

Imperial College London
Department of Computing

Reconfigurable Acceleration of Big Data Analytics

Philippos Papaphilippou

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London and
the Diploma of Imperial College London, 2021

Declaration of Originality

I Philippos Papaphilippou of Imperial College London hereby declare that this thesis and all presented material is my own work, except where specified explicitly. It does not contain material that has already been used for obtaining another degree. Due to the relatively long history and great breadth of research in computer science and engineering, it could be possible to find unintentional strong similarities with previous works, though I am not aware of such instances to the best of my knowledge.

Copyright

For the avoidance of doubt the rights of dunnhumby limited (and of its parent and group companies) relating to this thesis are set out under the Agreement between dunnhumby Limited, Imperial College of Science Technology and Medicine and Philippos Papaphilippou. The following statement shall not apply to dunnhumby limited or to any parent or group company of dunnhumby limited.

Unless otherwise indicated, its contents are also licensed under a Creative Commons Attribution-NonCommercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose, unless there is an agreement with Imperial College which specifies otherwise.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Acknowledgements

I would like to express my gratitude towards my supervisor Prof. Wayne Luk who gave me the opportunity to do research on topics I found interesting, and for his enthusiasm about my work. The support of dunnhumby through the industrial PhD studentship is gratefully acknowledged, as well as Microsoft through a remote internship. I would like to thank Chris Brooks and Rosie Prior from dunnhumby for their involvement in the academic partnership program. For example, many of the presented ideas were influenced by our weekly meetings with Chris. I would also like to thank Prof. Paul H. J. Kelly, and Dr. Holger Pirk for our collaborations, as well as their advice on the PhD in general. The valuable comments from the Viva with the examiners Prof. Dirk Koch and Dr. Thomas Heinis are greatly appreciated. Finally, I would like to thank my family for their continuous support in pursuing my goals.

Abstract

The amount of data stored and processed in data centers is growing at an unprecedented rate. At the same time, the improvement in processing capabilities of central processing units (CPUs) has relatively stagnated over the last decade, creating an increasing demand for specialised processing. Specialised accelerators have gotten the spotlight for computationally intensive applications, such as with deep learning on graphics processing units (GPUs), though such special purpose processors tend to be optimised for the trending applications and are not efficient for each entity's computational needs. On the other hand, reconfigurable computing has shown impressive potential in accelerating specialised tasks, including database applications. Hence, field-programmable gate arrays (FPGAs) have started evolving into an integral part in the data center. However, the heterogeneity found in today's systems featuring FPGAs, such as through non-uniform memory accesses (NUMA), has complicated the deployment and development of database accelerators.

This PhD introduces novel parallel algorithms and FPGA designs for database acceleration that take into consideration the inter-chip communication limitations. The presented designs accelerate fundamental database operators such as sorting, sort-merge join and distinct count, with notable advantages over the state-of-the-art. Additionally, some building blocks such as the parallel round-robin arbiter and the fast lightweight merge sorter (FLiMS) are shown to have a wider applicability, including in single-instruction multiple-data (SIMD) algorithms and network switches. The proposed designs operate in a streaming access pattern with a wide path in order to achieve scalability to input size and future high-bandwidth architectures. Finally, a discussion on future architectures with reconfigurable instructions is provided as future work to further address the challenges appearing when accelerating big data using today's FPGAs.

Glossary

- ASIC** Application-Specific Integrated Circuit.
- ASSP** Application-Specific Standard Product.
- AVX** Advanced Vector Extensions.
- AXI** Advanced eXtensible Interface.
- BRAM** Block Random-Access Memory.
- CAS** Compare-and-Swap.
- CGRA** Coarse-Grained Reconfigurable Arrays.
- CPRRA** Combined Parallel Round-Robin Arbiter.
- CPU** Central Processing Unit.
- DBMS** Database Management System.
- DDR** Double Data Rate.
- DIMM** Dual In-line Memory Module.
- DMA** Direct Memory Access.
- DRAM** Dynamic Random-Access Memory.
- DSP** Digital Signal Processing block.
- FF** Flip-Flop register.
- FIFO** First-In First-Out queue.
- FLiMS** Fast Lightweight Merge Sorter.
- FPGA** Field-Programmable Gate Array.
- GCC** the GNU Compiler Collection.
- GPU** Graphics Processing Unit.
- HDL** Hardware Description Language.

HLS High-Level Synthesis.

HOL Head-Of-Line blocking.

HPC High-Performance Computing.

IQ Input-Queued.

ISA Instruction Set Architecture.

LLC Last-Level Cache.

LSB Least Significant Bit.

LUT Lookup Table.

MPSoC Multi-Processor System on a Chip.

MSB Most Significant Bit.

MUX Multiplexer.

NUMA Non-Uniform Memory Access.

OLAP Online Analytical Processing.

OLTP Online Transaction Processing.

OQ Output-Queued.

PCI Peripheral Component Interconnect.

PCIe Peripheral Component Interconnect Express.

PMT Parallel Merge Tree.

PRRA Parallel Round-Robin Arbiter.

RISC Reduced Instruction Set Computer.

RTL Register-Transfer Level/Language.

SIMD Single Instruction, Multiple Data.

SLR Super Logic Region.

SoC System on a Chip.

SQL Structured Query Language.

SSI Stacked Silicon Interconnect.

VLEN Vector register LENgth.

VOQ Virtual Output Queues.

WNS Worst Negative Slack.

XLEN Register LENgth.

Contents

Declaration of originality	i
Copyright	iii
Acknowledgements	v
Abstract	vii
Glossary	viii
1 Introduction	2
1.1 Motivation	5
1.2 Research challenges	8
C1 Inter-chip communication	8
C2 Development	9
C3 Applicability and portability	10
C4 Scalability	12
1.3 Publications and contributions	13
2 Background	18
2.1 Classification	19
2.2 Database acceleration frameworks	20
2.2.1 Frameworks directly related to databases	21
2.2.2 General-purpose frameworks	22
2.3 Sorting accelerators	24

2.3.1	Small-scale sorters	24
2.3.2	High-throughput merge sorters	26
2.3.3	High-throughput sorter building blocks (2-way mergers)	27
2.3.4	Many-leaf merge sorters	30
2.3.5	Non-merge-focused sorters	31
2.4	Database-specific operator accelerators	33
2.4.1	Joins	33
2.4.2	String matching	35
2.4.3	Filtering	35
2.4.4	Distinct count	37
2.4.5	Arithmetic	38
2.5	Alternative and future architectures	39
2.5.1	Other co-processors	39
2.5.2	High-performance multi-core processors	40
2.5.3	Many-core processors	41
2.5.4	Faster interconnection protocols	43
2.5.5	Near-data processing/ processing in memory	45
2.6	Summary	45
3	FLiMS: a fast lightweight 2-way merger	47
3.1	A novel 2-way high-throughput merger	49
3.2	Additional functionality	52
3.2.1	Skewness optimisation	52
3.2.2	Stable merge	54
3.3	Correctness	55
3.3.1	Basic version	55
3.3.2	Including the skewness optimisation	57
3.3.3	Including the stable merge functionality	59
3.4	Comparison of FLiMS with the related work	60

3.5	FPGA implementation	64
3.6	Software implementation using SIMD intrinsics	68
3.6.1	Merge function implementation	69
3.6.2	Sort-in-chunks function implementation	71
3.6.3	Complete sorting function evaluation	71
3.7	Summary	74
4	Parallel round-robin arbiter	75
4.1	Pipelined building blocks	78
4.2	The parallel round-robin arbiter	79
4.2.1	High-level functionality	80
4.2.2	Proposed implementation	81
4.2.3	Adopting for data analytics	82
4.3	The combined parallel round-robin arbiter	83
4.3.1	Background	84
4.3.2	High-level functionality	86
4.3.3	Proposed implementation	87
4.4	Properties and optimisation	90
4.4.1	Non-blocking	91
4.4.2	Scalability	92
4.4.3	Complexities	93
4.4.4	Reducing the pipeline latency	94
4.4.5	Calculating the optimised latency	96
4.4.6	Optimality	97
4.5	Evaluation	99
4.5.1	As a switching algorithm	100
4.5.2	As an FPGA implementation	104
4.6	Summary	108

5	Accelerating database operators using FPGAs	110
5.1	An adaptable high-throughput FPGA merge sorter	111
5.1.1	Solution	113
5.1.2	Optimisations	117
5.1.3	Evaluation	118
5.1.4	Properties and related work	121
5.2	Accelerating the merge phase of sort-merge join	123
5.2.1	Solution	125
5.2.2	Evaluation	130
5.2.3	Properties and related work	132
5.3	Accelerating distinct count	135
5.3.1	Distinct count of keys with a restricted range	135
5.3.2	Distinct count of keys with a wider range	139
5.4	Accelerating distinct count with groups	140
5.4.1	Design	140
5.4.2	Evaluation	142
5.4.3	Comparison to related work	143
5.5	Summary	144
6	Conclusion	146
6.1	Summary of contributions	147
6.2	Future work	148
6.2.1	Further optimisation	148
6.2.2	Optimising the parallel round-robin arbiter	149
6.2.3	Integration to real database management systems (DBMS)	153
6.2.4	A fully-streaming merge sorter on datacenter FPGAs	153
6.2.5	Supporting reconfigurable SIMD instructions in high-end processors . . .	154
6.2.6	FPGA-friendly cache subsystem	156
6.2.7	ASIC equivalents and their placement in systems	157
6.3	Concluding remarks	157

Bibliography	158
Appendices	181
A FLiMS-based sorting function using SIMD intrinsics and optimisation	182
A.1 Merge function fine-tuning	182
A.2 Merge function implementation using AVX-512	184
A.3 Sorting in chunks using bitonic sort	185
A.4 Comparing sorting performance on different micro-architectures	187
B FPGA-based network switch implementations dataset	189
C Exploring a possible future path for CPUs to accelerate big data analytics	192
C.1 Simodense microarchitecture	193
C.1.1 Cache hierarchy optimisations	194
C.1.2 Main core	198
C.2 Custom SIMD instructions	199
C.2.1 Vector instruction types	199
C.2.2 Instruction templates	201
C.3 Evaluation	202
C.3.1 Design space exploration	202
C.3.2 Performance as a RV32IM core	204
C.3.3 Custom SIMD instruction use cases	205
C.4 Comparison to related work	209
C.5 Summary	210

List of Tables

3.1	Merging two descending sequences: Example execution for $w=4$ and random sorted lists A and B. (see [133] for an online visualisation)	52
3.2	Comparing high-throughput 2-way mergers	61
4.1	Indicative pipeline latency after register reduction	97
5.1	Results summary for full system sorting designs for 32-bit sorting	120
5.2	Comparison with various small-scale sorters, inspired by [167], [168]	122
5.3	Results summary for full system 64-bit sorting with distinct counter	143
B.1	Synthesis and implementation dataset, based on reports from Vivado 2020.1	190
C.1	Selected configuration of Simodense	203
C.2	Indicative comparison ignoring SIMD	204

List of Figures

1.1	An abstracted view a representative database schema	6
1.2	Analysing an example dataset from dunnhumby	7
2.1	Main classification of the related research work	20
2.2	Three sorting network examples for 8 inputs	25
2.3	Linear sorter example	26
2.4	Parallel merge tree (PMT [69]), for 8 input lists	27
2.5	“Unoptimised” parallel merge tree, for 8 input lists	27
2.6	A 16-to-8 bitonic partial merger block as found in a bitonic sorter for a list of 16 elements	28
2.7	High-level view of the merger used in PMT [69]	29
2.8	High-level view of the merger used in MMS [35]	30
3.1	High-level architecture of FLiMS [36]	49
3.2	FLiMS in low-level: <i>MAX</i> selector stage plus a <i>CAS</i> network (butterfly topol- ogy), $w = 4$. Pruned CAS units are shown in grey	49
3.3	WMS [74] architecture (top) and CAS network (bottom)	62
3.4	FPGA resource utilisation comparison of FLiMS with the latest mergers	65
3.5	Maximal operating frequency for FLiMS, WMS and EHMS	65
3.6	Aggregate throughput of the solutions in the design space	66
3.7	Results for AVX2-based FLiMS for merging 2 lists of 2^{24} 32-bit integers	70
3.8	Evaluating the FLiMS-based SIMD complete sort C++ functions	72
4.1	Pipelined versions of the building blocks for 8 inputs	79

4.2	Parallel round robin arbiter visualisation	80
4.3	Parallel round robin arbiter waveform	80
4.4	A pipelined implementation for the parallel round-robin arbiter, for $P=4$	81
4.5	A <i>stable</i> parallel round-robin arbiter, for $P=4$	83
4.6	An input-queued switch showing two limitations	85
4.7	Abstracted view of the proposed network switch, $P=4$	86
4.8	Unoptimised network switch based on the parallel round-robin arbiter, $P=4$. . .	88
4.9	The proposed high-performance FPGA network switch	89
4.10	Dividing the latency of a sorter by 3, before (left) and after (right)	95
4.11	A uniform bursty traffic Markov model	101
4.12	Average packet latency using different algorithms and traffic models (16×16 switch, simulation)	102
4.13	Average worst-case packet latency using different algorithms and traffic models (16×16 switch, simulation)	103
4.14	Impact of queue size, for the proposal and the best alternative per traffic model (16×16 switch)	104
4.15	Flip-flop register (FF) and lookup table (LUT) utilisation	106
4.16	Performance for switch designs with 256-bit (left) and 512-bit (right) packets .	106
4.17	Comparison of 16×16 switches implementations on Xilinx Alveo U280	108
5.1	Data movement during the three sorting phases	113
5.2	Merge-capable linear sorter example	114
5.3	The merge-capable high-throughput sorter (for a throughput of 4). Wire widths are multiples of the value width.	115
5.4	Sorting performance	120
5.5	High-level view of the merge-phase accelerator	125
5.6	Building blocks of the proposed solution	126
5.7	Co-grouping engine	127
5.8	Complementary figures for algorithms 5 (left) and 6 (right)	127
5.9	Join module, for $w = 8$	129
5.10	FPGA speedup for different key distributions	132

5.11	High-throughput design of distinct count on FPGAs, $w = 4$	137
5.12	High-throughput distinct counter performance for different inputs	138
5.13	Distinct counter engine	141
5.14	Distinct count performance	142
6.1	Butterfly network (a) as found in a bitonic sorter (b), and reverse butterfly (c) .	150
6.2	Reverse butterfly working as (a) a rotator, (b) a parallel round-robin arbiter . .	151
6.3	Proposed arrangement for fully-streaming merge sorting	154
6.4	Reconfigurable instructions in CPUs: example data movement	155
A.1	Vectorising the CAS network of FLiMS using min, max and shuffle AVX2 intrinsics	183
A.2	Finding the optimal w value for AVX2-based FLiMS 2-way merger	184
A.3	AVX2 vs AVX-512 FLiMS performance comparison on Xeon 8124M	185
A.4	Finding the optimal chunk size for the sort-in-chunks loop that uses an AVX2- based bitonic sorter	186
A.5	Comparison of FLiMS-based sorting and alternatives on Intel CPUs of different generations	187
C.1	Data movement in an example configuration	194
C.2	Last-level cache block RAM organisation	197
C.3	Two variations of the I and S instruction types	200
C.4	Memcpy() read and write (bidirectional) throughput for different vector register widths (left) and last-level cache block sizes (right)	203
C.5	Adapted STREAM benchmark results, no SIMD	204
C.6	Two new custom instructions based on odd-even mergesort	206
C.7	Instruction execution times from the loop for sorting in chunks	206
C.8	New SIMD instruction for prefix sum	208

Chapter 1

Introduction

General purpose computing has proven to be a powerful tool capable of solving most algorithmic problems. Most systems today feature a general purpose processor (CPU) that is connected to a main memory and other components, and use the von Neumann architecture model. More specifically, they use a “modified” Harvard architecture, where the main memory is shared for both instructions and data, but there is a distinction at the lower levels of the cache hierarchy. More abstracted models could also be used to describe today’s systems, including Turing machines (1936) [1] for instruction-based CPUs, and input-output automata in distributed systems (1987) [2] for multi-processors.

However, these models have their limitations. The performance overhead is more apparent in highly-parallelised tasks, such as when comparing the CPU performance to specialised accelerators. Today, specialised accelerators include implementations using graphics processing units (GPUs), field-programmable gate arrays (FPGAs) and application-specific integrated circuits (ASICs). Each one of these technologies have their own advantages and disadvantages, such as different deployment costs according to how many chips need to be used [3].

One of the main ideas behind using accelerators is to reduce or eliminate the instruction overhead required to perform a special task. By pre-defining the logic required to solve a problem, such as by using dataflow techniques on FPGAs [4], any instruction-related complications such as branch misprediction is eliminated. By removing the instructions, there is also a consider-

able amount of relevant circuits that can be removed, including instruction caches and adaptive techniques (instruction pipelines, prefetching, bypassing, wrong path execution etc.). In practice, the removal of the unused logic, including the instruction-related, leaves room for more specialised computational units for highly-parallelisable tasks. Even on modern CPUs, a related “workaround” that exploits data-level parallelism (DLP) is the use of single-instruction multiple-data (SIMD) instructions [5] or, more-generally, vector instructions [6] that increase the number of operations performed by a single instruction call.

FPGAs have appeared in the datacenter [7] and in the cloud [8]. Eventually, the range of applications for FPGAs has widened not only to include prototyping, but to also facilitate embedded accelerators [9], as well as big data acceleration in the datacenter [10]. Virtualised environments provided by cloud computing vendors, such as Amazon AWS, Microsoft Azure and Alibaba, have started offering FPGA in certain types of virtual instances (i.e. cloud-based virtual machines, usually priced based on usage, time and resources, one of them being FPGA(s)). They have been successfully utilised for streaming applications, as well as for other scientific applications [11], as a rival to GPU-based acceleration. This was possible due to the promotion and commercialisation of advanced FPGAs, in combination with the use of high-level synthesis (HLS) frameworks to ease hardware development for software developers. There are also “app stores”, such as with subscription-based Amazon AWS EC2 Linux images, to accelerate specialised workloads without the need for the user to know details about the low-level implementation. However, there are still important challenges when considering FPGA-based solutions for big data analytics.

In this thesis the term “big data” is used interchangeably with databases, as the research on database acceleration on FPGAs is usually for analytics on large-scale databases. Database operations are also similarly as diverse as any computer program, and their technical context is overlapping with that of big data. In literature there is no convention on defining big data, and the work of De Mauro et al. has identified a list of main aspects commonly associated with defining “big data” [12]. First are the *attributes of data*, including size and access patterns. Additionally, there are definitions that focus on the *technological advances* needed to process such data, and *crossing thresholds*. Finally, there is the *social impact* of big data, by

creating valuable information from processing them. In this case, FPGAs in the datacenter are the technological advancement needed for such problems, crossing performance and input size thresholds for data with special attributes. It also has a potential for impacting society, as described in the motivation section 1.1.

The scale of the data or the problem size could be the number one factor of how efficiently an arbitrary computing system can process the data. In the smallest scale, such as in silicon with standard cells (gates and registers), one of the most complex operations is any fully-interconnected logic, where the number of wires grows quadratically with the number of entities. An example of this is found in network switches and many-core interconnects, where a hierarchical approach may be necessary for scalability to a two-digit number of entities [13]. The memory hierarchies in computers, such as with a main memory, low-latency private caches and higher-latency last-level caches, aim for a smooth transition between problem sizes that fit on-chip to those that do not [14]. In the largest scale, main memory was traditionally abstracted as of (near) infinite size, for future-proofing the CPU designs and algorithms. This abstraction also helped distinguishing spatial [15] and temporal [16] localities for optimisation in both software and hardware. Any data outside the main memory, such as from storage, the internet and in distributed systems, can also be abstracted through memory for fine-tuning a processing node, though more specialised models exist [17].

