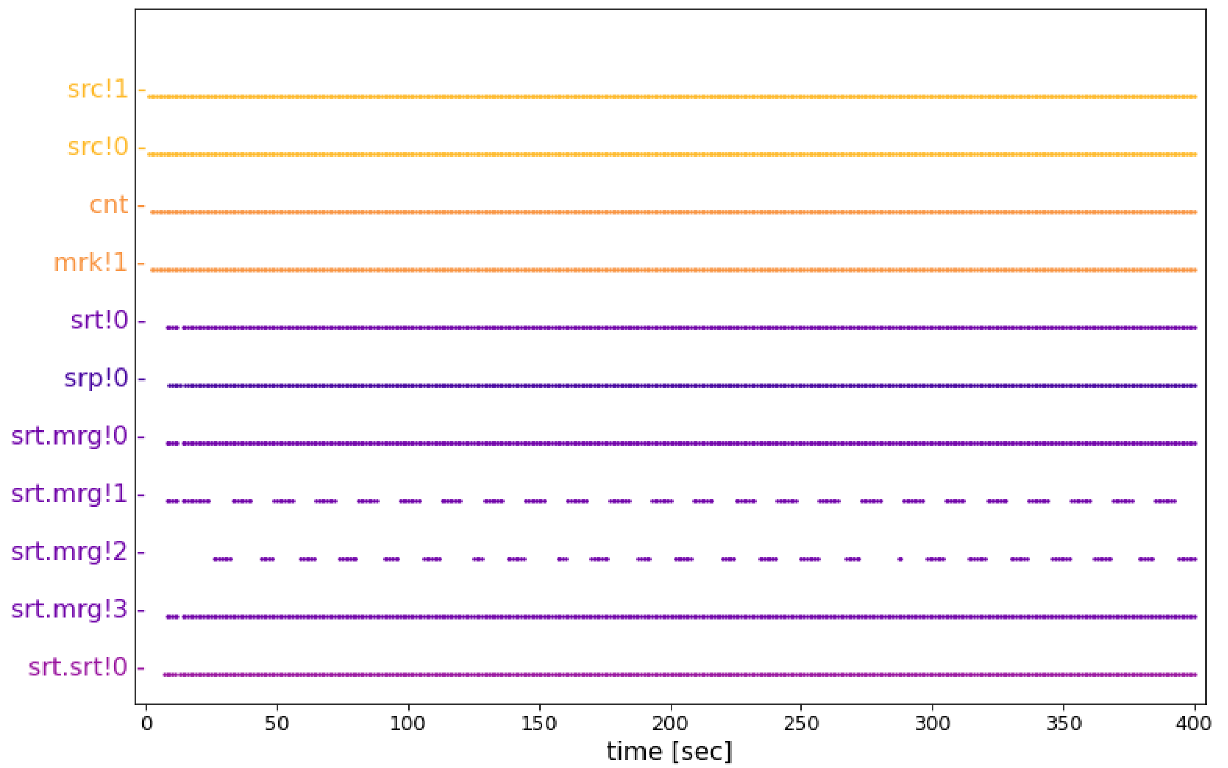
Out[ ]:



In [ ]:

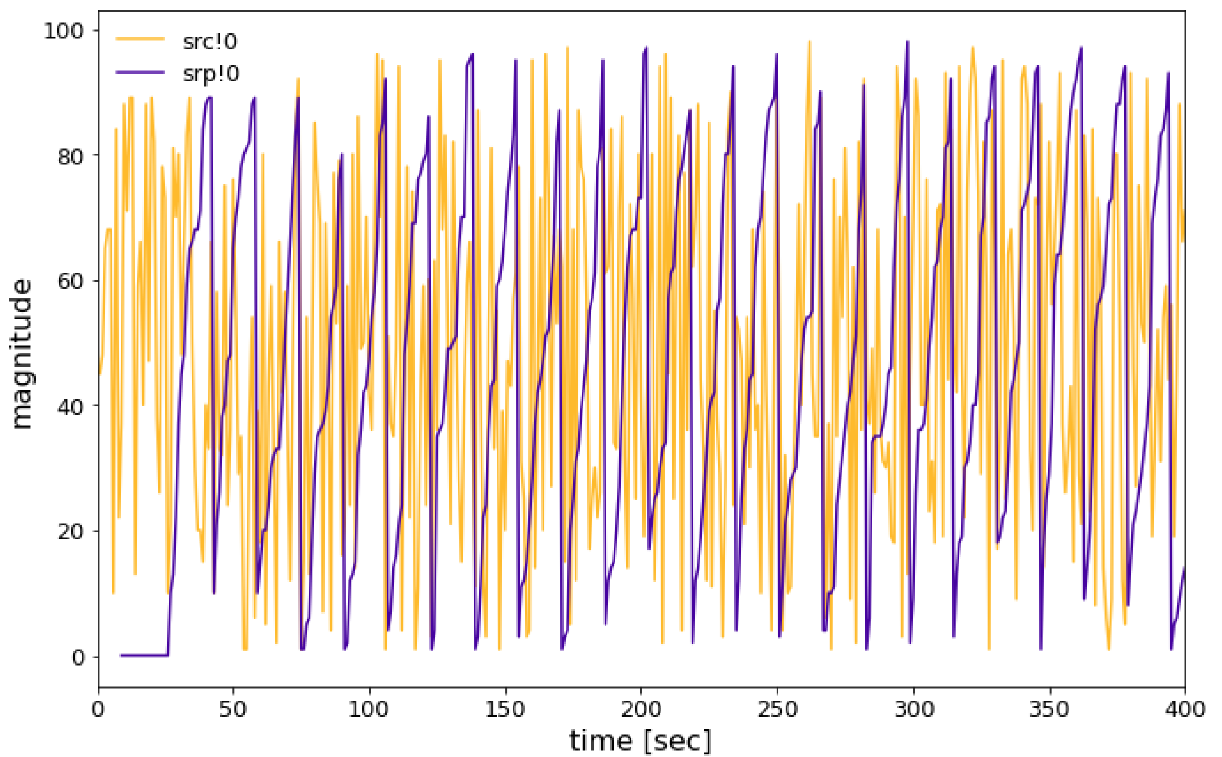
```
np.random.seed(0)
Gsort.reset()
Gsort.sim(T=400)
Gsort.plot_flow(Edges=Gsort._E+Gsort.srt._E);
```

```
#cycles real time  cpu time   #events  SDF=N  rate=1.000 Hz
    400      400s      1.0s      7192  pause (7ke/cs)
```



In [ ]:

```
G.plot_data(Edges=[Gsort.src.O[0], Gsort.snk.I[0]]);
```



In [ ]:

```
din = Gsort.src.O[0].data()
dat  = Gsort.snk.I[0].data()

latency = 0
for d in dat:
    if (d > 0):
        break
    latency += 1

dat = dat[latency:]
dat = dat[:BS*(len(dat)//BS)]
```

```python
din = din[:len(dat)]

split = [dat[i:i+BS] for i in range(0, len(dat), BS)]
splin = [din[i:i+BS] for i in range(0, len(din), BS)]

# print(split)
errors = 0
for i in range(0, len(split)):
    if not np.array_equal(split[i], sorted(splin[i])):
        print('Block', i, 'is not sorted')
        print('Block was', split[i])
        print('expected ', sorted(splin[i]))
        errors += 1

if errors == 0:
    print("No errors detected. Output matches sorted input. Checked",
        len(split), "blocks.")
```

```
No errors detected. Output matches sorted input. Checked 23 blocks.
```

Appendix C

Multi-rate sorting code

In [ ]:

```
import sys
from    StaccatoLab import *
import  numpy as np
```

In [ ]:

```
class Net4(Node):
    def __init__(self, I=[]):
        super(Net4, self).__init__(I=I)
        self.set_attributes(label='net')
        self.con = lambda a,b: ((a,b) if a < b else (b,a))
        self.n0 = lambda x: self.con(          x[0],           x[1] )
        self.n1 = lambda x: self.con(          x[2],           x[3] )
        self.n2 = lambda x: self.con( self.n0(x)[0], self.n1(x)[0] )
        self.n3 = lambda x: self.con( self.n0(x)[1], self.n1(x)[1] )
        self.n4 = lambda x: self.con( self.n3(x)[0], self.n2(x)[1] )

        self.set_fo(lambda x: (
            self.n2(x)[0],
            self.n4(x)[0],
            self.n4(x)[1],
            self.n3(x)[1],
            x[4],
            x[5]
        ))
```

In [ ]:

```
class BNet4(Node):
    def __init__(self, I=[]):
        super(BNet4, self).__init__(I=I)
        self.set_attributes(label='bnet')
        self.con = lambda a,b: ((a,b) if a < b else (b,a))
        # self.n0 = lambda x: self.con(          x[0],           x[1] )
        # self.n1 = lambda x: self.con(          x[2],           x[3] )
        # self.n2 = lambda x: self.con( self.n0(x)[0], self.n1(x)[0] )
        # self.n3 = lambda x: self.con( self.n0(x)[1], self.n1(x)[1] )
        # self.n4 = lambda x: self.con( self.n3(x)[0], self.n2(x)[1] )
        self.n0 = lambda x: self.con( x[0],          x[2]          )
        self.n1 = lambda x: self.con( x[1],          x[3]          )
        self.n2 = lambda x: self.con( self.n0(x)[0], self.n1(x)[0] )
        self.n3 = lambda x: self.con( self.n0(x)[1], self.n1(x)[1] )

        self.set_fo(lambda x: (
            self.n2(x)[0],
            self.n2(x)[1],
            self.n3(x)[0],
            self.n3(x)[1],
            x[4],
            x[5]
        ))
```

In [ ]:

```
class Buffer(Node):
    def __init__(self, N=2**10, W=1, I=[], T=None):
        super(Buffer, self).__init__(I=I, T=T)
        self.size = N*W
        self._graph._errors.check(self, self.size, 'size', 'scalar', 'nat')
        self._graph._errors.check(self, W,    'W',    'scalar', 'nat')
        self.W  = W
        self.ram= np.zeros(self.size, dtype=np.int16)
        self.a = Edge (self.O[2], self.I[1])
        self.a.init(D=1, x=[self.ram], incremental=True)    # initial token=RAM
```

```
        self.i = [Edge(self.O[3], self.I[2]), Edge(self.O[4], self.I[3])]
        self.i[0].init(D=N//2, S=N, x=list(range(0,N//2)))
        self.i[1].init(D=N//2+1, S=N, x=list(range(N//2,N)))

        # x[0] - Input
        # x[1] - Read/Write Address
        # x[2] - Indices A
        # x[3] - Indices B
        # x[4] - Feedback

        self.convi = lambda x: (x[0][0], x[0][1], x[0][2],
                                x[0][3], x[0][4], x[0][5]//2)

        tapa = Rule(I=[1,0,1,0,1], Of=[1,0,0,1,0], Od=[1,0,1,1,0], s=0)
        tapb = Rule(I=[1,0,0,1,1], Of=[0,1,0,1,0], Od=[0,1,1,1,0], s=0)
        tbpa = Rule(I=[1,0,1,0,1], Of=[1,0,0,0,1], Od=[1,0,1,0,1], s=0)
        tbpb = Rule(I=[1,0,0,1,1], Of=[0,1,0,0,1], Od=[0,1,1,0,1], s=0)
        absorb = Rule(I=[1,0,0,0,0], Of=[0,0,0,0,0], Od=[0,0,0,0,0], s=0)
        self.set_fsm([Select([tapa, tapb, tbpa, tbpb, absorb])])
        self.set_fs([lambda x:
            4 if x[0][0] == 0 else (
            2*((x[0][4] & 1) ^ (x[0][5] & 1)) +
            x[4]
            )
        ])

        self.fin= lambda x: x[2] if x[4] == 0 else x[3]
        self.ro = lambda x: self.fin(x)*W
        self.fh = lambda x: ((False))
        self.fh_m= 'address out of range'
        self.fo =[lambda x: tuple(x[1][x[2]*self.W:(x[2]+1)*self.W]),
                  lambda x: tuple(x[1][x[3]*self.W:(x[3]+1)*self.W]),
                  # (address, value)
                  lambda x: (self.ro(x), self.convi(x)),
                  self.fin,
                  self.fin
        ]

    def write(self, a, v, M):
        print('RAM: write: ', a, v)
        M[a:a+self.W]= v
        # print('RAM: write: ', M)
        return M
```