Abstracting the problem size by default, however, has not been the case with most FPGA designs. In contrast, the provided flexibility is used to benefit specialised problems with certain requirements, such as quantisation for neural networks [18]. It is often less straightforward to implement FPGA designs for input of arbitrary length or format. This is more noticeable in data center analytics situations, where database operators can have an arbitrary complexity, and the problem sizes easily reach “big data” territories. An example limitation in current database accelerators is to resort to hard input size limits [19], even if these are measured in gigabytes [8]. Nevertheless, FPGAs also offer “reconfiguration”, with which the logic can be replaced with pre-generated bitstreams to adapt to the workload. This can increase the flexibility of FPGAs up to a point, as discussed more thoroughly in section 1.2.

This thesis presents new parallel algorithms and improvements on existing FPGA approaches, including for sorting and sort-merge joins for relational databases. The proposed designs are based on novel fundamental building blocks with a wider applicability. Additionally, it provides discussions on future paths in computer architecture in an attempt to further address the presented research challenges for big data analytics on FPGAs.

1.1 Motivation

This PhD is funded by dunnhumby through an industrial studentship, and was heavily influenced by the computational challenges as found in an industrial setting. dunnhumby is a customer data science company based in the United Kingdom. They help retailers and brands achieve sustainable growth and smarter operations through customer first strategies using advanced data science.

Large retailers, such as supermarket chains, use customer analytics to provide insight on key areas including channel, ranging and marketing activities, with the aim to improve customer satisfaction and drive long-term loyalty. Mostly conducted on a relational database, some specialised big data queries are computationally expensive resulting in considerable latency impacting the quality of the service.

From the data organisation perspective, the general idea is that the analytics concern a relational database following a traditional “warehouse” model. Figure 1.1 presents an abstracted schema of such databases, where the green field denotes a primary key. There is a main facts table representing transactions, and has a r_0 rows and k_0 columns. Each row of the facts table has a series of key attributes pointing to each of the N lookup tables, also called dimensions, alongside some other metadata.

The available database management systems (DBMS) yield suboptimal performance in dunnhumby’s specialised queries, hence they have built in-house tools for accelerating a series of special queries. These queries share some unique properties. The expensive aspect of those queries is that they involve “non-aggregable” metrics. The “non-aggregable” aspect is about the challenge

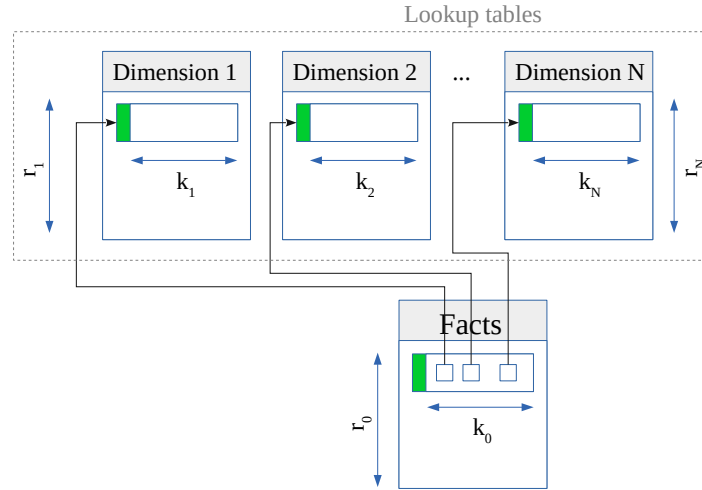


Figure 1.1: An abstracted view a representative database schema

to pre-summarise parts of the analytics, or cache them, to reduce the computation. Dynamic programming, memoisation and caching could be appropriate techniques to solve this problem, but the scale and problem size would be one of the main concerns. One of the reasons for the difficulty of aggregating is the high number of combinations of the query parameters. A common dimension that complicates the aggregability is time. An example query could ask what is the maximum number of unique products bought by a single customer during the last 7 days from now.

The exact computational workload of dunnhumby is commercially sensitive and is not disclosed. For this reason, this project focuses on innovating on the use of fundamental operators widely used in relational databases, as well as in other areas of computer science. Algorithm 1 presents an example representative pseudo-SQL query and how it relates to some of the operators explored in this thesis. Sorting and joins are well known to occupy the majority of computation time in relational databases, though distinct count is also considerably expensive in the studied workload.

Sometimes sorting is not necessary for some parts of this query, including equi-joins, where a hash-join implementation can be applied. However, accelerating sorting is rewarding on FPGAs, because some tasks such as distinct count or equi-joins (using sort-merge join), can

```

1  SELECT table0.key1, table2.key3, count(distinct table3.key1)
2  FROM table0, table2, table3, table4
3  WHERE
4      table0.key2 = table2.id and
5      table0.key3 = table3.id and
6      table0.key4 = table4.id and
7      table4.key2 in (1, 5, 9)
8  GROUP BY table0.key1, table2.key3
9  ORDER BY table0.key1, table2.key3

```

→ distinct count
 → equijoin (e.g. sort-merge join)
 → filtering
 → (e.g. sorting, co-grouping)
 → sorting

Algorithm 1: A representative (pseudo-) SQL query and primitive operators

be applied with linear complexity and streaming fashion on the sorted data. Large scale data favour linear access patterns when using FPGAs, to minimise the impact of the latency induced by the heterogeneity in today’s systems (e.g. through PCIe), and also avoid storing considerable amounts of information on-chip. In the world of databases, sorting is still a widely used operator (including the mandatory “ORDER BY” clause in Algorithm 1), but for traditional software there may be different heuristics to trigger a sort-based algorithm when considered beneficial over hash-based approaches etc.

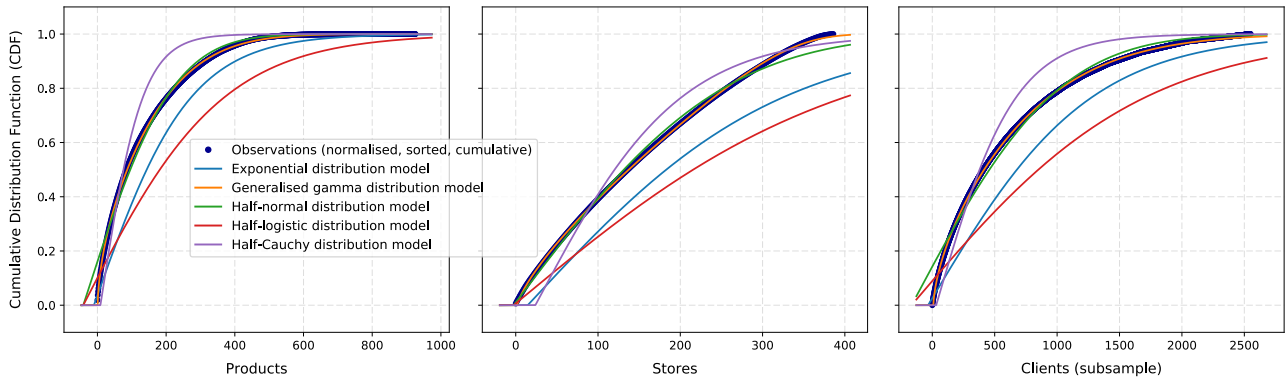


Figure 1.2: Analysing an example dataset from dunnhumby

Last, figure 1.2 presents a preliminary experiment that uses a representative dataset from dunnhumby [20], [21]. Here, the provided experiment tries to fit different cumulative distribution functions (CDFs) on different dimension tables. In this example, the dimensions are products, stores and clients, and the CDF is for the amounts a foreign key is referenced by the main facts table. The winner model is the generalised gamma distribution model and was found using the scikit.learn Python library [22]. This dataset is supposed to be representative for real word analytics, although it is at a smaller scale. The fact that CDF is not a uniform distribution indicates that there can be both sparsity, and keys with high number of duplicates. This is an important piece of information that is taken into account for the developed designs,

to be able to perform well on data with such distribution characteristics.

1.2 Research challenges

This section classifies the challenges about using FPGAs to accelerate big data applications. The solutions presented in this thesis try to address the following challenges either directly or indirectly, such as by improving on inefficiencies found in existing designs.

C1 Inter-chip communication

FPGAs are often left behind with regard to the main memory accesses, as with systems following the non-uniform memory-access model (NUMA). For this reason, high-end FPGAs have their own off-chip memory, such as DRAM or a high-bandwidth memory (HBM) and communicate through PCIe for accessing main memory. This approach is similar in today's GPUs, which have similar limitations. The metrics where the performance overhead is usually reflected are the main memory throughput and latency.

Main memory throughput In general, database accelerators require streaming big amounts of data and they are bottlenecked by the memory bandwidth [23]. What is usually referred to as simply memory bandwidth is the throughput of the co-processor's bigger memory (DRAM) [24], but the memory access throughput to the main memory is much lower. This is essentially because the co-processor (FPGA) usually does not have direct access to the main memory controller [25].

In some systems such as Intel's HARP v1, the FPGA does not have its own big memory [26] and the data still needs to be obtained through the processor. There are also some SoCs (System on Chip) that combine CPU cores with an FPGA and they share similar access speeds to DRAM, but they can be low-end in this aspect.

Latency In big data streaming situations, the FPGA-induced latency can sometimes be unnoticeable. The latency to access main memory becomes important in access patterns that are more irregular, such as with “random” memory accesses. A database accelerator whose performance is susceptible to memory latency is hash join [27], as the entries are references to main memory locations for the elements to be joined. For that reason, workarounds inspired from CPU’s attempts to hide memory latency are used. The FPGA-induced latency is also important in accelerators whose performance is affected by the attributes of the input, such as with many sorter accelerator implementations [28].

C2 Development

Developing an FPGA-based accelerator is not as straightforward as writing plain software for CPUs. First, there are important trade-offs between high-level synthesis (HLS) and (lower-level) hardware description (HDL) languages. As the downsides of each might not be apparent from the start of a project, the presence of these trade-offs add to the time and cost of developing for FPGAs.

HLS offers a faster and more approachable development experience, by being based on procedural and object-oriented languages like C/C++ or Java, and functional languages, such as Scala. One advantage of HLS is the ability to quickly explore implementations and optimisations for different targets (**C3** applicability challenge). On the downside is the reduced expressibility, such as when the provided programming models cannot implement a lower-level feature or optimisation. Moreover, the implementation from HLS tools can have a sub-optimal performance, as the generated logic may not be as efficient as pure Verilog or VHDL (HDL) designs.

Additionally, the aforementioned heterogeneity problem is often reflected in FPGA designs, making them more complex, such as from the need to implement communication-related logic, including for supporting bursts. Considering that the throughput and latency ‘inside’ the FPGA is many orders of magnitude faster than the ports to the host processor or DRAM, research is leaning towards complicated designs for minimising the inter-chip communication. Common workarounds include the implementation of cache hierarchies [27], [29], data prefetching [30]

and compression [31]. Sometimes, the additional off-chip memory is manually added to the memory hierarchy by working as a cache [30]. The use of on-chip DRAM as a scratchpad could be considered the simplest and most common approach. Example applications of the latter include sorting [8], as well as with data off-loading through a general programming model in modern HLS tools [10].

For this reason, vendors try to abstract some of the communication and memory organisation complexity with HLS tools. One related limitation from the reduced expressibility of HLS languages is through the Vitis toolchain from Xilinx [10], where data offloading is enforced by the programming models to happen before processing. This is a problem for streaming applications, as the task completion time could otherwise be decreased dramatically, if some preprocessing can actually happen during the data transfers. Though, this is not a hardware limitation at the moment, and the benefits of overlapping data movement and computation is often demonstrated in other platforms [32]. Nevertheless, the abstraction of such complex systems with closed source toolchains has implications in the understanding of the system as a whole, and limits the possibilities for lower-level optimisations, as well as for systems research.

C3 Applicability and portability

Since FPGAs benefit from specialisation for achieving high performance, FPGA designs are often not portable to other problems or devices. An FPGA design might be optimised to work on a very narrow range of possible input formats (such as 8-bit integers [18]), and re-engineering it for a similar input could require reimplementing the entire design from scratch. According to the design, it may be possible to increase applicability by changing the algorithm upon which the design is based. For example, by sorting the data beforehand, a linear algorithm implemented on the FPGA might be able to process the data in streaming fashion without storing lookup tables and growing structures (as in the sort-merge join accelerator of section 5.2).

The same holds for portability to other FPGAs. For example, the designer might have applied device-specific optimisations for a Xilinx Alveo U280 design, while none of these optimisations

could apply to some different board. These include manual floorplanning, usage of Xilinx UltraRAMs and a high-bandwidth memory (HBM). This can sometimes be solved or avoided to a degree, such as by the use of HLS, compartmentalisation using IPs, such as in the form of AXI peripherals, and avoiding such low-level manual optimisations.

As a workaround to the applicability challenge, the “reconfigurability” feature of the FPGA can be exploited to allow quick transitions between different logic. There are different granularities for reconfiguration, such as with partial reconfiguration of pre-defined regions on the chip. Real world examples include interchanging different bitstreams for different sub-tasks [33] or even different algorithm phases during runtime [17], [34].

There are different overheads at different partial reconfiguration region sizes. Larger designs may perform better a specialised task than combining smaller designs together [33], but they are more costly to develop. Smaller partial designs may have a wider applicability, but their performance could rely heavily on the fixed logic, such as for efficient task pipelining. Appendix C elaborates more on a reconfigurable instruction concept, that could be considered an extreme approach of partial reconfiguration but for usage in CPUs. As with other forms of partial reconfiguration, one challenge in the context of the latter approach is the presence of internal states, elaborated in section 6.2.5.

Still, reconfiguring parts or the whole of an FPGA assumes pre-built bitstreams, which are costly to develop and generate. Even with automated approaches such as with high-level synthesis, the place-and-route is too time consuming for supporting arbitrary tasks. In contrast, in the software world, just-in-time (JIT) compilers are able to generate bytecode from scripts rather seamlessly during runtime. The placement and the size of FPGAs in systems is an important open research question, as every choice has trade-offs. See section 2.5.4 for the latest trends in systems architecture regarding FPGAs, and how these design choices can affect database analytics.

C4 Scalability

The scalability concerns are usually reflected on the hardware utilisation and complexity. For keeping up with technological advances in the data center, high-throughput designs can be challenging to develop. The main considerations are the high hardware resource utilisation and the timing constraints. As with sorting [35], [36], it is desirable for a pipelined stream processor to have a narrow critical path. Otherwise, the operating frequency will need to be lowered to allow complex pipeline stages operating under one cycle. Another example is from the acceleration of sort-merge joins, where the Cartesian product in past attempts was known to limit the output rate [37].

The scalability challenge can also refer to the problem size. Since we target big data, it is very desirable if the proposed designs are agnostic to the problem size, such as with the presented hardware merge-sorter, which can be applied recursively (see section 5.1). One problem with supporting larger problem sizes is the growing number of states that are sometimes required to be kept. As cache hierarchies are rather costly to implement on FPGAs, it is tempting to restrict the specification to limited problem sizes. According to the algorithm, keeping all states on-chip could yield the highest performance, but this may not be applicable for arbitrarily long data, which is the studied use case.

Another aspect of scalability is the possibility to use a distributed FPGA approach to solve a problem collectively. In this case, a problem would need to be able to be broken down into multiple sub-problems. In general, there are two main approaches for parallelisation, horizontal and vertical [38]. Horizontal parallelism can be achieved naturally on FPGAs, as it involves SIMD-like parallel processing through a single worker. Alternatively, vertical parallelism involves multiple workers, and this is also possible with FPGAs, as with Microsoft's Brainwave project that demonstrates high-performance for the Bing search engine using a scalable fabric of servers with FPGAs [7]. Finally, a common technique for minimising inter-FPGA communication in certain problems is pipelining. A task-pipelining example in the literature about accelerating database analytics is a sort-merge join accelerator, which uses dedicated FPGAs for the sorting phase, while the merging takes place on a separate FPGA that processes the

sorted data on-the-fly [23].

A common theme with the scalability issues of FPGA designs is the requirement for full interconnection between multiple processing entities. This applies from lower-level circuits, such as monolithic FPGA designs to large-scale distributed systems, such as with multiple FPGAs. For monolithic designs, this can appear as routing complexity, which can translate into a high latency [13] or infeasibility for a high inter-connection radix. For instance, on U280 there are 3 FPGA dies that are inter-connected by the stacked silicon interconnect (SSI), and highly-interconnected designs like network switches are challenging to split across multiple dies for accommodating high numbers of ports [39]. In distributed systems, this challenge can appear as a performance overhead due to the increased communication and access latency between nodes through centralised means, including a shared memory or a network stack. An example challenging application in high-performance computing is the N-body problem, where each simulated body acts forces on all other objects in the universe. As a workaround to the scalability challenge, different techniques try to either limit the connectivity or the communication between the different computing entities with a minimal overhead. These include, but not limited to, hierarchical and modular approaches [13], approximations [40], and other approaches that exploit temporal and spatial localities, such as with adaptive cache systems in multi-core processors [14].

1.3 Publications and contributions

This section presents a list of the first-author peer-reviewed publications, and how they relate to the contributions of the thesis. Each one of the list items represents the basic relationship between a paper, a set of contributions and a chapter, though their actual relationship is more interconnected.

1. Initially, a systematic literature review was written for the topic on accelerating database analytics using FPGAs. This was presented in the publication “**Accelerating database systems using FPGAs: A survey**” [41].

Chapter 2 could be considered an extended version of this literature survey, as it contains general up-to-date references and comparisons on the state-of-the-art.

2. One of the main contributions is an efficient building block for a merge sorter, through the publication “**FLiMS: Fast Lightweight Merge Sorter**” [36]. FLiMS is a high-throughput 2-way merge processor, based on the half-cleaner structure, plus a butterfly network, as originally found in the bitonic sorter. It is able to sort 2 input lists in streaming fashion with high-throughput, given that the datapath is sufficiently wide. It was at least twice as efficient as the then-state-of-the-art (MMS [35]) in terms of FPGA resource utilisation, with subsequent small improvements in the maximal operating frequency. Today, it is still the most efficient and scalable 2-way high-throughput merger for FPGAs.

Chapter 3 elaborates on FLiMS, including an optimisation for skewed datasets, as well as an equivalent implementation using plain software in C++. FLiMS addresses the challenges **C1** inter-chip communication, **C3** applicability, portability and **C4** scalability, as its simple and lightweight logic achieves high-throughput and low latency for high input widths and different platforms.

3. The second sorting paper “**An Adaptable High-Throughput FPGA Merge Sorter for Accelerating Database Analytics**” [32] introduces an end-to-end practical evaluation that uses FLiMS to achieve up to a 19x speedup over a single core from Kaby Lake i7, but on a Linux-running low-power FPGA board. The same work also introduces variations and optimisations on more classical sorting approaches, including the linear sorter. Here, FLiMS is used hierarchically to build a merge tree to achieve high-throughput from using multiples of a novel single-rate structure for merging.

The findings of this paper are presented in **chapter 5**, mostly focusing on the many-leaf merge aspect, for merging many input lists with a few passes efficiently. It mostly addresses the throughput challenge (**C1**) for sorting by minimising the number of passes [42], as well as its scalability for advanced and future technologies that allow a wider datapath (**C4**). While it can be configured to use a low number of resources, it is also applicable to sorting arbitrarily long input, improving its portability (**C3**) to devices of

different sizes.