In [ ]:

```
class Split(Node):
    def __init__(self, I=[]):
        super(Split, self).__init__(I=I)

        take_a = Rule(I=[1], O=[1,0], s=0)
        take_b = Rule(I=[1], O=[0,1], s=0)
        self.fsm.add(0, Select([take_a, take_b]))
        # Switch A and B for each block:
        self.set_fs(lambda x: (x[4] & 1) ^ (x[5] & 1))

        fo = lambda x: (x[0], x[1], x[2], x[3], x[4], x[5]//2)
        self.set_fo([fo, fo])

class Pass(Node):
    def __init__(self, I=[], feedback=False):
        super(Pass, self).__init__(I=I)
        self.M = 4
        self.d = Edge(self.O[1], self.I[2])
        self.d.init(D=1,x=[(self.M,self.M)])
```

```
self.ia = lambda x, i: ((x[2][0]+i)//self.M, (x[2][0]+i)%self.M)
self.ib = lambda x, i: ((x[2][1]+i)//self.M, (x[2][1]+i)%self.M)

# Comparing (a3,a2,a1,a0) with (b3,b2,b1,b0)
# a0 - x[0][self.ia(x,0)[0]][self.ia(x,0)[1]]
# a1 - x[0][self.ia(x,1)[0]][self.ia(x,1)[1]]
# a2 - x[0][self.ia(x,2)[0]][self.ia(x,2)[1]]
# a3 - x[0][self.ia(x,3)[0]][self.ia(x,3)[1]]
# b0 - x[1][self.ib(x,0)[0]][self.ib(x,0)[1]]
# b1 - x[1][self.ib(x,1)[0]][self.ib(x,1)[1]]
# b2 - x[1][self.ib(x,2)[0]][self.ib(x,2)[1]]
# b3 - x[1][self.ib(x,3)[0]][self.ib(x,3)[1]]

self.ca = lambda x: (
    0 if ( x[1][self.ib(x,3)[0]][4], x[1][self.ib(x,3)[0]][5],
           x[1][self.ib(x,3)[0]][self.ib(x,3)[1]]) <
         ( x[0][self.ia(x,0)[0]][4], x[0][self.ia(x,0)[0]][5],
           x[0][self.ia(x,0)[0]][self.ia(x,0)[1]]) else
    4 if ( x[0][self.ia(x,3)[0]][4], x[0][self.ia(x,3)[0]][5],
           x[0][self.ia(x,3)[0]][self.ia(x,3)[1]]) <
         ( x[1][self.ib(x,0)[0]][4], x[1][self.ib(x,0)[0]][5],
           x[1][self.ib(x,0)[0]][self.ib(x,0)[1]]) else
    1 if ( x[1][self.ib(x,2)[0]][4], x[1][self.ib(x,2)[0]][5],
           x[1][self.ib(x,2)[0]][self.ib(x,2)[1]]) <
         ( x[0][self.ia(x,1)[0]][4], x[0][self.ia(x,1)[0]][5],
           x[0][self.ia(x,1)[0]][self.ia(x,1)[1]]) else
    3 if ( x[0][self.ia(x,2)[0]][4], x[0][self.ia(x,2)[0]][5],
           x[0][self.ia(x,2)[0]][self.ia(x,2)[1]]) <
         ( x[1][self.ib(x,1)[0]][4], x[1][self.ib(x,1)[0]][5],
           x[1][self.ib(x,1)[0]][self.ib(x,1)[1]]) else
    2
)
self.cb = lambda x: self.M-self.ca(x)

cons_a  = Rule(I=[1,0,1], O=[1,1], s=0)
cons_b  = Rule(I=[0,1,1], O=[1,1], s=0)
self.set_fsm([Select([cons_a, cons_b])])

self.ifs= lambda x: (
    0 if x[2][0] >= self.M else 1
)
self.fs = self.ifs

self.fo0 = lambda x: (
    ( x[1][self.ib(x,3)[0]][self.ib(x,3)[1]],
      x[1][self.ib(x,2)[0]][self.ib(x,2)[1]],
      x[1][self.ib(x,1)[0]][self.ib(x,1)[1]],
      x[1][self.ib(x,0)[0]][self.ib(x,0)[1]],
      x[1][self.ib(x,0)[0]][4],
      x[1][self.ib(x,0)[0]][5]) if self.ca(x) == 0 else
    ( x[0][self.ia(x,0)[0]][self.ia(x,0)[1]],
      x[1][self.ib(x,2)[0]][self.ib(x,2)[1]],
      x[1][self.ib(x,1)[0]][self.ib(x,1)[1]],
      x[1][self.ib(x,0)[0]][self.ib(x,0)[1]],
      x[1][self.ib(x,0)[0]][4],
      x[1][self.ib(x,0)[0]][5]) if self.ca(x) == 1 else
    ( x[0][self.ia(x,0)[0]][self.ia(x,0)[1]],
      x[0][self.ia(x,1)[0]][self.ia(x,1)[1]],
      x[1][self.ib(x,1)[0]][self.ib(x,1)[1]],
      x[1][self.ib(x,0)[0]][self.ib(x,0)[1]],
      x[1][self.ib(x,0)[0]][4],
      x[1][self.ib(x,0)[0]][5]) if self.ca(x) == 2 else
    ( x[0][self.ia(x,0)[0]][self.ia(x,0)[1]],
```

```
                x[0][self.ia(x,1)[0]][self.ia(x,1)[1]],
                x[0][self.ia(x,2)[0]][self.ia(x,2)[1]],
                x[1][self.ib(x,0)[0]][self.ib(x,0)[1]],
                x[1][self.ib(x,0)[0]][4],
                x[1][self.ib(x,0)[0]][5]) if self.ca(x) == 3 else
              ( x[0][self.ia(x,0)[0]][self.ia(x,0)[1]],
                x[0][self.ia(x,1)[0]][self.ia(x,1)[1]],
                x[0][self.ia(x,2)[0]][self.ia(x,2)[1]],
                x[0][self.ia(x,3)[0]][self.ia(x,3)[1]],
                x[0][self.ia(x,0)[0]][4],
                x[0][self.ia(x,0)[0]][5])
        )
        self.fo1 = lambda x: (
            (x[2][0]-self.M if self.ifs(x) == 0 else x[2][0]) + self.ca(x),
            (x[2][1]-self.M if self.ifs(x) == 1 else x[2][1]) + self.cb(x)
        )
        if (feedback):
            self.set_fo([self.fo0, self.fo1, self.ifs])
        else:
            self.set_fo([self.fo0, self.fo1])
```

In [ ]:

```
class MSort(Subgraph):
    def __init__(self, I=[], N=2):
        super(MSort, self).__init__(I=I)
        self.N = N
        self.set_attributes(label='Sort.'+str(N))
        if N&(N-1) != 0:
            print('N must be a power of 2')
        elif N==4:
            self.net = Net4 (I=self.I[0])
        else:
            self.srt = MSort(I=self.I[0], N=N//2)
            self.spl = Split(I=self.srt)
            self.pss = Pass (I=[self.spl, self.spl])
            self.net = BNet4 (I=self.pss)
        Conn(self.net, self.O[0])
        self.pop()
    def init(self):
        if self.N>4:
            self.pss.I[0].init(D=0,S=self.N//4,
                x=[(0,0,0,0,-1,-self.N*2-1),(0,0,0,0,-1,-self.N*2)])
            self.pss.I[1].init(D=0,S=self.N//4,
                x=[(0,0,0,0,-1,-self.N*2-1),(0,0,0,0,-1,-self.N*2)])
            print((self.N//4, self.N//4))
```

In [ ]:

```
class MBSort(Subgraph):
    def __init__(self, I=[], N=2):
        super(MBSort, self).__init__(I=I)
        self.N = N
        self.M = 4
        self.set_attributes(label='Sort.'+str(N))
        if N&(N-1) != 0:
            print('N must be a power of 2')
        elif N <= 16:
            self.srt = MSort(I=self.I[0], N=N)
            Conn(self.srt, self.O[0])
        else:
            self.srt = MBSort(I=self.I[0], N=N//2)
            self.buf = Buffer(I=self.srt, N=N//self.M+1, W=self.M+2)
            self.pss = Pass(I=[self.buf, self.buf], feedback=True)
            self.net = BNet4(I=self.pss)
```

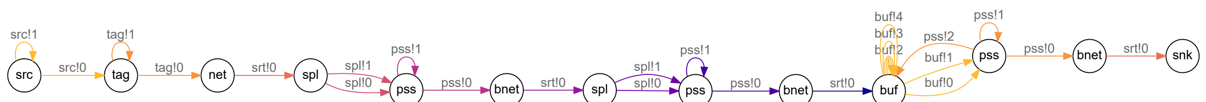
```
            Edge(self.pss.O[2], self.buf)
            Conn(self.net, self.O[0])
        self.pop()
    def init(self):
        if self.N > 16:
            self.pss.I[0].init(D=2,S=2, x=[(0,0,0,0,0,0),(0,0,0,0,0,0),
                                           (0,0,0,0,0,0),(0,0,0,0,0,0)])
            self.pss.I[1].init(D=2,S=2, x=[(0,0,0,0,0,0),(0,0,0,0,0,0),
                                           (0,0,0,0,0,0),(0,0,0,0,0,0)])
```