4. The paper “**Accelerating the Merge Phase of Sort-merge Join**” [43] presents a more database-centric design, for accelerating equi-joins. A series of high-throughput building blocks are developed to enable scalable equi-joins. The join block was a modification of FLiMS, for facilitating the join operation after the records are being compacted using a co-grouping engine. The co-grouping engine mainly consists of “parallel round-robin arbiters”, which is a pipelined structure for removing sparsity. The sparsity only happens in intermediate steps of processing here, and an equivalent methodology can be followed to create other scalable analytics pipelines for wide datapaths, and is also reminiscent of SIMD instructions in modern CPUs.

Chapter 5 also introduces the sort-merge join accelerator, along with some other useful relational database designs. The provided designs mostly focus on applicability (**C3**) and scalability (**C4**). This is because of the use of algorithms of streaming nature and the column-store model, for accepting input of arbitrary length and distribution. Saturating the bandwidth (**C1**) through high-throughput is also one of the fundamental features of these designs.

5. Another contribution from the work on accelerating equi-joins was the “parallel round-robin arbiter”, as it is found to be useful in other applications, including network switches. The best paper candidate “**High-Performance FPGA Network Switch Architecture**” [39] evaluated an idea for using a variation of this structure for implementing an FPGA-based network switch, called the “combined parallel-round-robin arbiter” instead of the commonly used crossbar. It achieved an aggregate bandwidth of 1.7 Tbps for a 16-port based switch on U280, as well as better algorithmic performance than any ordinary scheduling algorithm on crossbars, for common traffic patterns.

Chapter 4 elaborates on the “parallel round-robin arbiter”, and presents briefly the analysis of the network switch as a data center use case. For database applications, it mainly addresses the **C1** inter-chip communication challenges, since it can be used as a building block for high-throughput stream processors.

The following relate to the thesis material as well:

- A journal paper expands the network switch paper [39], and is called “**Hipernetch: High-Performance FPGA Network Switch**” [44]. It extends the material with datacenter simulations and a more thorough analysis of the achieved performance on an updated platform.

This is partly reflected in **chapter 4** and appendix B, but only for providing an updated dataset for the network switch implementation [39] using a higher-end FPGA.

- The paper “**Simodense: a RISC-V softcore optimised for exploring custom SIMD instructions**” [45] introduces an open-source framework for implementing and evaluating novel SIMD instructions for CPUs. The idea is that having small reconfigurable areas in general purpose CPUs, for implementing custom instructions would be a very practical and beneficial future organisation of the architecture. Small bitstreams could be loaded by each piece of software, that contain implementations for specialised instructions for increased performance and other benefits, including reduction in deployment costs.

This novel softcore is presented in **appendix C** and addresses the (C2) development challenge. While it is more of an exploration platform than an acceleration platform, it also addresses the inter-chip communication challenges (C1) indirectly, by supporting and elaborating on the idea to move FPGAs inside the processor. CPUs have already solved the challenges C3 and C4, as the computation and memories are rather-efficiently abstracted through software, and this chapter shows how these benefits can be applied to FPGA designs.

- The workshop version of the softcore publication [45], called “**Extending the RISC-V ISA for exploring advanced reconfigurable SIMD instructions**” [46] elaborates more on the programmability aspect of reconfigurable SIMD instructions.

Some material of this paper is also discussed in **section 6.2.5** and appendix C.

Finally, the following peer-reviewed publication is indirectly related to the presented material, but is not discussed in the thesis:

- Also in the area of processor microarchitecture, the workshop paper “**Pangloss: a novel Markov chain prefetcher**” [15] presents a novel data prefetcher for increasing the instruction level parallelism for a wide range of access patterns. This prefetcher, or more specifically its structure for representing Markov chains, is also applicable to FPGA designs, since it achieves high performance without associating instruction pointer values (IPs) with any of its decisions.

Chapter 2

Background

Database systems are a key component to a plethora of computing applications, and FPGA-based solutions have shown promise in accelerating high-performance database systems. This chapter is a literature survey that presents a systematic review of research about accelerating analytical database systems using FPGAs. It includes studies of database acceleration frameworks, as well as accelerator implementations for various database operators. Finally, it also discusses some promising current and future technologies that would help with database analytics acceleration on FPGAs.

FPGAs are an attractive option for accelerating different database systems, and provide big amounts of flexibility in a relatively cheap package. Since the functionality of a database can be specific to the user's needs, FPGAs may offer the most cost-effective solution, according to the individual case. Due to the need for specialisation, developing a general database processor, such as with static ASIC designs, would be more expensive [3].

Database systems can be demanding, as their size can easily reach 'Big Data' dimensions, with tables acquiring terabytes of memory. Traditionally, the database systems were mostly On-line Transaction Processing (OLTP) systems. These systems are characterised by simplicity in terms of the transactions. With the rise in popularity of data mining and machine learning in big data, another database category emerged, called On-line Analytical Processing (OLAP) which has more complex queries that need to access and summarise multi-dimensional data. In

the latter category, FPGAs have been perceived as an attractive solution as in many operations the data is accessed in a streaming fashion and the queries are sometimes pipelineable.

FPGA development has its own challenges. Different architectures, including CPUs and GPUs, have different inefficiencies and introduce different bottlenecks to performance. In order for an FPGA accelerator to be effective, the problem needs to have certain qualities, such as to be naturally parallel. FPGAs may perform significantly worse than a CPU if the application is control-heavy, i.e. if it has a lot of unpredictable branches. The challenges in design can also include the steep learning curve, as well as the inefficiencies and proprietary nature of most FPGA development toolchains today. See section 1.2 for a discussion on the challenges of FPGAs with relation to the acceleration of database analytics.

The chapter is an updated version of the paper “Accelerating database systems using FPGAs: A survey” [41] and covers a range of research wider than the solutions presented in the thesis. The material on sorting was mostly based on the more recent sorting paper “An Adaptable High-Throughput FPGA Merge Sorter for Accelerating Database Analytics” [32].

2.1 Classification

The related work on database accelerators can be divided into frameworks and specialised accelerators.

The frameworks class (**section 2.2**) includes solutions that provide a software and hardware stack for accelerating user-defined database operations. For instance, this class includes virtualisation frameworks for providing a VM-like functionality in data centers for users to deploy their own accelerators in the cloud.

In the ‘accelerators’ class (**section 2.4**), there are database processors that are specifically designed to support a set of common query clauses. This also includes relatively simplistic FPGA designs to accelerate single operators, but they can usually be used in a framework of the previous class.

As sorting accelerators are one of the main topics discussed in the thesis, and their applicability extends beyond SQL queries, the related work on hardware sorters is discussed in a separate section in more detail (**section 2.3**).

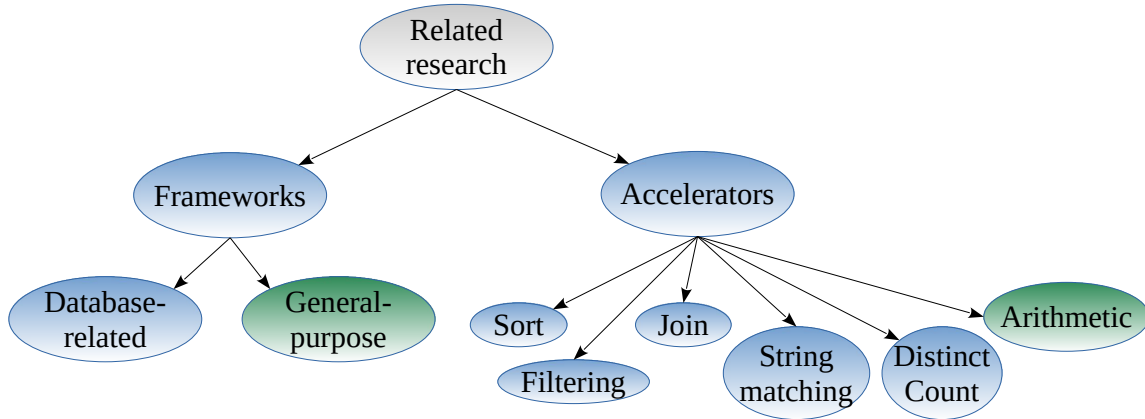


Figure 2.1: Main classification of the related research work

Figure 2.1 presents an overview of this classification of the research related to accelerating database systems using FPGAs. The two main categories are the ‘frameworks’ and ‘accelerators’. The ‘frameworks’ class is further divided in database-related frameworks, as well as more-general purpose frameworks that could support database functionality but are not domain-specific. The ‘accelerators’ class contains different families of accelerators related to databases and big data, including sort, join, filtering, distinct count and string matching operator accelerators. A more broad subcategory of accelerators is the ‘arithmetic’, which is a selection of hardware designs performing a usually-simple arithmetic function, or machine learning-related, on big amounts of data.

Note that this classification mostly serves the organisation of the thesis, and the relationship of each related piece of research is more complex. For instance, individual works on operators, including filtering [33], may provide a specialised framework to potentially explore a wider range of similar operators in a database context.

2.2 Database acceleration frameworks

2.2.1 Frameworks directly related to databases

One example framework providing a complete hardware-software solution is Centaur [26]. Centaur consists of two parts, the hardware part called the “FThreads manager” and the software part called the “Application interface”. The software supports API calls in C++ through the UDF (user defined function) functionality of MonetDB [47]. The hardware component is responsible for managing the different user-defined hardware in terms of resource utilisation, memory accesses, and also pipelining.

Centaur implements a dynamic pipelining functionality. The FThreads manager connects consecutive hardware operators with FIFO queues for fast intermediate communication to enable pipelining in hardware. This is abstracted in the software, as the user is able to pipeline the operators in source code. The user can define it programmatically by using the FPipe object to pipe the output from one operator to another. The UDFs also allow pipelining software and hardware components. This is useful because the CPU is shown to perform better on some operators and the FPGA resources might be too limited for all the required operators to co-exist. The addition of more heavily-optimised accelerators (and UDFs in this case) is left to the user.

DoppioDB [48] can be considered an extension of Centaur. DoppioDB uses MonetDB along with a number of optimized hardware designs for the FPGA, exploiting the UDF functionality for making calls to the hardware accelerator. DoppioDB implements regular expression, skyline and stochastic gradient descent on the FPGA and the rest of the workload is left for the CPU to process. This resulted in a system that can execute SQL queries for an analytical relational database with over 3 times speedup when compared to the baseline, which was to execute the queries on a high-performance CPU without acceleration.

DoppioDB 1.0 [25] extends doppioDB to also enable accelerating a broader range of machine-learning (ML) related workloads, on an updated multi-threading platform. It extends the SQL language with ML-focused clauses, such as ‘INFER’ and ‘CREATE MODEL’ to enable using different models. It is found to accelerate a broad range of applications with stochastic gradient

descent (adopting [49]), and a physics application based on decision tree ensembles (adopting [50]), all with a speedup of around 3x over a highly-optimised software implementation. DoppioDB 2.0 elaborates more on the ML-aspect of hardware-based database acceleration [51], such as with integrating a generalized linear model (GLM) training accelerator [52].

The selection of MonetDB, a database management system (DBMS), was specifically selected in these pieces of research because it supports giving access to whole tables, as well as adding the UDFs in plain C++. Giving access to entire tables is an advantage over the more traditional row-by-row processing model, because it allows the table to be accessed in a streaming fashion by the FPGA. MonetDB is a column-store database, as it uses binary association tables (BATs) as a data representation [47], being focused more on high performance and big data. This is partly due to the natural ability to ignore fields who belong in infrequently accessed columns, having therefore more efficient memory access patterns. Other advantages of MonetDB for future research is that it is open-source and it also approaches an in-memory database, which is an increasingly popular storage model and is here to stay [53].

There are also some database analytics that perform best on data that uses the row-store format instead of the column-store, due to the nature of the queries, such as for machine learning on multi-dimensional data points. DoppioDB 1.0 [25] introduces an on-chip (using FPGA's block RAM) customised cache that translates column-oriented data into rows for achieving high-performance for queries that favour either a column-store or a row-store format.

2.2.2 General-purpose frameworks

There is also a selection of frameworks that are not necessarily related to databases, but could be a useful component in a system for accelerating database analytics using FPGAs.

One example framework that targets a wider range of applications is an attempt to provide virtualisation services for FPGAs in data centers [54]. The main contribution was the extension of the OpenStack framework with tools to enable FPGA resource assignment to clients without making compromises on performance and ease of use. The solution is based on par-

tial reconfiguration, as the main component of the FPGA design is a static logic to manage the client-programmable VFRs (Virtualized FPGA resources). The static logic mainly consists of input and output arbiters that ensure fast and secure communication and eliminate unpredictable states. There are contributions with both hardware designs and software, as a complete compile flow and boot sequence is provided, such as a netlist file for the Xilinx tools to connect client designs in Verilog and produce bitstream files. The added latency for arbitration between different FPGA designs is considered negligible [54].

Related projects are also frameworks that provide ease of use for partial reconfiguration. For instance, GoAhead [55], [56] is a framework that can automate the process of building complex reconfigurable systems. The main idea for these FPGA-based systems is to offer easy management of the partial reconfiguration regions, such as for custom instruction implementations for softcore processors. It has already been evaluated in the context of database acceleration [33] and other datacenter uses [57].

High-level synthesis (HLS) tools could also be seen as frameworks for database acceleration, as they provide a way to interact more easily between software and hardware. Relatively popular general purpose frameworks include the MaxCompiler [4], the Xilinx Vitis Unified Software Platform [10], as well as Intel's OneAPI [58]. By using any of these solutions, simple accelerators like some of the 'arithmetic' type can usually be expressed in a few lines of code. One of the advantages of these frameworks is the easy system integration in highly heterogeneous systems, which are sometimes provided by the vendor themselves. For example, Maxeler systems provide a high-level synthesis tool for FPGA accelerators along with their own dedicated high-performance servers. More advanced database accelerators, such as for sorting and joining, have also already been demonstrated on a Maxeler system [23], using server configurations combining multiple FPGAs with less programming effort. See challenge **C2** of section 1.2 for a discussion on the advantages and disadvantages of high-level synthesis.

2.3 Sorting accelerators

A sorting module can be very useful in a database processor as many queries perform aggregations or table merging, and according to their implementation and requirements this can be their costliest operation [23]. It can be used for sort-merge joins, which is a join algorithm easily benefiting from FPGA use [23]. Joins is one of the cases where different algorithms perform better on different platforms and data distributions [59]. Quicksort and variations are sometimes advantageous on CPUs, but not on FPGAs [59]. Optimisations of quicksort can be found in the Oracle database implementation [60] and this research area is still fairly active. Other uses of sorting on databases include various analytics-related queries, hence the tau notation in relational algebra which represents sorting.

A considerable amount of the latest research on sorting on FPGAs, as well as in this thesis, is based on mergesort. This is most likely because of the streaming nature of mergesort and the fact that it can be applied recursively for arbitrarily long input with a fixed amount of hardware resources.

2.3.1 Small-scale sorters

This category of sorters, refers to efficient designs for sorting on-chip data (residing on registers and block RAM). They can be used to replace the first stages of mergesort [42] as well as in other FPGA sorting approaches [8].

One notable example for FPGA implementation is sorting networks [61]. Batcher's odd-even mergesort and the bitonic sorter [61] are two popular examples. These two sorting network topologies can be implemented as pipelines of $\log_2(n) \cdot (\log_2(n) + 1)/2 \in \Theta(\log_2^2(n))$ stages, and each stage consists of up to $n/2$ compare-and-swap (CAS) units, where n its input size is in powers of 2. The functionality of the CAS units is to sort two values (i.e. $a, b \leftarrow \min(a, b), \max(a, b)$). Collectively, a sorting network in its entirety can sort any list of n inputs. For efficient FPGA implementations, each layer can easily be transformed into a pipeline stage, by appending a set of registers at the end of each stage [28].

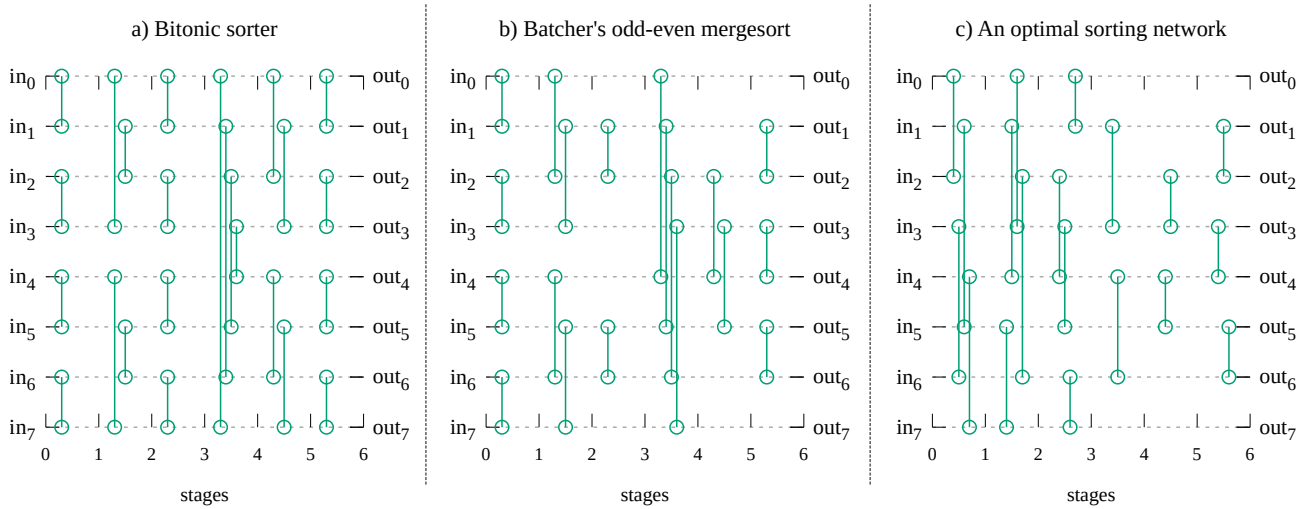


Figure 2.2: Three sorting network examples for 8 inputs

Figure 2.2 shows three example sorting networks for 8 inputs. The sorting networks (a) and (b) are the bitonic sorter and the Batcher's odd-even mergesort [61] respectively, and are easy to develop for any number of inputs, due to their regular and hierarchical structure. For example, the first 3 stages of (a) contain two bitonic sorters of 4 inputs, and then the results are merged using the bitonic merge structure, which takes 3 more stages. The equivalent sentence could be said for (b), as it is also based on mergesort. The bitonic sorter has more comparators than the odd-even sorter, hence the odd-even sorter performs slightly better on FPGAs. Though, the bitonic sorter is more regular in the sense of having the same number of comparators in each layer. For this reason, the bitonic sorter is preferred for SIMD-intrinsic-based implementations [5], [62].

The sorting network (c) is a depth-optimal (in the number of stages) and size-optimal (in the number of comparators), generated by a custom-made genetic algorithm. Optimal sorting networks are difficult to find, as the number of possible CAS combinations grows very quickly with size. Validating a single CAS network is also part of the challenge, as the number of possible permutations of the input grows exponentially with the number of inputs. This holds even when applying design space reduction techniques [63]. Note that the example (c) is just for illustrative purposes, as for as few inputs as 8 (b) is also depth-optimal (6 layers) and size-optimal (19 comparisons), and (a) is depth-optimal too, but not size-optimal (6 layers $\times 8/2 = 24$ comparators). However, the general idea remains that optimal sorting networks are

less regular and less desirable for scalable designs, as they cannot be expressed recursively and have yet to be found for 18 elements or more [64]. Other attributes that could be useful for fine-tuning FPGA implementations, such as the total wire length are mostly unexplored.