In [ ]:

```
M=4
N=M*8
BS=N

Gsort = Graph()
G = Gsort
G.src = SRC(f=lambda x: (
    np.int16(np.random.randint(1,99)),
    np.int16(np.random.randint(1,99)),
    np.int16(np.random.randint(1,99)),
    np.int16(np.random.randint(1,99))
))

G.tag = Node(I=G.src, fo=[lambda x: (x[0][0], x[0][1], x[0][2], x[0][3],
                                      x[1]//(BS//M), x[1]%(BS//M)), lambda x: x[1]+1])
G.tag.d = Edge(G.tag.O[1], G.tag.I[1])
G.tag.d.init(D=1, x=[0])

G.srt = MBSort(I=G.tag, N=N)

G.snk = Node(I=G.srt)
G.build()
G.plot_graph(depth=8)
```

```
(4, 4)
(2, 2)
Gsort (Graph)      : no errors
```

Out[ ]:



In [ ]:

```
np.random.seed(0)
Gsort.reset()
Gsort.view(sim=True)
Gsort.sim(T=12)
```

```
 #cycles real time   cpu time    #events   SDF=N   rate=1.000 Hz
      12      12.0s       0.1s       106   pause
```

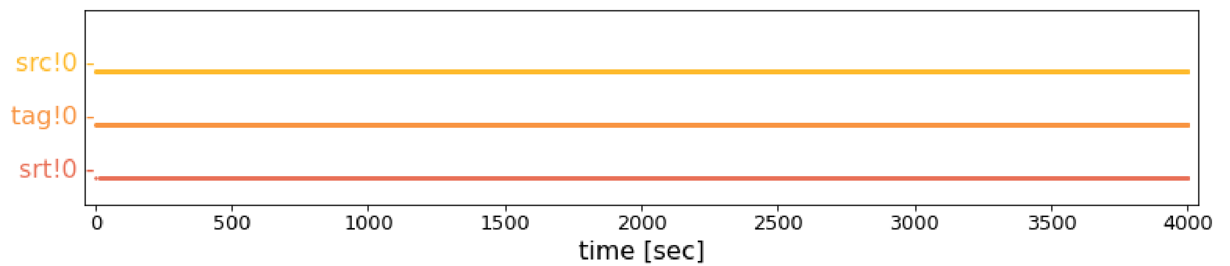
In [ ]:

```
np.random.seed(0)
Gsort.reset()
Gsort.sim(T=4000)
```

```
 #cycles real time   cpu time    #events   SDF=N   rate=1.000 Hz
    4000      4000s       5.9s     79610   pause. (13ke/cs)
```

In [ ]:

```
Gsort.plot_flow();
```



In [ ]:

```
M = 4

din = Gsort.src.O[0].data()
din  = [e for s in din for e in s]

dat  = Gsort.snk.I[0].data()
dat  = [e for s in dat for e in s[:M]]

latency = 0
for d in dat:
    if (d > 0):
        break
    latency += 1

dat = dat[latency:]
dat = dat[:BS*(len(dat)//BS)]
din = din[:len(dat)]

split = [dat[i:i+BS] for i in range(0, len(dat), BS)]
splin = [din[i:i+BS] for i in range(0, len(din), BS)]

# print(split)
errors = 0
for i in range(0, len(split)):
    if split[i] != sorted(splin[i]):
        print('Block', i, 'is not sorted')
        print('Block was', split[i])
        print('expected ', sorted(splin[i]))
        errors += 1

if errors == 0:
    print("No errors detected. Output matches sorted input. Checked",
        len(split), "blocks.")
```

```
No errors detected. Output matches sorted input. Checked 494 blocks.
```

Appendix D

Multi pass sorting code

In [ ]:

```
import sys
import matplotlib as mpl
import matplotlib.pyplot as plt
from    StaccatoLab import *
import  numpy as np
from pprint import pprint
```

In [ ]:

```
class Net4(Node):
    def __init__(self, I=[]):
        super(Net4, self).__init__(I=I)
        self.con = lambda a,b: ((a,b) if a < b else (b,a))
        self.n0 = lambda x: self.con(          x[0],           x[1] )
        self.n1 = lambda x: self.con(          x[2],           x[3] )
        self.n2 = lambda x: self.con( self.n0(x)[0], self.n1(x)[0] )
        self.n3 = lambda x: self.con( self.n0(x)[1], self.n1(x)[1] )
        self.n4 = lambda x: self.con( self.n3(x)[0], self.n2(x)[1] )

        self.set_fo(lambda x: (
            self.n2(x)[0],
            self.n4(x)[0],
            self.n4(x)[1],
            self.n3(x)[1],
            x[4],
            x[5]
        ))

class BNet4(Node):
    def __init__(self, I=[]):
        super(BNet4, self).__init__(I=I)
        self.set_attributes(label='bnet')
        self.con = lambda a,b: ((a,b) if a < b else (b,a))
        self.n0 = lambda x: self.con( x[0],          x[2]          )
        self.n1 = lambda x: self.con( x[1],          x[3]          )
        self.n2 = lambda x: self.con( self.n0(x)[0], self.n1(x)[0] )
        self.n3 = lambda x: self.con( self.n0(x)[1], self.n1(x)[1] )

        self.set_fo(lambda x: (
            self.n2(x)[0],
            self.n2(x)[1],
            self.n3(x)[0],
            self.n3(x)[1],
            x[4],
            x[5]
        ))

class Pass(Node):
    def __init__(self, I=[], feedback=False):
        super(Pass, self).__init__(I=I)
        self.M = 4
        self.d = Edge(self.O[1], self.I[2])
        self.d.init(D=1,x=[(self.M,self.M)])

        self.ia = lambda x, i: ((x[2][0]+i)//self.M, (x[2][0]+i)%self.M)
        self.ib = lambda x, i: ((x[2][1]+i)//self.M, (x[2][1]+i)%self.M)

        # Comparing (a3,a2,a1,a0) with (b3,b2,b1,b0)
        # a0 - x[0][self.ia(x,0)[0]][self.ia(x,0)[1]]
        # a1 - x[0][self.ia(x,1)[0]][self.ia(x,1)[1]]
        # a2 - x[0][self.ia(x,2)[0]][self.ia(x,2)[1]]
        # a3 - x[0][self.ia(x,3)[0]][self.ia(x,3)[1]]
        # b0 - x[1][self.ib(x,0)[0]][self.ib(x,0)[1]]
```

```
# b1 - x[1][self.ib(x,1)[0]][self.ib(x,1)[1]]
# b2 - x[1][self.ib(x,2)[0]][self.ib(x,2)[1]]
# b3 - x[1][self.ib(x,3)[0]][self.ib(x,3)[1]]

self.ca = lambda x: (
    0 if ( x[1][self.ib(x,3)[0]][4], x[1][self.ib(x,3)[0]][5],
           x[1][self.ib(x,3)[0]][self.ib(x,3)[1]]) <
         ( x[0][self.ia(x,0)[0]][4], x[0][self.ia(x,0)[0]][5],
           x[0][self.ia(x,0)[0]][self.ia(x,0)[1]]) else
    4 if ( x[0][self.ia(x,3)[0]][4], x[0][self.ia(x,3)[0]][5],
           x[0][self.ia(x,3)[0]][self.ia(x,3)[1]]) <
         ( x[1][self.ib(x,0)[0]][4], x[1][self.ib(x,0)[0]][5],
           x[1][self.ib(x,0)[0]][self.ib(x,0)[1]]) else
    1 if ( x[1][self.ib(x,2)[0]][4], x[1][self.ib(x,2)[0]][5],
           x[1][self.ib(x,2)[0]][self.ib(x,2)[1]]) <
         ( x[0][self.ia(x,1)[0]][4], x[0][self.ia(x,1)[0]][5],
           x[0][self.ia(x,1)[0]][self.ia(x,1)[1]]) else
    3 if ( x[0][self.ia(x,2)[0]][4], x[0][self.ia(x,2)[0]][5],
           x[0][self.ia(x,2)[0]][self.ia(x,2)[1]]) <
         ( x[1][self.ib(x,1)[0]][4], x[1][self.ib(x,1)[0]][5],
           x[1][self.ib(x,1)[0]][self.ib(x,1)[1]]) else
    2
)
self.cb = lambda x: self.M-self.ca(x)

cons_a  = Rule(I=[1,0,1], O=[1,1], s=0)
cons_b  = Rule(I=[0,1,1], O=[1,1], s=0)
self.set_fsm([Select([cons_a, cons_b])])

self.ifs= lambda x: (
    0 if x[2][0] >= self.M else 1
)
self.fs = self.ifs

self.fo0 = lambda x: (
    ( x[1][self.ib(x,3)[0]][self.ib(x,3)[1]],
      x[1][self.ib(x,2)[0]][self.ib(x,2)[1]],
      x[1][self.ib(x,1)[0]][self.ib(x,1)[1]],
      x[1][self.ib(x,0)[0]][self.ib(x,0)[1]],
      x[1][self.ib(x,0)[0]][4],
      x[1][self.ib(x,0)[0]][5]) if self.ca(x) == 0 else
    ( x[0][self.ia(x,0)[0]][self.ia(x,0)[1]],
      x[1][self.ib(x,2)[0]][self.ib(x,2)[1]],
      x[1][self.ib(x,1)[0]][self.ib(x,1)[1]],
      x[1][self.ib(x,0)[0]][self.ib(x,0)[1]],
      x[1][self.ib(x,0)[0]][4],
      x[1][self.ib(x,0)[0]][5]) if self.ca(x) == 1 else
    ( x[0][self.ia(x,0)[0]][self.ia(x,0)[1]],
      x[0][self.ia(x,1)[0]][self.ia(x,1)[1]],
      x[1][self.ib(x,1)[0]][self.ib(x,1)[1]],
      x[1][self.ib(x,0)[0]][self.ib(x,0)[1]],
      x[1][self.ib(x,0)[0]][4],
      x[1][self.ib(x,0)[0]][5]) if self.ca(x) == 2 else
    ( x[0][self.ia(x,0)[0]][self.ia(x,0)[1]],
      x[0][self.ia(x,1)[0]][self.ia(x,1)[1]],
      x[0][self.ia(x,2)[0]][self.ia(x,2)[1]],
      x[1][self.ib(x,0)[0]][self.ib(x,0)[1]],
      x[1][self.ib(x,0)[0]][4],
      x[1][self.ib(x,0)[0]][5]) if self.ca(x) == 3 else
    ( x[0][self.ia(x,0)[0]][self.ia(x,0)[1]],
      x[0][self.ia(x,1)[0]][self.ia(x,1)[1]],
      x[0][self.ia(x,2)[0]][self.ia(x,2)[1]],
      x[0][self.ia(x,3)[0]][self.ia(x,3)[1]],
      x[0][self.ia(x,0)[0]][4],
```

```
                x[0][self.ia(x,0)[0]][5])
        )
        self.fo1 = lambda x: (
            (x[2][0]-self.M if self.ifs(x) == 0 else x[2][0]) + self.ca(x),
            (x[2][1]-self.M if self.ifs(x) == 1 else x[2][1]) + self.cb(x)
        )
        if (feedback):
            self.set_fo([self.fo0, self.fo1, self.ifs])
        else:
            self.set_fo([self.fo0, self.fo1])
```