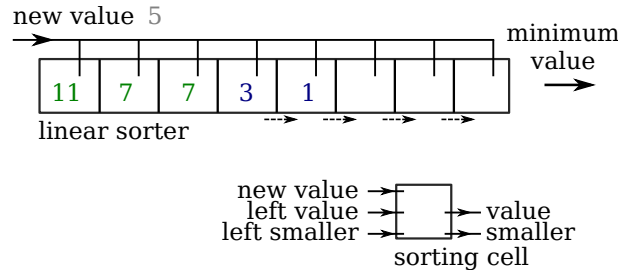


Figure 2.3: Linear sorter example

A simple small-scale sorter is the linear sorter [65], [66] and can be seen as a parallel version of insertion sort [67]. As shown in the example of figure 2.3, a new value is inserted on every cycle. This value is broadcasted across all sorting cells, which compare it against their current value. The result of each comparison is known to the immediately right cell, after a sub-cycle delay. When a cell currently holds a smaller value, it adopts the value on its left. When a cell is the leftmost to have a value smaller than the new value, it adopts the new value instead. This procedure produces a sorted list in $n + 1$ cycles, where n is the input size. There are variations and similar structures, such as fast systolic sorters at the expense of additional latency [68].

2.3.2 High-throughput merge sorters

High-throughput merge sorters can merge a number of sorted lists simultaneously, while providing an output rate of more than 1 element per cycle. This can be achieved by building a merge tree (PMT [69]), mainly consisting of high-throughput mergers of 2 lists and FIFO queues.

Figure 2.4 shows how these building blocks can be arranged to merge 8 sorted inputs of throughput 1, with an output rate 8. The ‘merge rate’ denotes the throughput capacity of the merger for each level. For instance, a merger of rate 4:2 merges two inputs of width 2 (times the element width) and outputs two elements per cycle. The difference in widths from level to level

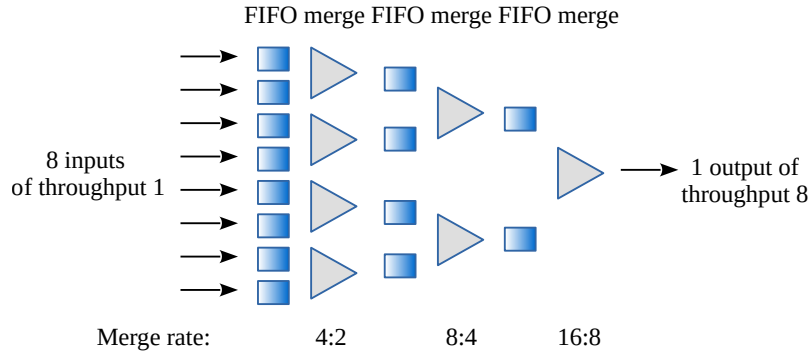


Figure 2.4: Parallel merge tree (PMT [69]), for 8 input lists

is managed by rate converters and the appropriate stall signals.

A simpler unoptimised tree [37], [70] can also be implemented with mergers of the maximum throughput, which also squares the aggregate bandwidth requirements of the inputs [69]. This leads to underutilised resources, such as unnecessarily large merger blocks. As shown in the example figure 2.5, the merger blocks used are identical. However, in some special cases it has a slight advantage over the optimised PMT [69], such as when recovering from rate mismatch events more quickly (see 3.2.1 for more information on rate mismatch).

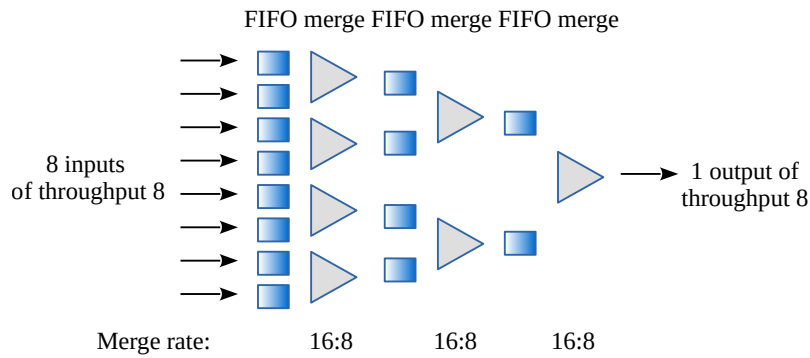


Figure 2.5: “Unoptimised” parallel merge tree, for 8 input lists

2.3.3 High-throughput sorter building blocks (2-way mergers)

A merger for 2 already-sorted sublists of fixed length can be modified to merge 2 lists of arbitrary length in streaming fashion. Then, it can be used as a building block for a parallel merge tree, in order to merge many lists simultaneously.

Most of them are based on the two popular sorting networks: Batcher’s bitonic sorter and odd-even mergesort [61] (figure 2.2 a, b). These two sorting networks have the same number of stages and can be built hierarchically using 2 sorters of half the input plus an appropriately sized merger for merging two equally-sized sorted sublists. The merger part consumes the last $\log_2(n)$ stages in both sorting networks.

Some 2-way hardware mergers (including FLiMS of chapter 3) utilise an optimization of the bitonic merger, the $2w$ -to- w bitonic partial merger [71] which outputs just the top half of the result (or lower if the CAS units are flipped). The bitonic partial merger is a subset of the bitonic merger. These are summarised in figure 2.6. This “pruned” merge block is combined with additional logic to work as a parallel merger for longer lists as streams.

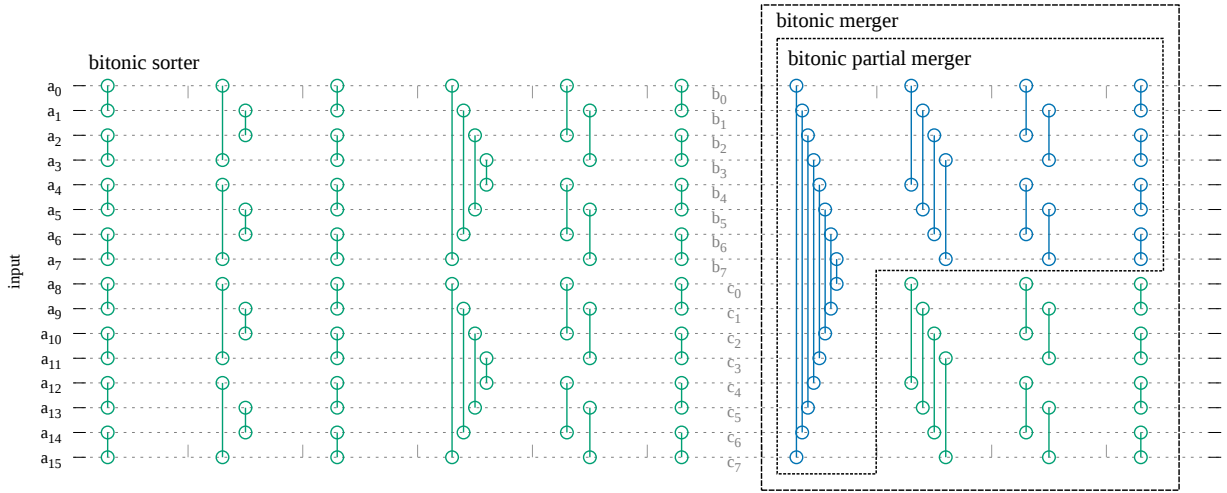


Figure 2.6: A 16-to-8 bitonic partial merger block as found in a bitonic sorter for a list of 16 elements

In 2016, Song et al. built the parallel merge tree (PMT) with $2w$ -to- w bitonic partial mergers [69]. Figure 2.7 illustrates a high-level view of the merge block. This merger works as follows: two input queues, A and B, output 0 to w elements each per cycle, according to how many made it in the last result of w elements. This is known from just the first stage of the $2w$ -to- w bitonic partial merger and is used as a feedback to select the amount of elements to be dequeued from A and B. Since some elements remain from the previous cycle, each input of the bitonic partial merger needs to be rotated by an offset equal to the number of dequeued elements (so far). This is done to ensure that the bitonic partial merger gets sorted inputs. The problem with this approach is that the crossbars implementing the barrel shifters create a critical path

that increases with w . This leads to lower frequency designs and it does not scale very well [72].

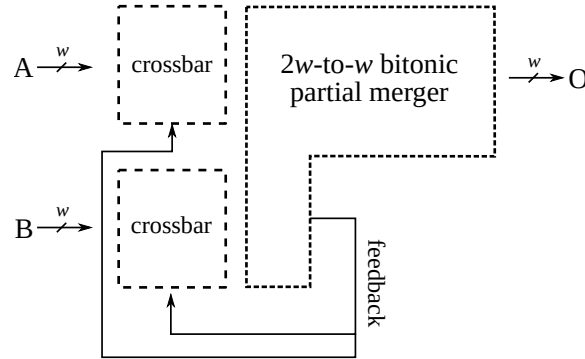


Figure 2.7: High-level view of the merger used in PMT [69]

Note that the crossbar is a more general structure and can have different implementations and alternatives. For example, it can be implemented in a single-pipeline stage, but the timing characteristics would be suboptimal and this does not scale well. It could also be implemented as pipelined barrel shifters working as rotators (or banyan rotators, section 6.2.2), with $\log_2(w)$ pipeline stages, but limiting the throughput to the $1/(\log_2(w) + 1)$, due to the feedback occupying many cycles.

At some point, the most attention was drawn on removing the expensive feedback length that existed in traditional merger designs [37], [69], that prevented scalability in terms of operating frequency for an increased degree of parallelism (w) (SHMS[72], MMS[35], VMS[73]).

In 2017, Mashimo et al. [72] proposed a lower-latency-feedback architecture, SHMS, to solve the long critical path problem with the previous approaches. While achieving much higher frequencies, as high as 3.14 times more than PMT [69] for 32 different input queues (also focusing on multiple inputs), the register utilisation was much higher. For 32 inputs, it uses 7.26 times more registers than PMT. This does not scale well, not from the long feedback datapath, but due to the high FPGA resource utilisation.

Then, Saitoh et al. [35] proposed a feedback-less architecture, MMS, to increase the performance and scalability of the merge operator. MMS uses two $2w$ -to- w partial merge blocks along with shift registers and an extra comparator and multiplexer. Figure 2.8 shows a high-level

view of this design.

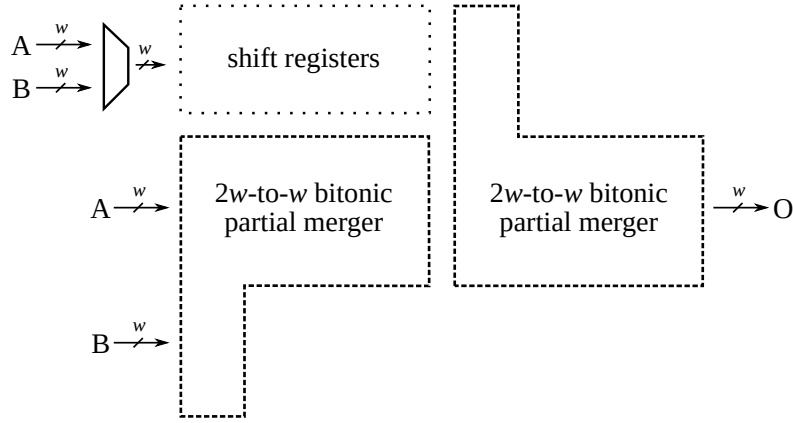


Figure 2.8: High-level view of the merger used in MMS [35]

Finally, FLiMS [36] (chapter 3) and WMS/EHMS [74] /EHMSP [75] offered further improvements by focusing on efficiency, for minimising the required hardware resources, usually with a subsequent improvement in operating frequency. Essentially, WMS is an optimisation of MMS [35] (and VMS [73]), because it fuses the two $2w$ -to- w partial mergers into one bigger merge block, and also eliminates the need for additional shift registers. EHMS and EHMSP try to move some complexity to the selector stage for a lower resource utilisation at the expense of a lower operating frequency.

See section 3.4 for an extensive comparison of the latest 2-way merge blocks, including FLiMS.

2.3.4 Many-leaf merge sorters

Many-leaf (or large-utility) sorters try to merge as many input queues as possible (currently up to a few thousands [42], [76]), with minimal hardware resources. The comparators are proportional to the number of merge stages in the merge tree ($O(\log_2(k))$, where k is the number of input queues).

Even though their output rate is a single element per cycle, they can still have a considerable advantage over the traditional merge trees, which are high-throughput. This is especially true for large-scale sorting (i.e. for sorting off-chip data), because it requires fewer merge iterations,

and subsequently fewer data passes. As the inter-chip data movement is expensive, the number of passes is roughly proportional to the time for the whole task to finish. Therefore, reducing the number of passes reduces the overall time significantly. Thus, this idea, sometimes referenced more generally as high-utility [42], [77], is orthogonal to the trade-off of CPU-bound versus memory-bound computation software classification.

Many-leaf mergers were first demonstrated for data stored in the on-chip block memory (BRAM) [76] and afterwards with data stored in FPGA’s DRAM in a bare-metal configuration [42]. The latter was a more realistic use case, as DRAM is more appropriate for large-scale data. In this case the BRAM was used for buffering instead, for mitigating the increased latency from the occurring random memory access pattern.

Section 5.1 introduces an approach that combines a FLiMS-based merge tree with many-leaf/single-rate sorters, to enable both high-throughput and many-leaf sorting [32].

2.3.5 Non-merge-focused sorters

One recent large-scale FPGA sorter uses samplesort as its main algorithm [8]. Although not merge-based, it is reminiscent of mergesort in the sense that there need to be partitions that need to be sorted individually, using an open-source small-scale sorter [78]. In principle, samplesort’s advantage is the elimination of a merge phase, although many-leaf merge can hide this phase if it happens on-the-fly (see section 6.2.4). Being mostly focused on a specialised full-system implementation, the evaluation for small lists, narrow data and different distributions is left as future work.

With regard to older research, in 2005, J. Harkins et al. [59] evaluated the performance of five different sorting algorithms on an FPGA: Quick sort, Heap sort, Radix sort, Bitonic sort, and Odd-even mergesort. The conclusion was that Radix sort was the winner but it was only twice as fast as quicksort on the CPU, due to the technology limitations of that time, such as the lower memory throughput and the lower clock frequencies. Mueller et al. reached similar observations for sorting networks with 2012 technology [28]. The findings were that

an FPGA consumes much less power than a processor for similar performance and also that more tight collaboration of an FPGA with a CPU can yield better performance. They studied the performance of sorting networks (odd-even sort and bitonic sort) on FPGAs and evaluated heterogeneous mergesort, which made use of the CPU for scalability to longer inputs.

There are also works that focus more on memory and energy efficiency. Chen et al. [79] in 2015 showed that a bitonic sort implementation can be 2 times more memory efficient and 5 times more energy efficient when compared to a CPU implementation. They also showed ways of trading performance for more memory and energy efficiency by reducing the throughput with which the input data are accessed from the FPGA. Srivastava et al. [80] have further optimised a bitonic sorter implementation for accelerating big data applications. They showed that the last merge operations of mergesort are a bottleneck for large-scale sorting and replaced them with a heavily-pipelined bitonic network for higher throughput. It was inspired by the aforementioned work on energy efficiency [79] and a work on sort-merge join acceleration [23]. The result was a considerable speedup over the then-current mergesort implementations.

Relatively recently, Bonsai was introduced to enable customising merge trees and modelling performance for FPGAs with different capabilities [34]. Although it is based on older merge approaches, the focus was given on other innovative aspects, such as reprogramming the FPGA during runtime for implementing different algorithm phases, and an analytical performance model. FANS [17] extended this approach and the analytical model to target Samsung's SmartSSD, and was inspired by an older near-data-processing (NDP) sorting solution [81]. Another sorting accelerator on Samsung's SmartSSD is NASCENT [82], which used bitonic sort and plain bitonic mergers as found in the bitonic sorter. One of the main scopes in this work was to evaluate the energy efficiency, as well as to use distributed computation through combining multiple SSDs with FPGAs.

2.4 Database-specific operator accelerators

In this subsection, a series of accelerators are reviewed that are specific to databases or applicable to databases. Most of these are inspired from SQL queries, since it is an almost universal language for expressing database operations, more extensively than relational algebra (a more theoretical representation). Database operator accelerators can be divided into one of the following categories: **join**, **string matching**, **filtering**, **distinct count** and **arithmetic**.

2.4.1 Joins

The join operation is a frequent operation on relational databases. The main functionality of joins is to combine the columns of two tables by using different criteria. One common join operation is equi-join, which takes as input two tables and performs an equality comparison for merging rows with common values in specified columns. There are multiple join algorithms, such as the nested-loop join, sort-merge join and hash join, all of which have already been implemented on FPGAs.

Casper et al. [23] created a full equi-join implementation in hardware by pipelining different modules in a multi-FPGA solution, as found in a high-performance Maxeler system. Their approach was to do the sort-merge completely in hardware for minimizing the bandwidth bottleneck and therefore including a sorter module. Sort-merge joins first require the two tables to be sorted by the respective key/column on which the comparison will be performed.

The main drawback of this sort-merge join approach is the presence of a replay buffer, which is explained to be a bottleneck on certain inputs, as it makes the access pattern less linear. This buffer is finite and a workaround to support many duplicate keys would further go towards having a random access pattern. In terms of scalability, the sorters in this work are not feedback-less, a matter that is solved in later pieces of research (see sections 2.3.3 and 3.4). One related consideration is that the maximum output rate is relatively restricted because the hardware complexity of the join part grows quadratically with the degree of parallelism, as it needs N^2 comparators for matching keys from N rows in each input at a time [23].

There is also another sort-merge join accelerator, that focuses on an embedded heterogeneous platform (Xilinx Zynq) [83]. This work combines a sort accelerator with a merge phase algorithm for the CPU. It does not provide a merge accelerator in hardware, as it is left for software.

Hash joins have also been successfully implemented on FPGAs. The main consideration with hash joins is that hash operations may perform suboptimally when the hashed locations are not provided efficiently in a memory hierarchy. For example, when using a CPU, the multiple-level cache system will help hide the memory latencies from accessing the data if the working set is sufficiently small. Halstead et al. [27] showed that by mimicking multi-tasking as found in multi-threaded implementations for multiprocessors, there is a better memory hierarchy utilisation resulting in increased performance. The approach was to use a hardware-aware join operation to eliminate the effects of the absence of cache in the FPGA.

Another research work on hash joins on FPGAs was by K. Kara et al. [84]. It focused more on accelerating the data partitioning for parallel hash joins, as a workaround for the random access pattern to enable saturating the main memory bandwidth.

Roozmeh et al. [24] demonstrated how a heterogeneous system that combines an FPGA, a CPU and a GPU can accelerate database join operations. Their approach was to compare the performance of nested-loop join and sort-merge join for FPGA and GPU and they showed that a high-end FPGA could outperform the selected GPU test platforms.

Another category of related accelerators targets small scale databases, such as for embedded systems. A more recent work [19] uses a parallel tree-based join (PTJ) structure that resides on the FPGA. The solution performs well enough for medium-sized input, but the tree structure increases and slows down considerably for bigger inputs and it also inherits a size limitation from BRAM, thus making it inappropriate for big data.

2.4.2 String matching

Another focus of database acceleration research is string matching operations. One of Centaur’s [26] and also DoppioDB’s [48] hardware user-defined functions (HUDFs) that were provided for evaluating the performance of the frameworks was a regular expression engine.

Regular expressions can be seen as short scripts which define complex string matching routines. With the rise of machine learning and its integration in relational algebras for analytic operations, the non-strictly formatted data, such as strings, can become the bottleneck in analytical databases [48]. The claimed speedup is up to 3 times in comparison with a high-performance CPU implementation. The accelerator implementation was described in more detail in [85]. This work focuses on database applications and provides more flexibility and lower resource utilisation by breaking down the regular expressions into parameters. This is achieved by having a flexible state graph that changes according to the automatically-generated parametrisation configuration vector.

Accelerating regular expressions on FPGAs is not only found in database analytics. The work of Sidhu et al. [86] dates back to 2001 and demonstrates non-deterministic finite automaton implementations on an FPGA working with $O(n)$ speed, where n is the number of characters in a string.

2.4.3 Filtering

Filtering is one of the basic operations in relational databases and is used to get a subset of records based on certain criteria. In SQL, this is achieved through the clause “WHERE”, and is often used with joins to combine tables and produce new information from cross-referencing records. The complexity of filtering is easy to overlook, as equi-joins could be enough in some use-cases, hence only considering the equality predicate in some join accelerators [43]. However, filtering can get very demanding, such as with logic expressions of 24 DNF clauses in a popular decision support benchmark [33], [87].

Manev et al. [33] proposed a scalable architecture to enable building arbitrary filtering accelerators from boolean logic as found in SQL queries. The evaluated accelerator is for TPC-H's most complex filtering-heavy query [87]. The solution provides a stream manager so that each custom filtering module can access multiple dimensions simultaneously and achieve a high aggregate bandwidth with less engineering effort. This architecture is based on FPGAs that support partial reconfiguration, as a means to enable easy deployment of arbitrary filtering modules. The reconfigurability aspect was important since SQL is a programming language with an infinite number of possible routines, and it is shown that combining pre-build finer-grade filtering modules is less effective [88]. This finding is somewhat analogous to how today's CPUs have a performance overhead from using a pre-defined instruction set architecture (see appendix C).

Another accelerator designed to do filtering is a database processor from 2016 [89] designed for fast database analytics and is implemented on an FPGA. This work takes advantage of a highly-coupled SoC architecture that provides high-throughput communication between two ARM cores and the FPGA, residing on the same die (Altera Arria V SoC FPGA). In contrast with other accelerators, it focuses on small and embedded devices. The main idea was to use a content addressable memory (CAM), with which a series of queries can be executed in parallel but inside the chip. The queries are encoded in a format that resembles an instruction set architecture (ISA) as found in general purpose processors.

One more complex way of filtering is by using Skyline operators [90]. Skyline is a family of queries that filter multi-dimensional data using multiple criteria. This requires streaming the same data multiple times and it can be expensive for CPUs. Centaur [26] and DoppioDB [48] explore an FPGA implementation of a skyline query engine and demonstrate a notable performance improvement over CPUs.

Overall, the main consideration seems to be the ease of which arbitrary problems are supported. The basic approach is to build accelerators for specific queries, including the skyline examples above [26], [48] (even though they are integrated into frameworks). Another approach is to build accelerators from generators based on code [91], [92]. Finally, there are solutions that can load

dynamically a library of provided bitstreams through partial reconfiguration during runtime [33], [88], [93]. Still, being based on specialisation introduces some limitations. Although each case can be further customised and the probability of exceeding hard-wired limitations may be minimised, CPUs are still more suitable when the characteristics of the workloads are unknown.

2.4.4 Distinct count

A common task in data analytics is to count the number of distinct elements in a list, also known as the cardinality problem. It can be considered a special case of the *set membership problem*, i.e. to be able to say if an element exists in a set, because such a question can be asked for each element to answer how many unique elements have been observed.

In relational databases, an integral feature of SQL, which is the prominent domain-specific language, is the ‘DISTINCT’ clause. The ‘DISTINCT’ clause can be found within the ‘COUNT()’ function to express the distinct count functionality. It also has applications in computer architecture. For instance, hardware implementations of bloom filters have been demonstrated to work effectively on CPU caches for improving the cache block replacement decisions [94].

Set membership has applications in many fields and it is seen as computationally expensive [95]. The worst case for run time and space complexity is when the entire input consists of unique elements. Thus, the space and time complexity for a serial processor of an accurate implementation could be $O(N)$ and $O(N^2)$ respectively, where N is the number of elements. This example is for a naive implementation where all elements are stored and checked one-by-one per membership query. Consequently, there are applications where this operation becomes challenging, as with Terabytes of “big data” as input in the datacenter, or even seemingly endless streams, such as the huge amounts of data flowing inside a CPU. For this reason, approximation techniques are used widely in data science, but always taking into account how the inaccuracies of the model relate to the nature of the problem and the features of the data.

Such an approximation technique is the Bloom filter, which combines multiple hash functions in a relatively small hash table achieving a low false positive rate [96]. False negatives are

impossible. There are also hardware-optimised implementations of Bloom filters [97], [98]. For big data analytics and distinct count the storage is still a consideration, especially for the limited amounts of on-chip memory on the FPGA, as it cannot count more elements than the available space. For distinct count specifically, LogLog [99] and its successor HyperLogLog [100] are two popular algorithms to perform the task with a fraction of the resources, and with FPGA implementations achieving a speedup of up to around 2 times over an optimised software implementation on a 32-thread Xeon server [101].

Even with a reduced probability of mistakes [102], approximation techniques on FPGAs [101] are usually seen stand-alone, outside the database management system (DBMS) scope. This is most likely because an exact approach is more appropriate for legacy applications (conforming with SQL), as well as for problem specifications where there is no margin for errors.

With respect to exact solutions for distinct count, N. Bandi et al. [103] have used fully-associative structures to answer membership queries on ASICs, such as for processing network traffic. Absalyamov et al. [29] have built on the ideas of content addressable tables (CAT) in a database context, for building a custom cache hierarchy to accelerate ‘GROUP BY’ aggregations. These approaches are analogous to hash joins, such as with the latter also using multi-threading on the FPGA to saturate the main memory bandwidth [27]. This is due to the random memory accesses that require frequent lookups from main memory [29].

Sections 5.3 and 5.4 present alternative approaches for data of high cardinality. One of the proposed approaches sorts the data first (analogous to sort-merge joins) and then processes it on the fly using non-blocking high-throughput stream processors, leading to a more deterministic performance, due to the linearised access pattern.

2.4.5 Arithmetic

Other accelerators that would help increase modern database systems are a set of operators that are related to big data analytics and machine learning or statistics. The list could be very extensive, due to the many different use-cases and therefore only presented are some that were

seen in a database acceleration context.

Simpler examples such as “muladd” (scale one column by a constant and add another constant to each element) and “percentage” (the percentage of the sum of a subsequence of a column over the sum of all elements) and “testcount” (count all rows satisfying a common condition) are included in the exploration of the Centaur framework [26]. CPUs can also be efficient with such operations, especially when the architecture supports vectorisation, and the compiler or the programmer identifies the appropriate vector instructions for the task.

Note that for some arithmetic operators the performance of the FPGA may not be ideal, but they are sometimes supported as part of a database framework to eliminate the data transfers in heterogeneous systems. For instance, simple arithmetic operators can be used as part of a pipeline of different database operators, eliminating the need to transfer data to the host for intermediate tasks, which is a costly operation.

A more complex accelerator is a stochastic gradient descent (SGD) implementation [49]. SGD is used for building linear models in machine learning and is an iterative operation, until a set of stopping conditions are met. Through quantization, the operation can be optimised for different accuracy needs. It is shown to outperform a high-end CPU implementation by an order of magnitude for the same level of accuracy, despite the memory bandwidth limitation [49].

2.5 Alternative and future architectures

2.5.1 Other co-processors

Graphics Processing Units (GPUs) are currently a competitor to FPGAs for database acceleration. There are similar disadvantages to FPGAs with respect to the communication limitations, as they utilise similar protocols and layers (see 2.5.4). They also suffer from the main bottleneck of FPGAs when doing memory accesses, which is the limited memory bandwidth.

There are specialised servers utilising customised protocols on top of PCIe, or separate interconnects. NVidia GPUs introduced the NVlink protocol [104] to achieve a main memory access speed of 25 GB/s per direction. Additionally, an NVlink switch can offer 300 GB/s theoretical interconnect bandwidth, with GPUs connected through the SXM2 form factor. Still, the peer-to-peer (GPU-to-GPU) unidirectional throughput is 48 GB/s and host-to-device bandwidth of 12 GB/s, for V100 GPUs utilising NVLink 2.0 [105].

Roozmeh et al. [24] compared high-end FPGAs with GPUs and found that FPGAs can actually be faster than GPUs for some database operators. Both the FPGA platforms were 40% more energy efficient than the GPUs. However, this work highlights that the high-end FPGA test platform (Virtex UltraScale VU440) that was the best performance-wise in comparison with the GPUs, costed two orders of magnitude more money. This is not the case today, especially after the launch of Xilinx Alveo cards, which are relatively affordable but still more expensive than GPUs.

While GPUs are not as flexible and power efficient as FPGA, there are many beneficial aspects as well. Modern GPUs support numerous abstractions by default that are generally challenging to implement on FPGAs. These include L1 and L2 caches, and a shared address space of a pool memory with coherency mechanisms [106].

2.5.2 High-performance multi-core processors

Using a high-performance multi-core processor is an important competing approach to accelerating using a co-processor such as an FPGA. This is due to the relative ease of use, the out-of-the-box high memory bandwidth and seamless integration with mainstream software that comes with CPUs. The move from multi-cores to many-cores may be successful for database workloads and may overcome the need for an FPGA, at least for performance.

There is a recent trend of increasing the core-count of conventional CPUs, such as the latest AMD EPYC server offerings, reaching up to 128 threads per server socket [107]. Currently, the highest thread-count for commodity multi-processors are found in AMD EPYC 7763, which has

64 cores with SMT enabled (128 threads in total) and also Intel Xeon Platinum 8380, which has a total of 40 cores, and a total of 80 threads [108]. They are also designed to support multi-socket configurations which can quadruple or quintuple, for instance, the core count in a system.

However, the increased number of cores does not come for free. While they can be convenient and have better memory bandwidth for databases, the complexity of the cores can yield much lower power efficiency or cost effectiveness in comparison with accelerators. Example limitations include Ahmdal’s law (how parallelisable is the accelerated task), the increased complexity of the interconnects (for memory arbitration between more workers), higher latencies and simpler cores. Another potential disadvantage of high-end multi-cores is that they rely on the non-uniform memory access (NUMA) model, although less apparent than in FPGAs. Moreover, the single-thread performance is rather slow when compared to workstation CPUs.

Additionally, developing for such a high core count is not that straightforward, with scientific applications sometimes relying on manual acceleration with SIMD intrinsics [5]. The available main memory bandwidth is higher than on FPGAs, but it is challenging to achieve memory-bound execution. See an example experiment with AVX-512 intrinsics (512-bit-wide SIMD instructions [109] as found in high-end Xeon processors [108]) in appendix A based on section 3.6, where the benefit of AVX-512 over AVX2 (256-bit) is questionable.

2.5.3 Many-core processors

Many-core processors, such as Intel’s Xeon Phi, could be considered another competitor to FPGAs. As a co-processor, Xeon Phi suffers from similar disadvantages as FPGAs for databases. Cheng et al. [110] have optimised state-of-the-art hash join algorithms for better utilisation of the resources of the latest Xeon Phis. The local high-bandwidth multi-channel DRAM is found useful for implementing a big cache for the main memory. Their work combined software prefetching and NUMA-aware partitioning, and concluded that Xeon Phis can give an important speedup for hash joins. An FPGA equivalent could be considered the hash join accelerator by Halstead et al. [27].

Some more experimental many-core architectures leave more control to the programmer. This is to achieve less hardware complexity, as well as flexibility for increased performance [111]. They can support direct inter-core communication which can be used for pipelining operations [112]. This is something very desirable in database queries [26] and it is not supported in mainstream CPUs. Though, for such innovative architectures, the need for specialised toolchains, as well as the use of uncommon programming models, has often led to very limited adoption. Of course, there have been many useful outcomes for research purposes from such projects [113].

Generally, modern many-cores are used as co-processors and can have similar bandwidth limitations to GPUs and FPGAs. As with multi-cores, an important limitation is that with an increased number of cores the main memory throughput each core can achieve will be restricted. Having many cores can also complicate the memory system and increase the memory access latency. The increased latency can occur from sharing common resources [13] including shared caches, or from hierarchical architectures which are sensitive to data localities, such as with non-uniform memory-access (NUMA) systems. Advanced CGRA-like techniques for task-pipelining are not possible on general purpose CPUs, but it is possible with some experimental architectures [111]. Also, too advanced cores, or with unused features in general (such as with high-performance multi-cores) will be less efficient than an FPGA. But, the ease of programming and an adequately-efficient implementation in commodity systems might outweigh these shortcomings for a wider adoption.

As of today, many-cores deployed as co-processors seem to have been phased-out in favour of many-cores as main processors (as in the preceding discussion in section 2.5.2 about multi-cores). A notable example is the end-of-life of the latest Intel Xeon Phi [114], as well as the wider inclusion of AVX-512 in Intel's general purpose processors [108]. Instead, the general trend for many-cores seems to be the adoption of simpler cores, such as the RISC-based (reduced instruction set computer) and the ARM A64FX processor in the Fugaku supercomputer [115]. To the best of the author's knowledge, no extensive evaluation of database applications in RISC-based many-core systems has been performed yet.

2.5.4 Faster interconnection protocols

One consideration with today's FPGAs is that they are connected with PCIe to a host system. PCIe, the most widely-used interconnection protocol for accelerators, has a high access latency [116], and is not appropriate for low-latency applications. Another shortcoming of PCIe is the bandwidth.

Today's commercial high-end FPGAs, such as Xilinx Alveo cards, which are also promoted for datacenter analytics, use up to 16 lanes of PCIe 3.0, or up to 8 lanes of PCIe 4.0 [117], which are both equal to 32 GB/s bidirectional bandwidth for 16 lanes. This number can be seen as a theoretical maximum throughput, and is lower than CPU's main memory bandwidth, such as up to 204.8 GB/s for the latest AMD EPYC multi-processors [107]. In industrial settings, using more PCIe lanes is possible but this approach has not matured.

Moussalli [118] discussed the future directions for accelerating databases on GPUs and FPGAs in 2017, and suggested that various forthcoming technologies could minimise the memory bandwidth limitation of accelerators. Among NVlink [104], which is for GPUs, the IBM POWER8 processor, with its Coherent Accelerator Processor Interface (CAPI) was also mentioned for faster database acceleration.

CAPI promises higher memory bandwidth from the accelerator (including FPGAs) by eliminating software abstractions for better bandwidth utilisation. This is implemented on top of PCIe Gen. 3 and has a bidirectional memory bandwidth of 32 GB/s [119] (16 lanes of 8 Gbps for read and write). Later, IBM introduced the POWER9 architecture, which has up to 48 PCIe lanes and OpenCAPI, the successor to CAPI, which can yield performance of 150 GB/s aggregate throughput (48 lanes of 25 Gbps each) [120]. IBM's CAPI 2.0 [121] is its successor that utilises PCIe 4.0 lanes and upgrades this number to 192 GB/s. Such protocols are expected to improve the FPGA performance for database operations, including multi-way joins [122].

Note that these accelerator-focused protocols are mostly intended for implementing coherency, as well as with the cache-coherent interconnect for accelerators (CCIX [123]). Although higher integration is important, the lane count is not a tangible benefit of architectures featuring

protocols such as CAPI (yet), unless all lanes are managed to be utilised. It is worthwhile speaking about aggregating PCIe lanes, however, the practicality of an increased lane count remains questionable for the studied use cases as the latest commercial FPGAs use up to 16 lanes, as with GPUs. Unconventional system architectures are generally expensive and slow or difficult to adopt. For example, practical applications are still not easily benefiting from such protocols as CAPI 2.0, including using a lower number of lanes in legacy mode for a 13 GB/s achievable throughput [124]. Moreover, x86 server processors have already increased their PCIe lane count to 128 [107].

Promising advancements in FPGAs revolve around System-On-Chip (SoC) packages that combine different kinds of processors. One such platform is Xilinx’s multiprocessor systems-on-chip (MPSoC) [9], [125] that combines an FPGA with ARM CPU and GPU cores. This could be a pivotal point for heterogeneous workloads, but in regards the bandwidth performance they are still relatively low-end [77], [89].

One recently emerging FPGA architecture is Versal [126], which is a 7nm SoC, that increases heterogeneity but also in finer-grain, such as by hardening commonly used logic. The goal behind these decisions was to reduce the place-and-route complexity, as well as to increase the performance of trending workloads (machine learning and communications). However, due to the embedded nature of these devices and the increased specialisation, they can be seen as quickly drifting away from large-scale deployments for big data.

An interesting platform for databases is Intel’s Xeon+FPGA prototypes [26]. HARP v1 allows an FPGA to co-exist in a second socket in a multi-socket server. While this might not be an important change over FPGAs connected with PCIe in high-performance servers, it could be considered as an early step for the introduction of higher-performing Xeon cores with FPGAs [127]. This would hopefully translate to the highest main memory bandwidth for FPGAs. Later, another experimental Intel Xeon+FPGA solution combined the Aria 10 FPGA and a 14-core Xeon on the same package. Still, the FPGA in this solution only achieves 20 GB/s of aggregate main memory throughput through the CPU’s memory controller, while the Broadwell E5 CPU peaks at 50 GB/s [25].

2.5.5 Near-data processing/ processing in memory

In the datacenter scale, near-data processing (NDP) is promising to overcome the power and performance implications of processing huge amounts of data. By moving computation to memory, NDP avoids chip-to-chip transfers and yields efficient in-memory computation. Due to the recent advances in 3D-stacking technology, NDP is predicted to gain more attention as a solution [128].

More related to FPGAs and databases [118], ConTutto was an experimental platform to bring FPGAs close to the memory by incorporating them with DRAM in the form of DIMM cartridges [129]. There are working prototypes of such DIMMs and plug into an IBM POWER8-based server. Acting as a memory buffer, the FPGA can have a memory access bandwidth of 11 GB/s per DIMM port, or an aggregate of 35 GB/s for a single memory channel.

The idea of near-data processing has matured to the point that consumer solid-state disks (SSDs) can be purchased, while also integrating a reprogrammable device. For instance, Samsung's SmartSSD technology brings FPGAs closer to the data, by embedding FPGAs on SSDs and enabling distributed computing for an increased aggregate bandwidth [130]. The Samsung SmartSSD computational storage drive (CSD) combines a 4 TB SSD with a Xilinx Kintex KU15P FPGA (with over a million logic cells), and is already evaluated for sorting [17], [82] and database queries [130].

2.6 Summary

FPGAs are an interesting platform for hardware acceleration of relational databases. They can be superior to other processors in terms of power consumption, but achieving a high performance has its challenges. It is often possible to achieve higher performance than CPUs, but this can require expensive and complicated designs. When evaluated in an end-to-end database solution, the achieved speedups from using FPGAs over CPUs are usually reported to be single-digit.