In [ ]:

```
class KMergeP4(Subgraph):
    def __init__(self, I=[], k=2):
        super(KMergeP4, self).__init__(I=I)
        self.k = k
        self.set_attributes(label='Mrg.'+str(k))

        assert len(I)==k
        if k&(k-1) != 0:
            print('k must be a power of 2')
        elif k<2:
            print('k must be greater than or be equal to 2')
        elif k==2:
            self.pss = Pass(I=self.I)
        else:
            il = [self.I[i] for i in range(0,k//2)]
            ir = [self.I[i] for i in range(k//2,k)]
            self.mrgl = KMergeP4(I=il, k=self.k//2)
            self.mrgr = KMergeP4(I=ir, k=self.k//2)
            self.pss = Pass(I=[self.mrgl, self.mrgr])

        self.net = BNet4(I=self.pss)
        Conn(self.net, self.O[0])
        self.pop()
    def init(self):
        self.pss.I[0].init(D=0,S=1, x=[(0,0,0,0,0,0),(0,0,0,0,0,0)])
        self.pss.I[1].init(D=0,S=1, x=[(0,0,0,0,0,0),(0,0,0,0,0,0)])
```

In [ ]:

```
class Buffer(Node):
    def __init__(self, N=2**10, W=1, I=[], T=None):
        super(Buffer, self).__init__(I=I, T=T)
        self.size = N*W
        self._graph._errors.check(self, self.size, 'size', 'scalar', 'nat')
        self._graph._errors.check(self, W,    'W',    'scalar', 'nat')
        self.W  = W
        self.ram= np.zeros(self.size, dtype=np.int16)
        self.a = Edge (self.O[2], self.I[1])
        self.a.init(D=1, x=[self.ram], incremental=True)    # initial token=RAM
        self.i = [Edge(self.O[3], self.I[2]), Edge(self.O[4], self.I[3])]
        self.i[0].init(D=N//2, S=N, x=list(range(0,N//2)))
        self.i[1].init(D=N//2+1, S=N, x=list(range(N//2,N)))

        # x[0] - Input
        # x[1] - Read/Write Address
        # x[2] - Indices A
        # x[3] - Indices B
        # x[4] - Feedback

        self.convi = lambda x: (x[0][0], x[0][1], x[0][2],
                                x[0][3], x[0][4], x[0][5]//2)
```

```
        tapa = Rule(I=[1,0,1,0,1], Of=[1,0,0,1,0], Od=[1,0,1,1,0], s=0)
        tapb = Rule(I=[1,0,0,1,1], Of=[0,1,0,1,0], Od=[0,1,1,1,0], s=0)
        tbpa = Rule(I=[1,0,1,0,1], Of=[1,0,0,0,1], Od=[1,0,1,0,1], s=0)
        tbpb = Rule(I=[1,0,0,1,1], Of=[0,1,0,0,1], Od=[0,1,1,0,1], s=0)
        absorb = Rule(I=[1,0,0,0,0], Of=[0,0,0,0,0], Od=[0,0,0,0,0], s=0)
        self.set_fsm([Select([tapa, tapb, tbpa, tbpb, absorb])])
        self.set_fs([lambda x:
            4 if x[0][0] == 0 else (
            2*((x[0][4] & 1) ^ (x[0][5] & 1)) +
            x[4]
            )
        ])

        self.fin= lambda x: x[2] if x[4] == 0 else x[3]
        self.ro = lambda x: self.fin(x)*W
        self.fh = lambda x: ((False))
        self.fh_m= 'address out of range'
        self.fo =[lambda x: tuple(x[1][x[2]*self.W:(x[2]+1)*self.W]),
                  lambda x: tuple(x[1][x[3]*self.W:(x[3]+1)*self.W]),
                  # (address, value)
                  lambda x: (self.ro(x), self.convi(x)),
                  self.fin,
                  self.fin
        ]

    def write(self, a, v, M):
        print('RAM: write: ', a, v)
        M[a:a+self.W]= v
        # print('RAM: write: ', M)
        return M
```