The main bottleneck is the limited main memory bandwidth found in current hardware, which

is around 32 GB/s bidirectional for FPGAs, and is low in comparison with that of CPUs, reaching over 200 GB/s. The trend in inter-chip communications is to gradually increase the bandwidth, such as with approximately doubling the throughput per lane per newer PCIe generation, but still not approaching CPU levels. The alternative approach to develop high-end SoC platforms such as Intel's Xeon+FPGA is more promising, but experimental results have still not approached the throughput of the CPU cores for the FPGA device.

With respect to database acceleration frameworks, since a database can cover a wide range of operators and their combinations, the specialisation aspect of FPGAs does not favour general use. For instance, SQL is a programming language that can express tasks of arbitrary complexity, including recursion. For this reason, the existing database accelerator frameworks are either accelerating a database system of restricted capabilities and/or scale, or providing means to accelerate user-specified operators in heterogeneous systems.

Another category of research is on accelerating individual operators, with applications wider than only in databases, such as with sorting. The evaluation results of operator accelerators are usually more promising, as it is done out of the DBMS context and can be more specialised. There are also clever algorithms or modifications that make better usage of the available memory throughput, such as by caching entries, compression, improving the inter-chip communication protocol and exploiting other data localities.

Future systems with a higher degree of integration and less heterogeneity might address the aforementioned challenges at a considerable degree.

Chapter 3

FLiMS: a fast lightweight 2-way merger

This chapter presents FLiMS [36], a highly-efficient and simple parallel algorithm for merging two sorted lists residing in banked or multi-ported memory. On FPGAs, its implementation uses less hardware resources than the studied alternatives. This is achieved with better performance and less latency, for the same amount of parallelism. FLiMS is also shown to perform well as conventional software on modern CPUs supporting single-instruction multiple-data (SIMD) instructions.

The challenges for the merge operations on FPGAs have been the low clock frequency due to the feedback datapath of the merger contributing to the critical path, and also the high resource utilisation in some early attempts to eliminate and remove the feedback datapath. FLiMS uses a modified version of the bitonic merge block, as found in bitonic sorters, repurposed for performing 2-way parallel merge for streaming data. As with some alternative solutions, it can be considered feedback-less since it only nests one parallel comparison for any desired level of parallelism (i.e. no pipeline stage receives data from the output of a successor stage).

At the time of writing this thesis, it still uses the least number of comparators and pipeline stages among the latest mergers, and allows higher throughput, while leaving more hardware resources for additional logic.

The merge operation is widely used for a variety of applications, including in many popular

sorting algorithms, such as mergesort and timsort [131], as well as in hardware for database operations, including sort-merge joins. It is thus desirable to accelerate, and recent research has proposed a variety of merge accelerators on FPGAs, each of them having their own shortcomings in comparison with the others.

The improvement over the state-of-the-art is on the computational aspect and hardware utilisation. The approach is initially evaluated out-of-context, with the data to be processed already residing locally, such as in an FPGA’s block RAM. While this is useful by itself, it can also be used as a building block for bandwidth optimised designs, since the proposed block RAM organisation allows it to be used for big data streaming situations, as they can be used as FIFO queues efficiently (see section 5.1).

The bitonic sorter is a sorting network based on merge-sort. Since it is based on merge-sort, it has a hierarchical merge block, labelled *bitonic merger* in figure 2.6 of chapter 2. Each merge block merges two sorted lists of w elements to a sorted list of $2w$ elements. The *bitonic partial merger* is a subset of the bitonic merger and is able to provide the upper half of the result. This “pruned” merge block is combined with additional logic to work as a parallel merger for longer lists as streams.

All alternative designs require the two input sequences of the bitonic (partial) merger (or odd-even merge sort-based mergers) to be sorted. The **novel idea** is to relax this condition. This idea eliminates the need for expensive rotations before the inputs of the merger [69], or alternative workarounds involving redundancy in logic [35], [74].

The contributions are the highly-efficient design of the merger and the evaluation in both on FPGAs and modern CPUs with SIMD units. Automatic generator scripts [132] provide the Verilog code for the FPGA implementation, as well as with SIMD intrinsics for CPUs, for a user-specified degree of parallelism w . Also presented are additional functionalities for FLiMS, such as stable sorting, and the implementation of a complete sorting function in SIMD-based software. Chapter 5 presents more hardware designs that make use of FLiMS to achieve FPGA acceleration of different analytics operations, including a full-system hardware sorter.

This chapter is partly based on the papers “FLiMS: Fast Lightweight Merge Sorter” [36] and “An Adaptable High-Throughput FPGA Merge Sorter for Accelerating Database Analytics” [32]. The presented material is the author’s, except with proof 3.3.1(2), which was done in collaboration with Chris Brooks.

3.1 A novel 2-way high-throughput merger

FLiMS is a novel 2-way high-throughput merger that only uses a single $2w$ -to- w bitonic partial merger. It merges 2 sorted inputs with high throughput without the use of barrel shifters or shift registers.

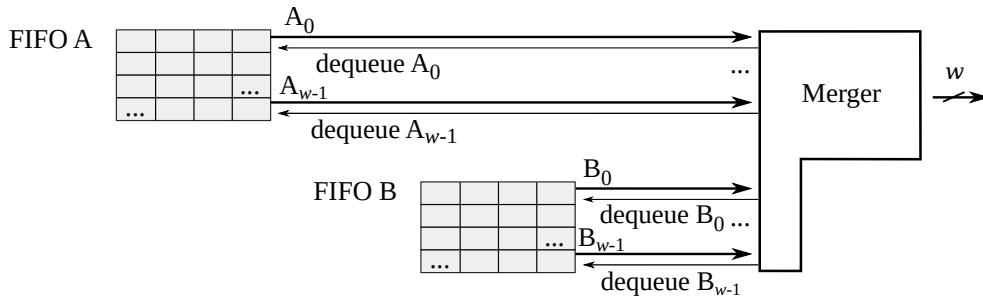


Figure 3.1: High-level architecture of FLiMS [36]

In figure 3.1, we can see a high-level visualisation of the proposed parallel merger. In contrast to previous works, FLiMS is shown to dequeue from the banks on an individual basis, rather than in batches of w elements, due to the decentralised selector logic.

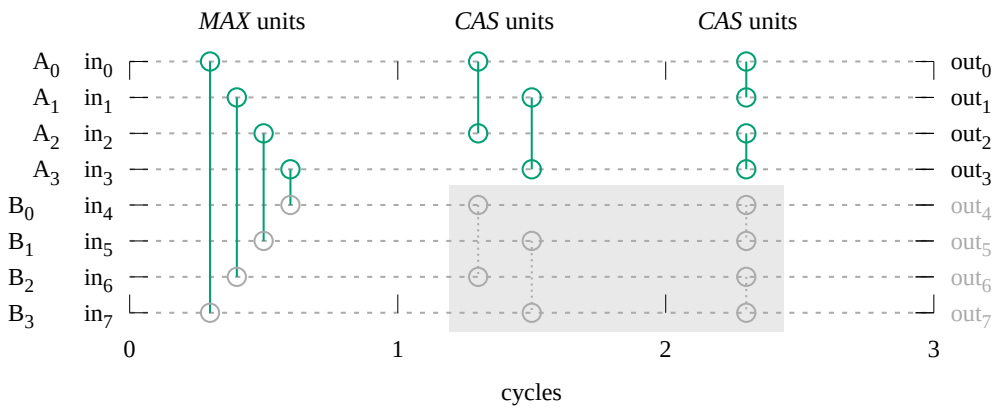


Figure 3.2: FLiMS in low-level: *MAX* selector stage plus a *CAS* network (butterfly topology), $w = 4$. Pruned *CAS* units are shown in grey

In figure 3.2, we see a lower-level representation, where the green circle pairs are the compare-and-swap units, with the exception of the first pipeline stage, which has w pairs of one green and one grey circle, representing the *MAX* units (selector stage). The greyed pairs are the pruned units when compared to the original bitonic merger, which would relate to sorting the bottom half. If we ignore the modification of the first stage to MAX units, this topology is known as a bitonic partial merger [71], because it originally produces the top w out of two sorted lists of w elements, and is a subset of the bitonic merger (see section 2.3.3).

The proposed parallel algorithm can be broken down into 3 segments: a *selector stage* for handling the input from multiple banks/queues, the pipelined *bitonic partial merger* (minus the first stage) and finally the *output logic*.

Selector stage After the data are written into the BRAM banks (stored with a round-robin priority), a set of independent entities (i.e. nodes) are responsible for controlling the input to the merger. The w entities respond to the same clock and their behaviour can be described by a distributed algorithm.

Let A and B be the input FIFO queues containing the two sorted lists to be merged. Let $A_0, \dots, A_{w-1}$ and $B_0, \dots, B_{w-1}$ be the queues corresponding to the respective banks for A and B . Each of the entities $MAX_0, MAX_1, \dots, MAX_{w-1}$ have as input the pair of queue heads $(a_0, b_{w-1}), (a_1, b_{w-2}), \dots, (a_{w-1}, b_0)$ respectively, with a_i being the head of A_i and b_i being the head of B_i . Each of these entities outputs one number per cycle when both inputs are valid.

Each entity MAX_i has the data registers cA_i and cB_i to store the last heads that were dequeued from banks A_i and B_{w-1-i} respectively. It also has a register for the sorting network input in_i . On each cycle with valid input, if $cA_i > cB_i$, then cA_i will make it into the top w in the result of the pipeline and therefore cA_i is copied into in_i . In this case, cA_i is replaced by the head a_i , which is dequeued from A_i , but cB_i will remain unchanged, since it will need to be compared again in the next cycle, being in the lower w . The equivalent logic goes for the case when $cA_i \leq cB_i$.

In algorithm 2, we can see the pseudocode for the distributed algorithm. Collectively, this

algorithm essentially replaces the first stage of the partial bitonic merge (half-cleaner), with MAX units instead of CAS units. It selects the current top w on each cycle and inserts them into pipeline registers for the rest of the CAS network to sort and produce the correct w -sized chunk of output.

```

1 int  $i$ ;
2 int  $cA_i, cB_i, in_i$ ;
3 while forever do
4   receive (positive clock edge);
5   if  $cA_i > cB_i$  then
6      $in_i \leftarrow cA_i$ ;
7      $cA_i \leftarrow \text{dequeue}(a_i)$ ;
8   else
9      $in_i \leftarrow cB_i$ ;
10     $cB_i \leftarrow \text{dequeue}(b_{w-i})$ ;
11  end
12 end

```

▷ i is the entity tag
▷ registers of data width

Algorithm 2: MAX_i unit pseudocode

There may be a need for some extra logic required to correctly handle the ending of the input queues, but it is omitted here for the sake of simplicity and portability to different architectures, as it is relatively trivial to construct. For example, when sorting natural numbers in descending order, the value 0 can be passed afterwards to handle the ending without additional dedicated logic.

CAS network The compare-and-swap (CAS) network of FLiMS is responsible for sorting the top- w result of the selector stage. It is a partial bitonic merger minus the first stage, or in other words, a butterfly network. It is not considered a sorting network on its own and does not sort arbitrary number sequences. For its input though, it behaves correctly and its output is always sorted (see proof 3.3.1).

Output logic The output logic is trivial, but is mentioned for the sake of completeness. On each cycle, if the output of the partial bitonic merger is marked as valid, it is written down as a w -sized chunk of the result, such as in w output banks implementing an output queue O . The output can then be read in a round robin fashion from the banks O_0 to O_{w-1} , for example.

One observation is that when there is valid output per cycle, it produces exactly w elements, as with other 2-way mergers. This is useful for easing synchronisation when embedding into a merge tree etc., as in section 5.1.

The sets of registers cA and cB are not visible in the first (selector) pipeline stage of figure 3.1. The notion of those registers is optional because they can also be considered the current sets of heads of the banked queues A and B respectively. Though, it is sometimes convenient to use cA and cB, such as when the input queues are block RAM sections, where a read register is already present for reading each memory. In table 3.1, an example execution is provided.

Cycle	Input A	Input B	cA	cB	Output (<i>after the pipeline delay</i>)
0	3 3 4 5 11 16 17 26 26 29	0 7 8 9 12 15 18 19 21 22			
1	3 3 4 5 11 16	0 7 8 9 12 15	17 26 26 29 22 21 19 18		22 26 26 29
2	3 3 4	0 7 8 9 12	17 5 11 16 15 21 19 18		17 18 19 21 22 26 26 29
3	3 3	0 7	4 5 11 16 15 12 9 8		11 12 15 16 17 18 19 21 22 26 26 29
4			4 5 3 3 7 0 9 8		5 7 8 9 11 12 15 16 17 18 19 21 22 26 26 29
5			4 3 3 0	0 3 3 4 5 7 8 9 11 12 15 16 17 18 19 21 22 26 26 29	

Table 3.1: Merging two descending sequences: Example execution for $w=4$ and random sorted lists A and B. (see [133] for an online visualisation)

3.2 Additional functionality

This section presents two extensions for FLiMS, which could be used to increase its applicability or performance, according to the distribution of the data and the requirements of the sorting problem.

3.2.1 Skewness optimisation

FLiMS can be used to build parallel merge trees (PMT), that merge many input lists hierarchically in a single pass. Parallel merge trees can suffer from rate mismatch, that happens when the input data distribution leads to underutilisation of certain mergers, resulting in reduced throughput [42], [69], [76].

While the memory access throughput is a matter of the memory system, it might be allowed

for the accelerator to receive the data from each of the input lists A and B with a fixed bandwidth, less than w , such as in a PMT. A contributor to rate mismatch is when there are a lot of duplicates in the input (skewed datasets). When the data are skewed, the merger only dequeues from one of the input queues for long periods of time. This results in stalled cycles from underutilising the aggregate bandwidth of the queues.

PMT [69] proposes a simple solution which causes the merger blocks to fetch from both inputs at a similar rate when there are duplicates. This ensures that the input queues are consumed with a similar rate, that collectively balances the utilisation of the merge tree. However, PMT’s mergers inherit the long feedback problem, which was addressed in subsequent works [35], [36], [73], [74].

The equivalent optimisation is proposed for FLiMS [36], while keeping the decentralised nature of the selector stage. On duplicates, there is an “oscillating” effect at the MAX units, which balances the dequeuing rate from the two groups of inputs. The code for the new selector units is illustrated in algorithm 3. An 1-bit register called $side_{Last}$ represents the queue out of which the head was dequeued during the previous cycle, and is appended to the least significant bit in the comparison, to enforce a sort priority on equal values.

```

1  int  $i$ ;
2  reg  $cA_i, cB_i, in_i$ ;
3  reg  $side_{Last}$ ;
4  while forever do
5      receive (positive clock edge);
6      if  $\{cA_i, side_{Last}\} > \{cB_i, !side_{Last}\}$  then
7           $in_i \leftarrow cA_i$ ;
8           $cA_i \leftarrow \text{dequeue}(A_i)$ ;
9           $side_{Last} \leftarrow 0$ ;
10     else
11          $in_i \leftarrow cB_i$ ;
12          $cB_i \leftarrow \text{dequeue}(B_{w-1-i})$ ;
13          $side_{Last} \leftarrow 1$ ;
14     end
15 end
```

▷ i is the entity tag

▷ registers of data width

▷ 1-bit register

Algorithm 3: Modified MAX_i unit pseudocode for skewness optimisation

Another occurrence of rate mismatch is when the data are already (or nearly) sorted, and this should be solved at a higher level, such as by shuffling the input as a preprocessing step [69].

The skewness optimisation does not solve that case of rate mismatch.

3.2.2 Stable merge

Stable sort is when the sorted output has the same order for duplicate values as they appear in the input. For implementing a stable merge sort, FLiMS would also need to be stable, i.e. to prioritise the duplicates of input A over the ones at input B, and keep their original order inside A and B accordingly. Such a modification cannot co-exist with the skewness optimisation, since duplicates will have priority based on the input source.

Implementing stable sort using FLiMS follows a similar methodology of adding functionality to sorting networks as in section 4.2.3, but requires further modification. Originally, FLiMS is not stable, as it is partly-based on the bitonic sorter, which is not stable. Temporarily appending the input source (1 bit) and the port number ($\log_2(w)$ bits) to the MSB would be required to disambiguate between the original order of duplicates inside the CAS network.

As the order of the inputs inside MAX units are naturally rotated by an offset, the port order is not enough to distinguish the order inside each batch containing duplicates. For this reason, a 2-bit value needs to be carried between the input source and the port number, that keeps count of the batch order. This has a resemblance to the design of section 5.1.1, where a single-bit counter is not enough for distinguishing which of the two compared entries came first. Algorithm 4 shows the modifications required on the *MAX* units to support stable merge.

Additionally, the *CAS* units would need to be modified as well, to correctly prioritise the case where the 2-bit order is “00” against “11”, as all other combinations (same values or other pairs having a difference of one) would correctly represent the original order priorities. The general idea of this approach is to emulate appending the original input order to the MSB of the data, but with a steady and low number of bits for merging arbitrarily long input. The count field can be seen as the last few bits of the input order.

```

1 int  $i$ ;                                ▷  $i$  is the entity tag,  $\log_2(w)$ -bits wide
2 reg  $cA_i, cB_i, in_i$ ;                  ▷ registers of data width
3 reg  $orderA_i, orderB_i$ ;                ▷ 2-bit registers
4 while forever do
5     receive (positive clock edge);
6     if  $cA_i > cB_i \parallel cA_i == cB_i$  then
7          $in_i \leftarrow \{1, orderA_i, w - 1 - i, cA_i\}$ ;    ▷  $0, orderA_i, i$  for ascending order
8          $cA_i \leftarrow \text{dequeue}(A_i)$ ;
9          $orderA_i \leftarrow orderA_i - 1$ ;                ▷ +1 for ascending order
10    else
11         $in_i \leftarrow \{0, orderB_i, i, cB_i\}$ ;          ▷  $1, orderB_i, w - 1 - i$  for ascending order
12         $cB_i \leftarrow \text{dequeue}(B_{w-1-i})$ ;
13         $orderB_i \leftarrow orderB_i - 1$ ;                ▷ +1 for ascending order
14    end
15 end

```

Algorithm 4: Modified MAX_i unit pseudocode for implementing stable merge

3.3 Correctness

In order to understand why FLiMS and its variations work as expected, a series of inductive or direct proofs are provided. FLiMS can be used for merging lists in descending order, as well as for ascending order, just by reversing all comparators (for the base version). In this chapter we study FLiMS designs with data of descending order, but the proofs can be easily adapted for the other case, without loss of generality.

3.3.1 Basic version

In order to prove that the main design behaves as expected we will show that it is functionally equivalent to a basic merger implementation, where on each cycle the input comes sorted to a $2w$ -to- w bitonic partial merger, as with the merger used in PMT [69]. We will show that (1) the selector stage algorithm always selects the top w out of the $2w$ -sized input and that (2) the butterfly network always sorts this top w list before writing the result in output.

(1) We denote as $l_A, l_B \in \{0, 1, \dots, w - 1\}$ the naturally-occurring rotation offsets for input queues A and B respectively. Supposedly, the selector stage dequeues consecutive elements from each banked input. Since A and B are written in a round robin fashion inside the banked memory, there are different rotation offsets that would be required to read a sorted set of w

elements form each set of FIFOs, at least to correspond to a design that rotates its inputs as in PMT [69]. Note that the MAX units in FLiMS receive elements from the corresponding banks without performing any additional rotation.

Induction hypothesis: On each cycle, if the selector stage has worked correctly on the previous cycle, it will load $k \in \{0, 1, \dots, w\}$ elements from A and $w - k$ from B, collectively corresponding to the combined top w out of the available $2w$ elements.

Base step: At cycle 0, the parallel merger behaves in the same way as the $2w$ -to- w bitonic partial merger, because the w inputs from each list are already sorted ($l_A = 0$ and $l_B = 0$), corresponding to the first w elements from A and first w from B (stored values in cA and cB registers respectively). The first stage of the $2w$ -to- w bitonic partial merger is known as a half-cleaner that produces the top w out of the $2w$ inputs, in a bitonic sequence.

Induction step: Given that the selector stage worked correctly in the previous cycle, $k' \in \{0, 1, \dots, w\}$ and $w - k'$ are the amounts of sorted elements dequeued last from A and B respectively.

On each cycle, l_A and l_B will be updated according to the number of dequeued heads, as the starting positions shall succeed the dequeued elements. That is, $l_A = (l'_A + k') \bmod w$ for A, and $l_B = (l'_B + (w - k')) \bmod w$ for B, where l'_A, l'_B the offsets of the previous cycle. Therefore,

$$\begin{aligned}
 l_B &= (l'_B + (w - k')) \bmod w \\
 \Rightarrow l_B &= (l'_B + (w - (l_A - l'_A))) \bmod w \\
 \Rightarrow l_B &= (l'_B - l_A + l'_A) \bmod w \\
 \Rightarrow (l_B + l_A) \bmod w &= (l'_B + l'_A) \bmod w
 \end{aligned}$$

As cycle 0 assumes $l'_A = 0$ and $l'_B = 0$ (due to the input already being aligned correctly in the banks), and that the induction hypothesis is assumed correct for all previous cycles, it always holds that $(l'_B + l'_A) \bmod w = 0$. Since $l'_B = l'_B \bmod w$, it holds that $l_B = (w - l_A) \bmod w$ for every cycle.

In order for the selector stage to produce the top w out of the current $2w$ elements, the comparisons that need to be made are between all pairs in $\{(Ta_i, Tb_{w-1-i}) | \forall i \in \{0, 1, \dots, w-1\}\}$, where $Ta_0, \dots, Ta_{w-1}$ are the current top w elements in A and $Tb_0, \dots, Tb_{w-1}$ are the current top w elements in B, sorted in descending order. These comparisons are required in order to emulate the bitonic partial merger, which performs this task in its first stage (half-cleaner).

The *MAX* units have the same topology as the bitonic (partial) merger (as seen in figure 2.6). We notice that whatever the rotation combination from l_A and l_B is, the correct comparisons will be made, as:

$$\begin{aligned}
 & (Ta_{(i+l_A) \bmod w}, Tb_{((w-1-i)+l_B) \bmod w}) \\
 \equiv & (Ta_{(i+l_A) \bmod w}, Tb_{((w-1-i)+(w-l_A)) \bmod w}) \\
 \equiv & (Ta_{(i+l_A) \bmod w}, Tb_{(w-1-i-l_A) \bmod w}) \\
 \equiv & (Ta_{(i+l_A) \bmod w}, Tb_{(w-1-(i+l_A)) \bmod w}).
 \end{aligned}$$

(2) The sorting network receives the correct top- w output from (1) and the task is to sort it correctly. Originally, this structure is supposed to sort bitonic sequences of size w [61]. The output of the distributed algorithm block is a rotated bitonic sequence, as we saw that the comparisons will be rotated by l_A . A rotated bitonic sequence is also a bitonic sequence [134] and therefore the input for the sorting network has the correct property, which completes the proof.

3.3.2 Including the skewness optimisation

In order to prove that FLiMS continues to sort correctly, the selector stage must be shown to still produce a bitonic sequence [36] (up to one local maximum and up to one local minimum).

Note that the order of consecutive duplicates from the same input is the one of the input, as it is considered already sorted. The bitonic sequence property needs to apply also on the order of each input when there are duplicates. Therefore, we need to show that there will still be up to

one local maximum and up to one local minimum in the bitonic sequence, even though there might be multiple additional entries with a value equal to the minimum or maximum. This ensures dequeuing consecutive entries from a and b , ensuring the integrity of the input data, as the data are stored in banks.

On each cycle, each of the w MAX_i units compares Ta_j to Tb_{w-1-j} , where i is a rotation of j by a common offset $o \in \{0, 1, \dots, w-1\}$ ($o = l_A = (w - l_B) \bmod w$), and Ta and Tb represent the remaining elements of the input queues in descending order. On the base version, this emulated a half-cleaner, that selects (and dequeues) a total of the greatest w elements from a total of $2w$ elements, generating a bitonic sequence.

Induction hypothesis: On each cycle, if the skewness-optimised selector stage worked correctly on the previous cycle, it will produce a bitonic sequence from k consecutive elements from A and $w - k$ consecutive elements from B, where $k \in \{0, 1, \dots, w\}$.

Base step: At cycle 0, the common offset is zero ($o = 0$), as the inputs are properly aligned inside the input FIFOs, and no element has been dequeued yet. Each MAX_i unit has an initial value of 0 stored in its $side_{Last}$ register. This means that on the event of comparing duplicates, it will behave as in the non-optimised case, where a single source (A) is preferred for duplicates, as the comparison of line 6 of algorithm 3 is now equivalent to line 5 of algorithm 2 (“if $cA_i > cB_i$ then”), which is already proven to produce a bitonic sequence from the proof of section 3.3.1.

Induction step: The skewness optimisation modification only takes effect where there are duplicates, i.e. $\exists h \in \{0, 1, \dots, w-1\} : Ta_h = Tb_{w-1-h}$. In these instances, we notice that duplicates only happen consecutively (including wrap-arounds from the natural rotation) and for the minimum value of the output, as $(Ta_0, Ta_1, \dots, Ta_{w-1})$ is monotonically decreasing and $(Tb_{w-1}, Tb_{w-2}, \dots, Tb_0)$ is monotonically increasing. As a consequence, the position of the minimum (split) in the bitonic sequence can be at the start, end or between this region of duplicates.

Given that FLiMS worked correctly on the last cycle, the $side_{Last}$ registers correspond to the last half-cleaner decisions, which will be of the form $\{1\}^{w-k}\{0\}^k, k \in \{0, 1, \dots, w\}$, after considering the offset o . The fact that ones and zeros appear only consecutively is because only

consecutive elements are dequeued from each input list on each cycle. Also, the sequence of $side_{Last}$ registers starts from 1, when we consider the current rotation offset. This is because the naturally-occurring rotation offset is updated according to the last position dequeued from A , the next of which corresponds to the first from list B , which yields the “first” 1 in the $side_{Last}$ sequence.

Therefore, the region of duplicates will be a sublist of the expression $\{1\}^{w-k}\{0\}^k$, with its 1s and 0s replaced by consecutive duplicates from A and B respectively. As a result, there will be up to one local minimum (split) in this region, and therefore up to one local minimum in the entire half-cleaner result, which is a bitonic sequence.

3.3.3 Including the stable merge functionality

In order to prove that the stable FLiMS works as expected, we are going to show that it behaves as a simpler version, where it is a basic version of FLiMS after appending the input order to the MSB of its input data. There are two lower-level building blocks in FLiMS that are modified: the MAX units and the CAS units.

With the proposed modification on the MAX_i units, the only difference with regard to the selection priority is the selection of head a_i in case of the head a_i being equal to the head b_{w-i-1} . The behaviour in the emulated case where the input order is appended to all input data would behave the same, as everything in list B is considered to be at a subsequent memory location, hence a lower input order than all elements in A .

With respect to the CAS units, the priority is now given first to the input source, which is correct because all elements in B are considered to succeed all elements in A . Then the next priority is about the 2-bit order, and this concerns the batch order of equal elements coming from the same input source. On each cycle FLiMS dequeues consecutive values from banked input queues A and B , with a length of up to w for each input (and totalling w for both A and B through the MAX units). Therefore, the forwarded elements (bitonic sequence) can only be from consecutive batches (i.e. addresses inside the queues) due to the round-robin data

organisation in banks. For this reason, the combinations where two inputs of the *CAS* units have a different 2-bit order field can be pairs of “00” with “01”, “01” with “10”, “10” with “11”, and “11” with “00”. The modification of the *CAS* unit to prioritise between “11” and “00” in the order field correctly identifies the overflow and behaves as these two bits were part of the input order. Finally, if the source and batch order fields are equal, it means that the compared data inside the *CAS* units are from the same batch. In this case, the remaining appended $\log_2(w)$ bits represent the order inside the same batch (i.e. same BRAM address), which also represents the order of the emulated case, as the input data is written in round-robin order inside the input queues, where the w -sized batches are already sorted.

3.4 Comparison of FLiMS with the related work

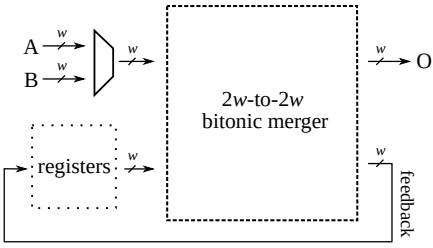
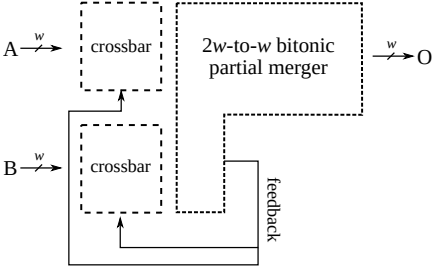
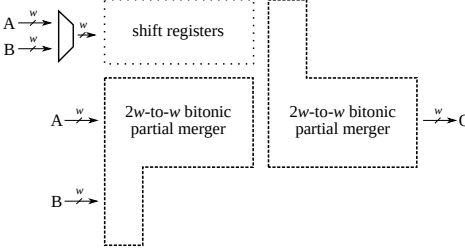
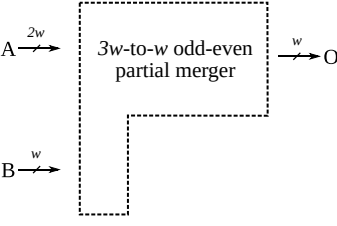
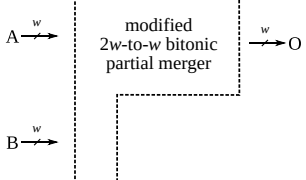
Table 3.2 compares FLiMS to the related work, in terms of feedback length, latency and hardware resources in high-level. FLiMS uses the least amount of resources by only requiring a single $2w$ -to- w bitonic partial merger. It has a single-stage feedback latency, which makes it feedback-less, and has the least amount of latency, which is $\log_2(2w)$.

The first entry is based on a well known algorithm for merging using SIMD instructions on Intel processors [62]. It was adopted for database use [23]. However the long feedback problem was more prominent on FPGAs [35], since it can negatively impact the critical path, and is not scalable for many inputs.

The second entry [69] needs barrel shifters before the input, and this is where the feedback problem is found. Even if the barrel shifters are squeezed in a single pipeline stage in the form of a crossbar, an increased number of inputs has a scalability problem, and was addressed by subsequent works.

MMS [35] and VMS [73] were the first solutions that provided a practical solution for the feedback problem with a relatively low resource utilisation. Their approach was to use either two $2w$ -to- w bitonic partial mergers (MMS) or two $2w$ -to- w odd-even mergers (VMS). Both

Table 3.2: Comparing high-throughput 2-way mergers

2-way merger	Feedback length	Latency	H/W modules
 Casper et al., 2014 [23], [62]	$\log_2(2w) + 1$	$\log_2(2w)$	1 (full) merger ^a
 Song et al., 2016 [69]	$\log_2(w) + 1$	$\log_2(w) + \log_2(2w)$	1 merger & 2 barrel shifters ^a
 MMS [35], VMS [73]^b	1	$2 \times \log_2(2w)$	2 mergers & shift registers
 WMS/EHMS [74]	1	$\log_2(2w) + 1$	1 merger ^c (for more inputs)
 FLiMS [36]	1	$\log_2(2w)$	1 merger

^a The logic inside the feedback can fit in single cycle, but with worse timing and scalability^b VMS uses the same high-level arrangement as MMS, but with the odd-even merge topology^c The extra stage is originally hidden by combining CAS units into more expensive units (CAS3)

of those topologies have $\log_2(2w)$ pipeline stages and are relatively similar. They are from the last $\log_2$ steps from the bitonic sorter and odd-even merge sort respectively [61].

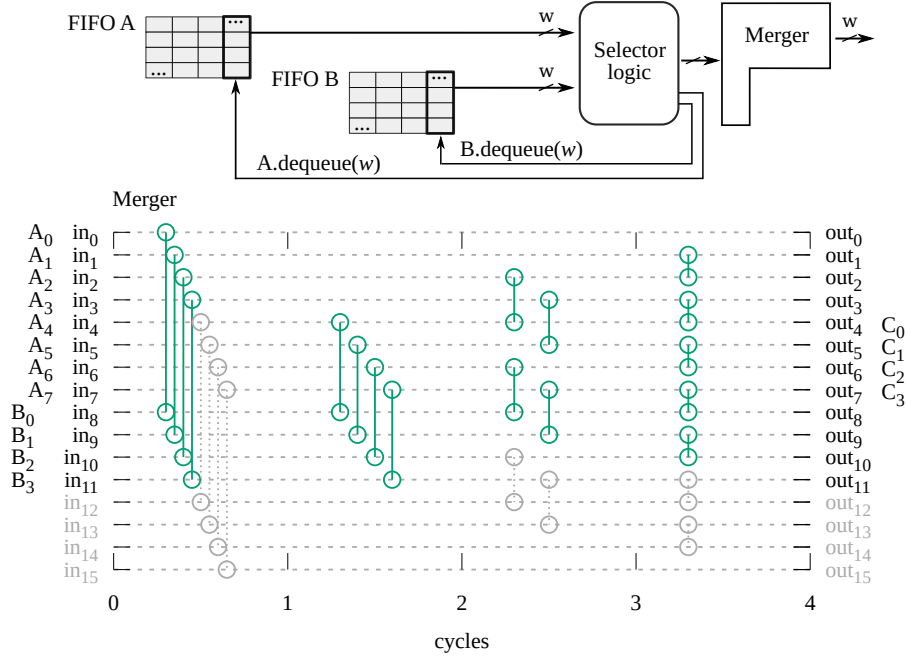


Figure 3.3: WMS [74] architecture (top) and CAS network (bottom)

WMS/EHMS [74] (figure 3.3) on the other hand, achieves to use a single feedback-less merger, as with FLiMS, but this merger is for double the inputs (plus optimisations), totalling to one more pipeline stage. Its compare-and-swap (CAS) network (figure 3.3 bottom) utilises the merger of odd-even mergesort instead of the one in bitonic sort, and for $4w$ inputs (hence the higher latency). There is additional pruning (greyed CAS units in the figure), as the first input list has double the length of the second one. The additional pipeline stage is hidden, as some compare-and-swap (CAS) units are combined into compare-and-swap units for 3 inputs (CAS3). This has a potential impact on the critical path and operating frequency, as more comparisons take place in a single cycle.

Both WMS/EHMS [74] and MMS [35]/VMS [73] have an additional selector stage in the beginning, consisting of multiplexers, and this could be considered an additional pipeline stage. This is not the case with FLiMS, as the selector stage is already implemented in the first pipeline stage of the modified bitonic merger through the *MAX* entities.

Although WMS/EHMS [74] came after FLiMS, they fundamentally require more resources

and pipeline stages, also adding the complexity of a selector logic. One advantage of WMS could be its centralised selector logic, which dequeues whole rows of length w from the input FIFOs (compare figure 3.1 to 3.3 (top)). This could be useful in special occasions, such as when dequeuing very narrow data from block RAM, though an efficient equivalent arrangement could be made for FLiMS as well.

Later, EHMS [75] was introduced as an alternative of EHMS [74], but is not considered here. This is because it is still based on EHMS and its resource utilisation is shown to be very similar to EHMS, but with a further performance overhead due to the increased complexity of its selector stage, worsening its critical path [75]. Nevertheless, FLiMS only considers inputs in powers of 2 and EHMS [75] can be useful in special cases where the inputs is of the form 3×2^i , where i is a power of two. Further modification of FLiMS to support a number of inputs other than in powers of two could be possible, but are outside the scope of this thesis.

Finally, it is important to mention the tie-record challenge, which is present in some parallel mergers including MMS [35], VMS [73], WMS and EHMS [74]. The tie-record challenge is when a merger produces corrupted output when there are duplicate values. Additional expensive workarounds have been introduced in past research to solve this challenge, such as carrying timestamp information alongside the data [35]. The reason that this challenge exists in these mergers is that their merger architecture actually consists of two simpler mergers (or combined into one [74]). As these mergers are based on sorting networks (the bitonic sorter or odd-even merge sort), they are not stable internally, which causes inconsistencies in the dequeuing decisions among duplicate values. This is a problem in key-value pairs, where a key appears multiple times, as the difference in internal orders can result in some key-value pairs appearing twice, in place of other (now vanishing) key-value pairs with the same key.

FLiMS (and the merger of PMT [69]) does not suffer from the tie-record challenge, because there is a single internal order of the data. The dequeue decisions only take place in the MAX stage once. The comparators used to find the top w always give first priority to the elements of list B (corresponding to the condition of line 6 in algorithm 3) and then (between duplicates from the same input list) the priority is based on the internal order of each list following the

natural rotation of the heads. Hence, when considering duplicates, the notion of top w in proof (1) corresponds to the top w when stable-sorting the available w from each input list. This ensures correct operation when there are duplicate values, but this information is then lost (i.e. not performing stable sorting), if the stable merge version (section 3.2.2) is not used.

3.5 FPGA implementation

In order to evaluate FLiMS on FPGAs, its resource utilisation and maximal operating frequency are compared with the latest alternatives WMS and EHMS [74]. Three generator scripts are implemented from scratch that produce Verilog code for a given degree of parallelism w and data width. This experiment uses 64-bit mergers and is evaluated for the Xilinx Alveo U280 board.

The generated designs work as simplistic AXI peripherals, that read already sorted data stored in distributed memory (on-chip) and write back also to distributed memory. The host places the sorted sublists and reads the merged result for validation purposes. For every different value of w , the FIFOs are only 4 elements deep (totalling $4w$ elements for input and output) to eliminate the differences between different merger designs to only their core logic. Though, the bitstreams generated for validation had longer queues. The reported numbers are about the implementations of bitstreams with the AXI peripherals, and not about out-of-context synthesis only.

The designs are not necessarily highly-optimised, and the provided dataset is platform-specific, but they aim for a direct comparison under the same environment. Additionally, in order to simplify the comparison, the fusion of some pairs of compare-and-swap units (CAS) is not followed in this evaluation of WMS and EHMS [74]. This omitted optimisation could be explored separately, as with the register reduction optimisation of section 4.4.4, and does not directly relate to the main structure/ algorithm of the mergers.