In [ ]:

```
class Split(Node):
    def __init__(self, I=[]):
        super(Split, self).__init__(I=I)

        take_a = Rule(I=[1], O=[1,0], s=0)
        take_b = Rule(I=[1], O=[0,1], s=0)
        self.fsm.add(0, Select([take_a, take_b]))
        # Switch A and B for each block:
        self.set_fs(lambda x: (x[4] & 1) ^ (x[5] & 1))

        fo = lambda x: (x[0], x[1], x[2], x[3], x[4], x[5]//2)
        self.set_fo([fo, fo])
```

In [ ]:

```
class MSort(Subgraph):
    def __init__(self, I=[], N=2):
        super(MSort, self).__init__(I=I)
        self.N = N
        self.set_attributes(label='Sort.'+str(N))
        if N&(N-1) != 0:
            print('N must be a power of 2')
        elif N==4:
            self.net = Net4 (I=self.I[0])
        else:
            self.srt = MSort(I=self.I[0], N=N//2)
            self.spl = Split(I=self.srt)
            self.pss = Pass (I=[self.spl, self.spl])
            self.net = BNet4 (I=self.pss)
        Conn(self.net, self.O[0])
        self.pop()
    def init(self):
```

```
        if self.N>4:
            self.pss.I[0].init(D=0,S=self.N//4,
                x=[(0,0,0,0,-1,-self.N*2-1),(0,0,0,0,-1,-self.N*2)])
            self.pss.I[1].init(D=0,S=self.N//4,
                x=[(0,0,0,0,-1,-self.N*2-1),(0,0,0,0,-1,-self.N*2)])

class MBSort(Subgraph):
    def __init__(self, I=[], N=2):
        super(MBSort, self).__init__(I=I)
        self.N = N
        self.M = 4
        self.set_attributes(label='Sort.'+str(N))
        if N&(N-1) != 0:
            print('N must be a power of 2')
        elif N <= 16:
            self.srt = MSort(I=self.I[0], N=N)
            Conn(self.srt, self.O[0])
        else:
            self.srt = MBSort(I=self.I[0], N=N//2)
            self.buf = Buffer(I=self.srt, N=N//self.M+1, W=self.M+2)
            self.pss = Pass(I=[self.buf, self.buf], feedback=True)
            self.net = BNet4(I=self.pss)
            Edge(self.pss.O[2], self.buf)
            Conn(self.net, self.O[0])
        self.pop()
    def init(self):
        if self.N > 16:
            self.pss.I[0].init(D=2,S=2, x=[(0,0,0,0,0,0),(0,0,0,0,0,0),
                                           (0,0,0,0,0,0),(0,0,0,0,0,0)])
            self.pss.I[1].init(D=2,S=2, x=[(0,0,0,0,0,0),(0,0,0,0,0,0),
                                           (0,0,0,0,0,0),(0,0,0,0,0,0)])
```

In [ ]:

```
class Signal(Node):
    def __init__(self, I=[]):
        super(Signal, self).__init__(I=I)
        self.set_fsm([Choice([
            Rule(I=[1], O=[1,1], s=0),
            Rule(I=[0], O=[0,1], s=0),
        ], round_robin=False)])
        self.set_fo([lambda x: x, lambda x, r: 1-r])

class Count(Node):
    def __init__(self, S, I=[]):
        super(Count, self).__init__(I=I)
        self.sig = S
        self.s=Edge(self.O[1], self.I[2])
        self.set_fo([lambda x: 1, lambda x: x[2]+x[0]-x[1]])

        r_empty = Rule(I=(1,1,1), O=(1,1))
        r_full  = Rule(I=(1,1,1), O=(0,1))
        self.set_fsm([Select([r_full, r_empty])])
        self.fs = lambda x: x[2] <= self.sig

    def init(self):
        self.s.init(D=1, x=[0])

class ResetEvent(Node):
    def __init__(self, C, R, I=[]):
        super(ResetEvent, self).__init__(I=I)
        self.C = C
        self.R = R
        Edge(self.O[1], self.I[1])
        self.set_fsm([
```

```
            Select([
                Rule(I=[1,1], O=[0,1]),
                Rule(I=[1,1], O=[1,1]),
            ])
        ])
        self.fs = lambda x: x[1]>=self.C
        self.fo = [
            lambda x: 1,
            lambda x, r: x[0] if r else (x[1]+x[0])
        ]
    def init(self):
        self.I[1].init(D=1, x=self.C-self.R)

class EmptyEvent(Node):
    def __init__(self, I=[]):
        super(EmptyEvent, self).__init__(I=I)
        # I = [Buffer empty, Finished filling event]
        # O = [Empty event]

        self.set_fsm([
            Choice([ # Full state
                Rule(I=(1,1), O=(1,), s=1), # Send empty signal
                Rule(I=(1,0), O=(1,), s=1), # Send empty signal
                Rule(I=(0,1), O=(0,), s=0), # Still full, consume done filling event
            ], round_robin=False),
            Choice([ # Empty state
                Rule(I=(1,1), O=(1,), s=1), # Still empty, send empty signal
                Rule(I=(1,0), O=(0,), s=1), # Do nothing
                Rule(I=(0,1), O=(0,), s=0), # Now full
            ], round_robin=False)
        ])
        self.set_fo([lambda x: 1])

class CBuffer(Subgraph):
    def __init__(self, N, C, R, I=[], S=0):
        super(CBuffer, self).__init__(I=I)
        self.N = N
        self.sig = N//2 if S == 0 else S

        self.first  = Signal(I=self.I[0])
        self.second = Signal(I=self.first.O[0])
        self.sdel   = Node(I=self.second.O[1])
        self.spl    = Node(I=self.first.O[1])
        self.count  = Count(I=[self.spl, self.sdel], S=self.sig)
        self.revt   = ResetEvent(I=self.spl, C=C, R=R)
        self.evt    = EmptyEvent(I=[self.count, self.revt])
        Conn(self.second.O[0], self.O[0])
        Conn(self.evt, self.O[1])
    def init(self):
        self.second.I[0].init(D=0, S=self.N)
```

In [ ]:

```
class BSwitch(Node):
    def __init__(self, I=[], C=4):
        super(BSwitch, self).__init__(I=I)
        self.c = C

        self.set_fsm([
            Choice([
                Rule(I=(1,0), O=(1,0,0), s=1),
                Rule(I=(0,1), O=(0,1,1), s=2)
            ], round_robin=False),
            Repeat(C-1, iter=Rule(I=(1,0), O=(1,0,0), s=1),
                        exit=Rule(I=(1,0), O=(1,0,0), s=0)),
```

```
            Repeat(C-1, iter=Rule(I=(0,1), O=(0,1,1), s=2),
                        exit=Rule(I=(0,1), O=(0,1,1), s=0))
        ])

        self.fo=[
            lambda x: x[0],
            lambda x: x[1][0],
            lambda x: x[1][1]
        ]
```

In [ ]:

```
class MinSel(Node):
    def __init__(self, I=[]):
        super(MinSel, self).__init__(I=I)
        self.k = len(I)
        Edge(self.O[1], self.I[self.k])

        self.set_fsm([

        ])

        self.fo = [
            lambda x: x[self.k][2],
            lambda x: (
    (x[self.k][0]+1)%self.k,
        (x[0],0) if x[self.k][0] == 0 else
        (x[x[self.k][0]], x[self.k][0]) if x[x[self.k][0]] < x[self.k][1][0] else
        x[self.k][1],
    x[self.k][1][1] if x[self.k][0] == 0 else x[self.k][2]
            )
        ]
    def init(self):
        self.O[1].init(D=1, x=(0,(0,0),0))

class AddrShift(Node):
    def __init__(self, N, I=[]):
        super(AddrShift, self).__init__(I=I)
        self.k = len(I)
        rules = []
        for i in range(0,k):
            r_in = [0,]*self.k
            r_in[i] = 1
            rules += [Rule(I=r_in, O=(1,))]
        self.set_fsm([Choice(rules)])
        self.fo=lambda x,r: (r*N+x[r][0], x[r][1])

class ReadBK(Node):
    def __init__(self, N, M, C, I=[]):
        super(ReadBK, self).__init__(I=I)
        self.offset = C*M
        self.pages = N//M//C
        Edge(self.O[2], self.I[2])

        # I = [evt,  w_fin_evt, self_bk]
        # O = [load, full_evt,  self_bk]

        self.set_fsm([
            Select([ # All pages read
                Rule(I=(1,1,1), O=(1,0,1), s=1)
            ]),
            Select([ # Ready to read
                Rule(I=(1,0,1), O=(1,0,1), s=1),
                Rule(I=(1,0,1), O=(1,1,1), s=0),
            ])
```

```
        ])
        self.fs = [lambda x: 0, lambda x: x[2][0] == self.pages-1]

        self.fo = [
            lambda x: (x[2][0]*self.offset, x[2][1]),
            lambda x: 1,
            lambda x,r: (0 if r == 1 else x[2][0]+1, x[2][1]+r)
        ]
    def init(self):
        self.I[2].init(D=1, x=[(0,1)])
```

In [ ]:

```
class WriteController(Node):
    def __init__(self, N, M, L, I=[]):
        super(WriteController, self).__init__(I=I)
        Edge(self.O[2], self.I[2])
        Edge(self.O[3], self.I[3])

        # I = [data, empty block, self, self fsmcnt]
        # O = [data, full  block, self, self fsmcnt]

        self.set_fsm([
            Select([
                Rule(I=(1,1,1,1), O=(1,0,1,1)),
                Rule(I=(1,0,0,1), O=(1,0,0,1)),
                Rule(I=(1,0,0,1), O=(1,1,0,1))
            ])
        ])
        self.fs = lambda x: 0 if x[3] == 0 else 2 if x[3] == N//M-L else 1

        self.fo = [
            lambda x: (N*(x[1] if x[3]==0 else x[2])+M*x[3],
                       (x[0][0],x[0][1],x[0][2],x[0][3])),
            lambda x: x[2],
            lambda x: x[1],
            lambda x: (x[3]+1) % (N//M),
        ]

    def init(self):
        self.O[2].init(D=1, x=0)
        self.O[3].init(D=1, x=0)

class ReadDistribution(Node):
    def __init__(self, k, I=[]):
        super(ReadDistribution, self).__init__(I=I)
        rules = []
        for i in range(0, k):
            r_o = [0]*k
            r_o[i] = 1
            rules += [Rule(I=[1,1], O=r_o)]
        self.set_fsm([Select(rules)])
        self.fs = [lambda x: x[1][0]]
        self.fo = [lambda x: (x[0][0], x[0][1], x[0][2], x[0][3], x[1][1], 0)]*k

class EventDistribution(Node):
    def __init__(self, k, I=[]):
        super(EventDistribution, self).__init__(I=I)
        self.set_fsm([
            Repeat(k, iter=Rule(I=[1], O=[0]*k), exit=Rule(I=[1], O=[1]*k, s=0))
        ])

class EventMerge(Node):
    def __init__(self, I=[], k=0):
        super(EventMerge, self).__init__(I=I)
```

```python
        self.k = len(I) if k == 0 else k
        rules = []
        for i in range(0,k):
            r_in = [0,]*self.k
            r_in[i] = 1
            rules += [Rule(I=r_in, O=(1,))]
        self.set_fsm([Choice(rules, round_robin=False)])
        self.fo=lambda x,r: r
```

In [ ]:

```python
# Buffer size calculation
def b(K, C, L):
    if K == 2:
        return 2*(C+L)
    else:
        return K*(b(K-1,C,L)+C+L)/(K-1)

def bs(K,C,L):
    return b(K,C,L)/K+C+L
```

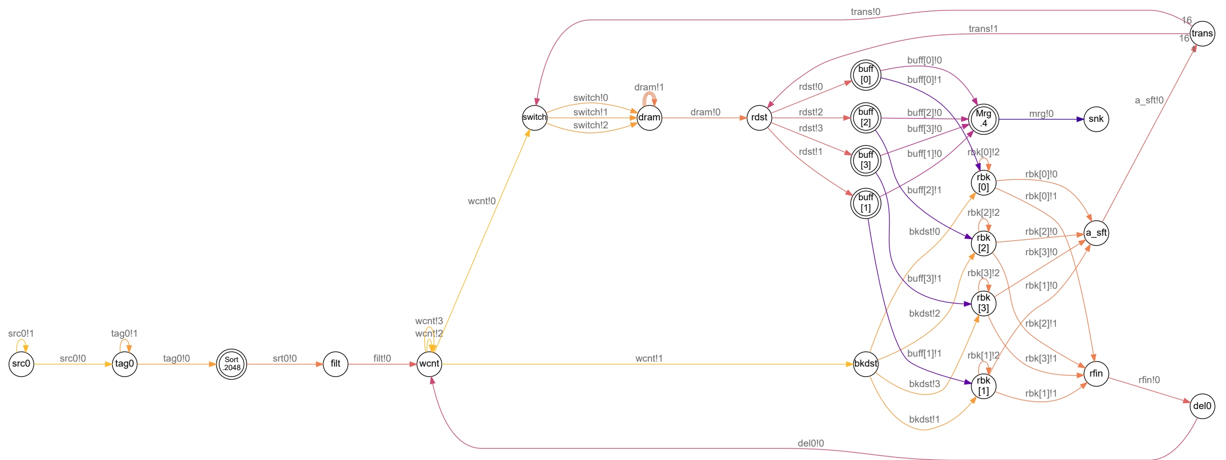
In [ ]:

```python
C=16    # Page block size
k=4    # Merge tree width
M=4     # Sorter parallelism
N=M*512 # Maximum streaming sort size
BS=N    # Block size <= N

Gsort = Graph()
G = Gsort
G.src0 = SRC(f=lambda x: (
    np.int16(np.random.randint(1,999)),
    np.int16(np.random.randint(1,999)),
    np.int16(np.random.randint(1,999)),
    np.int16(np.random.randint(1,999))
))

G.tag0 = Node(I=G.src0, fo=[
    lambda x: (x[0][0], x[0][1], x[0][2], x[0][3], x[1]//(BS//M), x[1]%(BS//M)),
    lambda x: x[1]+1])
G.tag0.d = Edge(G.tag0.O[1], G.tag0.I[1])
G.tag0.d.init(D=1, x=[0])

G.srt0 = MBSort(I=G.tag0, N=N)

G.filt = Node(I=G.srt0)
G.filt.set_fsm([Select([
    Rule(I=(1,), O=(1,)),
    Rule(I=(1,), O=(0,))
    ])
])
G.filt.fs = lambda x: x[0] == 0

G.rfin = EventMerge(k=k)
G.del0 = Node(I=G.rfin) # add 1 cycle delay to this signal
G.wcnt = WriteController(I=[G.filt, G.del0], N=N, M=M, L=4)
G.wcnt.I[1].init(D=k, x=list(range(0,k)))

G.trans = LM(L=C, fo=[lambda x, r: x[0]+r*M, lambda x: (x[0]//N, x[1])])

G.switch = BSwitch(I=[G.trans,G.wcnt], C=N//M*k)

G.dram = RAM(I=[G.switch, G.switch, G.switch], size=N*k, W=M)
```

```python
G.rdst = ReadDistribution(I=[G.dram, G.trans], k=k)
G.rdst.I[1].init(S=10)

latency = 8
G.buff = [CBuffer(I=G.rdst, N=math.ceil(bs(k,C,latency)),
            C=C, R=C-latency, S=C-1) for i in range(0,k)]
G.mrg = KMergeP4(I=G.buff, k=k)

G.bkdst  = EventDistribution(I=G.wcnt.O[1], k=k)
G.rbk = [ReadBK(I=[G.buff[i], G.bkdst], N=N, M=M, C=C) for i in range(0,k)]

G.a_sft = AddrShift(I=G.rbk, N=N)
Edge(G.a_sft, G.trans, S=k-1)

[Edge(G.rbk[i], G.rfin) for i in range (0,k)]

G.snk = Node(I=G.mrg)

G.build()
G.plot_graph()
```

```
Gsort (Graph)      : no errors
```

Out[ ]:



In [ ]:

```python
np.random.seed(0)
G.reset()
G.view(sim=True)
# G.sim(T=220)
G.sim(T=600)
# G.sim(T=1000)
# G.sim(T=3890)
# G.sim(T=16050)
```

```
#cycles real time  cpu time   #events  SDF=N  rate=1.000 Hz
    600      600s      3.3s     47553  pause (14ke/cs)
```

In [ ]:

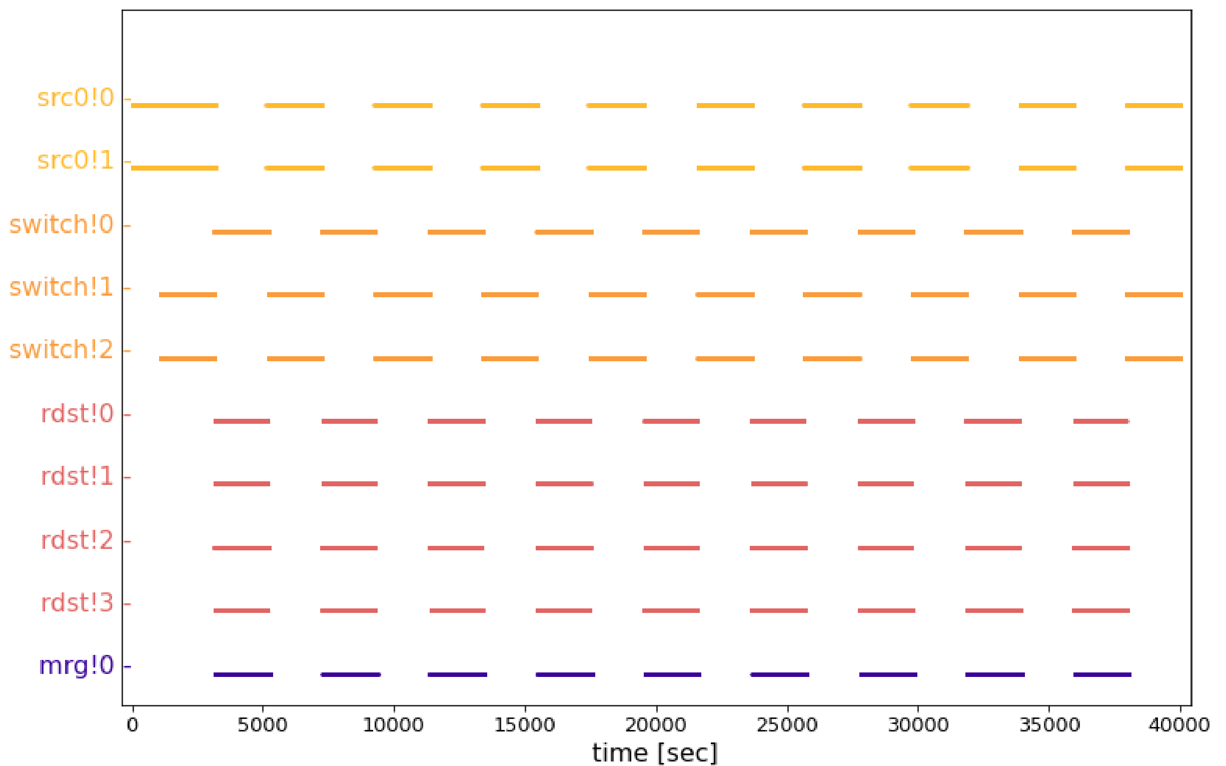
```python
T=40000
G.sim(T=T)
```

```
#cycles real time  cpu time   #events  SDF=N  rate=1.000 Hz
   2400     2400s     13.6s    223978
   4000     4000s     24.1s    336269
   6100     6100s     34.1s    479351
   8100     8100s     44.5s    631305
  10300    10300s     55.0s    783919
  12100    12100s     65.2s    922245
  14300    14300s     75.5s   1070068
```

```
16300     16300s     85.7s   1222620
18400     18400s     95.8s   1366206
20300     20300s    105.9s   1514181
22500     22500s    116.0s   1662267
24700     24700s    126.2s   1823093
26800     26800s    136.3s   1977562
29000     29000s    146.6s   2127322
31000     31000s    156.6s   2283033
33200     33200s    166.8s   2426855
35200     35200s    177.2s   2587838
37400     37400s    187.4s   2726828
39400     39400s    197.8s   2893635
40000     40000s    203.9s   2953569  pause (14ke/cs)
```

In [ ]:

```
G.plot_flow(Edges=[Gsort.src0.O, Gsort.switch.O, Gsort.rdst.O, Gsort.snk.I]);
```



In [ ]:

```
din = Gsort.src0.O[0].data()
din  = [e for s in din for e in s]

dat  = Gsort.snk.I[0].data()
dat  = [e for s in dat for e in s[:M]]

latency = 0
for d in dat:
    if (d > 0):
        break
    latency += 1

dat = dat[latency:]
dat = dat[:k*BS*(len(dat)//(k*BS))]
din = din[:len(dat)]

split = [dat[i:i+BS*k] for i in range(0, len(dat), k*BS)]
splin = [din[i:i+BS*k] for i in range(0, len(din), k*BS)]

# print(split)
errors = 0
for i in range(0, len(split)):
```

```
    if split[i] != sorted(splin[i]):
        print('Block', i, 'is not sorted')
        print('Block was', split[i])
        print('expected ', sorted(splin[i]))
        errors += 1

if errors == 0:
    print("No errors detected. Output matches sorted input. Checked",
        len(split), "blocks.")
```

```
No errors detected. Output matches sorted input. Checked 8 blocks.
```

In [ ]:

```
max_lines = 1
buff_fullness = [Gsort.buff[i].count.O[1].data() for i in range(0,k)]
print([max(buff_fullness[i]) for i in range(0,k)])
[plt.plot(buff_fullness[i][9000:18000]) for i in range(0,min(k,max_lines))];
```

```
[29, 27, 27, 30]
```