Figure 3.4 shows how FLiMS compares with WMS and EHMS with regard to the resource utilisation of look-up-tables (LUTs) and flip-flop registers (FF). As a conclusion, FLiMS uses

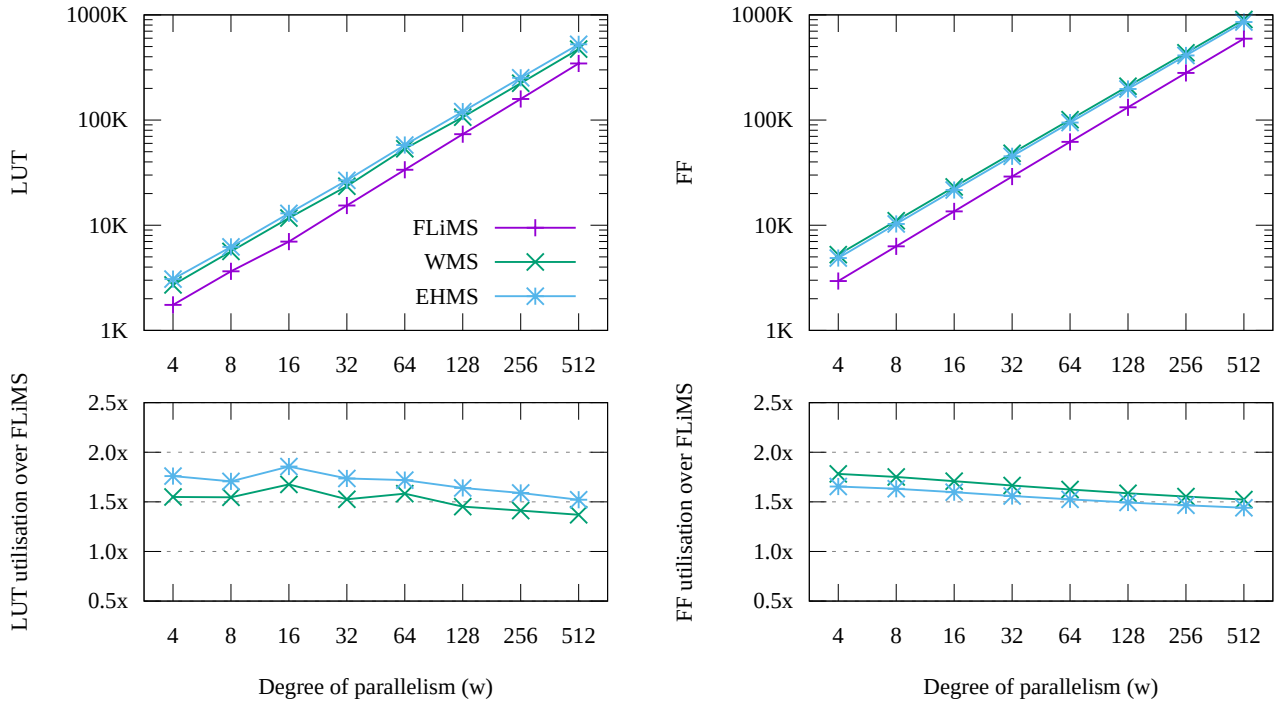


Figure 3.4: FPGA resource utilisation comparison of FLiMS with the latest mergers

the least amount of LUTs and FFs, while WMS and EHMS only differ marginally, as they are based on the same merger, with similar complexity but different optimisations. As expected [75], between WMS and EHMS, WMS wins in LUT utilisation, while EHMS wins in FF utilisation. Using the current implementation as an AXI peripheral, FLiMS is roughly about 1.4 to 1.8 times more hardware resource efficient.

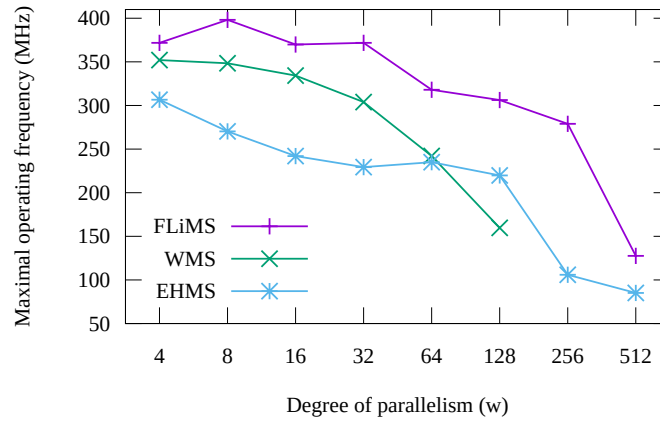


Figure 3.5: Maximal operating frequency for FLiMS, WMS and EHMS

Figure 3.5 presents the comparison of the obtained maximal operating frequencies as reported

by Vivado 2020.1 through the obtained worst negative slacks (WNS). With the default Vivado settings, all designs up to $w = 128$ were feasible. For $w = 256$ only EHMS was routable in the beginning, and for $w = 512$ only FLiMS was routable. After trying different implementation directives $w = 256$ was also possible for FLiMS, as well as the $w = 512$ for EHMS. For $w = 64$ and 256, EHMS originally yielded suboptimal timing results due to congestion, hence some additional directives were applied to avoid outliers. For WMS, the additional tested directives did not help. Having irregularity or small variations in the results are expected, as place-and-route is heuristic-based and becomes more challenging for bigger designs.

As an observation, FLiMS has a considerable advantage over both WMS and EHMS, sometimes yielding more than double the operating frequency. WMS is known to be better performing than EHMS, at the expense of additional hardware resources, but FLiMS still wins in both performance and resource utilisation by a great margin.

Note that the selector stage is the main bottleneck of the alternative approaches WMS and EHMS. Thus, some design choices and parameters such as the comparison width (64-bit here) may exacerbate the frequency difference at different degrees.

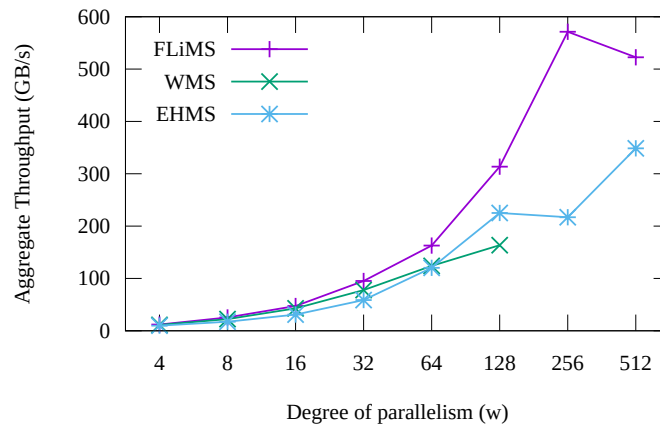


Figure 3.6: Aggregate throughput of the solutions in the design space

Finally, it is important to comment on the practicality of this design space. As can be observed from figure 3.6, the aggregate throughput in the output of each merger can largely exceed the memory bandwidth available to FPGAs today (up to a few tens of GB/s). The equivalent argument can be said for the input's perspective, where multiple (w) memory channels would need to be available for fully utilising these mergers, if the goal is to build a parallel merge tree.

Another observation is that for $w = 512$ for example, the impact of the increased hardware utilisation on the operating frequency results in a slower FLiMS merger overall. For low values of w , as with the sorter of section 5.1, using such a merger is perfectly sensible for saturating the bandwidth. This is also true when using BRAM instead of LUTRAM, whose organisation is more restrictive in this aspect. Hence, LUTRAM was used here to study the scalability of the logic in a physical implementation, though according to the application BRAM can be more suitable, as in the sorter design of section 5.1. Efficient mergers are important in other applications as well, including for building sufficiently large merge trees [17], which consist of such parallel mergers. Nevertheless, the many-input mergers in this design space, such as for $w > 32$, are provided for the sake of completeness rather than practicality. This is to show that FLiMS consistently outperforms the alternative state-of-the-art designs on a current FPGA micro-architecture, and without excluding potential future applications, such as with future technologies and novel applications including on narrower and on-chip data.

On a similar note, it might be desirable to sort data with a similar or greater width to the datapath. An example is with the Graysort benchmark [42], [135], which has 100-byte records (800-bit), of which the 10 are the key (80-bit). This can be representative of more traditional database systems, where the entire row needs to travel along the key (row-store). In these cases, many-leaf mergers are more suitable for achieving less data movement and faster sorting, as with the mergers of sections 2.3.4 and 5.1.1. On the other hand, column-store databases and plain integer sorting [136], high-throughput merging is still a rather active research topic, as with the latest alternative 2-way mergers [35], [73]–[75]. Section 2.2 provides background information about the database storage model in the context of FPGA-based research. Section 5.4 presents a practical use case in an industrial environment (introduced in section 1.2) that uses a stream of sorted 64-bit key-value pairs.

This evaluation, focusing more on the merging techniques, ignored the building of merge trees or the handling of the list endings. Chapter 5 presents full system solutions based on FLiMS for data in DRAM and includes the skewness optimisation. Section 5.1 introduces a hybrid parallel merge tree approach that uses FLiMS as a building block for achieving many-leaf merge with high-throughput, without the need to use impractically wide FLiMS mergers.

3.6 Software implementation using SIMD intrinsics

Today's general purpose processors (CPUs) feature vector or single-instruction multiple data (SIMD) instructions as a way to increase performance in numerous compute and memory-intensive applications. Parallel merging algorithms implemented using SIMD-intrinsics, have already been shown to improve the sorting performance on CPUs.

Chhugani et al. [62] used a rather simple merge algorithm, also based on the bitonic merger, to enable high-throughput merging on an older Intel processor. As with some of the presented FPGA-based related work, a single comparison of the first elements per batch in the input queues is used to distinguish the next batch to dequeue. However, it requires a full bitonic merger (non-partial), as both the lower w and upper w are used, as in [23]. The lower w is written to the output destination, to progressively merge the entire input. The upper w elements are transferred back to the input. This would have the feedback problem that the latest research on FPGA merging was addressing, as the upper w is calculated after $\log_2(2w)$ stages. However, on CPUs this is not much of a concern, because the pipelining functionality is not that advanced to achieve efficient task-pipelining. In other words, the pipelining in the CPU is done mostly between instructions implementing a single pass of data in the pipeline of the merger, rather than between different stages of merger, where a lengthy feedback would cause stalls. For example, for a single layer of compare-and-swap (CAS) units, there need to be at least three SIMD intrinsics, one for *min*, one for *max* and at least one *shuffle*. The latter is to appropriately permute the inputs to emulate the CAS network topology. This would work very similarly with the latest AVX-512 SIMD extension [109], and was evaluated at least once, but for an application with all layers in the sorting network [5].

In practice, how FLiMS could help is with a reduction in instruction count, as the lower w of the result is not needed. In other words, with roughly a similar number of instructions, FLiMS can merge with double the amount of parallelism w . Other desirable characteristics on FLiMS for SIMD are the bitonic merge topology, which is more regular/ symmetric when compared with the odd-even merge sort topology, and the elimination of the rotation of the inputs. More generally, a similar discussion supporting FLiMS can be made for all other hardware mergers

in the comparison in section 3.4, though in some cases [69] the performance may relate more easily to the micro-architectural implementation of the used intrinsics.

This section presents the evaluation of this approach. In order to assess the efficiency of FLiMS as an SIMD algorithm, a manually vectorised code in C++ is developed using Advanced Vector Extensions for 256-bit integer registers (AVX2). For more information on the implementation using AVX2 and AVX-512 intrinsics, as well as discussion on a design space exploration for fine-tuning, see Appendix A.

3.6.1 Merge function implementation

FLiMS consists of a MAX stage, plus a butterfly network. The MAX stage can be implemented by keeping the cA and cB in vector registers (in code) and generating the top w . The cB vector is represented in reverse order by using the AVX2 intrinsic `_mm256_permutevar8x32_epi32()` to reverse the contents of each vector register it holds after fetching from list B. The comparison of the MAX units is done by the intrinsic `_mm256_cmpgt_epi32()` and the result vector is also used to fetch the next heads selectively after blending `_mm256_blendv_epi8()` with the current set of heads. The result of the compare instruction `_mm256_cmpgt_epi32()` has the form of zeros and ones, each now denoting the source list of the next heads. The negation `_mm256_andnot_si256()` of this vector is also used to get the equivalent vector to be used when blending the next vector for the second input list.

With respect to the butterfly network part, it consists of compare-and-swap (CAS) units. Each layer of swap units can be calculated by calling both a min (`_mm256_min_epi32()`) and a max (`_mm256_max_epi32()`) instruction, since each CAS has two outputs. Alternatively, a compare instruction can be used once `_mm256_cmpgt_epi32()`, but there is a performance overhead from the additional calls of the `_mm256_blendv_epi8()` instruction, required to translate the result vector of `_mm256_cmpgt_epi32()` into min and max. Before each CAS layer, the inputs must be properly aligned, and this is done through different permute intrinsics such as `_mm256_shuffle_epi32()` and `_mm256_permute2x128_si256()`.

The SIMD-based merge function is evaluated stand-alone on an Intel i7-8809G as a target processor, with a single-thread running at a steady 4.2 GHz. Two sorted random inputs of 2^{24} are prepared to be merged by the FLiMS-based merge, as well as for a serial merge approach. The baseline is a simple serial merge function in C++, where two pointers are accumulated according to what is written to the output after each comparison. Then the results of both the manually vectorised code (the FLiMS-based) and the serial version are compared. The compiler flags for each case are $-O3$ and $-march = native$ to attempt automatic vectorisation of the serial function and support the respective Intel intrinsics subset for the current architecture. The lists reside in main memory without any explicit banking scheme, but they are aligned, for ensuring SIMD performance.

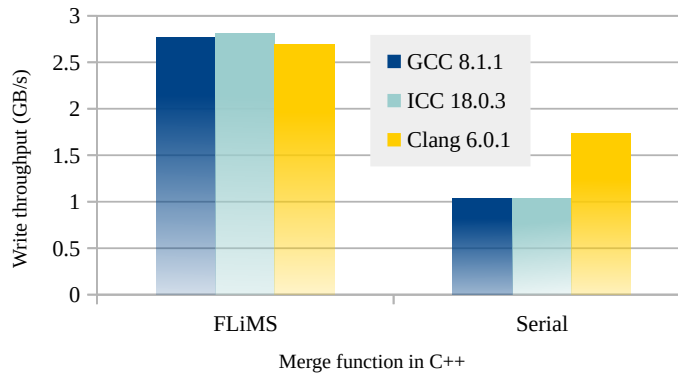


Figure 3.7: Results for AVX2-based FLiMS for merging 2 lists of 2^{24} 32-bit integers

In figure 3.7, we can see the results for 2-way merging using 3 different compilers. The data points are averages of 25 measurements. This implementation does not feature the skewness optimisation, and uses a FLiMS $w = 16$, with two AVX2 vectors per input (2×256 bits) holding 16 integers instead of one for 8 integers, as an optimisation. See appendix A for more details on implementation and optimisation.

An observation is that the FLiMS-based implementation has less variation between different compilers, as the SIMD intrinsics map to the respective instructions with less room for compiler optimisation (at least with these compiler versions). As a conclusion it seems that the manual vectorisation of the merge function was successful and achieved a speedup of at least 1.5x over the automatically-vectorised (where applicable) code.

3.6.2 Sort-in-chunks function implementation

The FLiMS-based CPU merge function can be used hierarchically to merge input of arbitrary length. However, FLiMS requires at least w -sized input lists, and any lower levels during merge-sort would require an alternative sorting approach. Therefore, the data need to be first sorted in small chunks.

As a complement to the FLiMS-based merge function, a sort-in-chunks function is developed to facilitate the need of sorted chunks, as well as to provide long-enough chunks for FLiMS to benefit from the streaming access pattern. This function is based on the bitonic sorter, with a similar technique for building a CAS network from section 3.6.1. An equivalent approach was followed in an SIMD-based quicksort implementation [5], although here it is implemented from scratch and optimised for the test platform, such as by using AVX2 intrinsics. The selected optimal chunk size was 512 integers, through the design space exploration of section A.3.

3.6.3 Complete sorting function evaluation

The performance of a FLiMS-based sort function is compared against the C++ Standard Library implementation of sorting `std::sort()`, as well as a highly-optimised [5] SIMD-based radix sort implementation from Intel's integrated performance primitives (IPP).

Additionally, a multi-threaded version of the FLiMS-based SIMD sort function is implemented, with the help of OpenMP pragmas. Both the merging function and the sort-in-chunks function are unaltered. The parallelisation was done on the calls of each. The sorting-in-chunks now happens on all cores, operating on equally-sized consecutive portions of the entire input. An equivalent loop initiates multiple instances of the FLiMS-based merge, as long as there are enough sublists in the current merge iteration. There is currently a bottleneck in the mergings of the latest iterations, when the cores can be underutilised. For instance, with 16 threads, the last 3 iterations of mergesort are merging 8 sorted sublists, then 4 sublists and finally 2 sublists. This implementation can be further optimised.

The performance of the multi-threaded FLiMS-based sort is compared against the single-threaded baselines, as well as a parallel sort implementation in the Boost C++ libraries. The *block_indirect_sort()* function implements the samplesort sorting algorithm, and is regarded as one of the best performing C++ sort implementations [137].

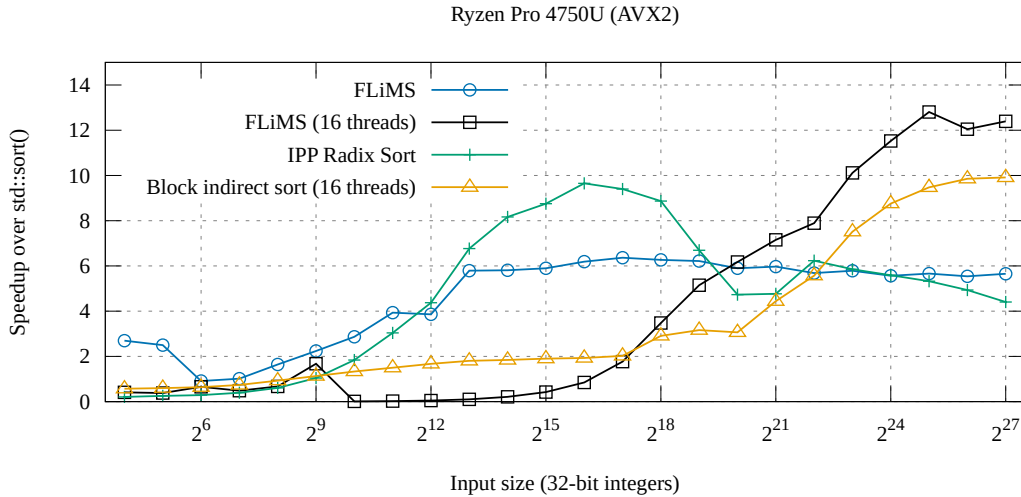


Figure 3.8: Evaluating the FLiMS-based SIMD complete sort C++ functions

Figure 3.8 presents the results of both the single-thread multi-thread experiments. The target processor is Ryzen Pro 4750U, a relatively new AMD processor able to run 16 threads with simultaneous multi-processing (SMP). This system has dual channel memory and uses 2×16 GB SODIMM modules of RAM running at 2400 MHz. Note that there is a higher fluctuation for shorter inputs, as each data point is an average of multiple runs of a binary, rather than averaging multiple iterations inside the program. This is used to try avoiding any bias from CPU’s adaptive mechanisms training on past iterations (though this thesis focuses on problem sizes past the cache size). The main observation is that the 16-thread FLiMS sort function surpasses the performance of the 16-thread *block_indirect_sort()* for the input range from 2^{17} to 2^{27} . A hybrid approach can be used to enable the single-threaded version of FLiMS for below 2^{20} to achieve the highest-performance overall, except radix sort.

Intel’s radix sort takes the lead between 2^{12} and 2^{19} on this AMD processor, but IPP radix sort has some notable limitations. These include the less predictable performance across different systems (see section A.4), license restrictions and closed source, and radix-sort related impli-

cations, such as a limit of the input length to about 2^{28} for the tested implementation, making it more challenging to adopt for big data.

With respect to the number of threads, the parallel version of the FLiMS-based sorting seemed to scale well in this platform. In other words, with increasing the number of threads the performance improves. This holds for up to 16 threads, which is the maximum threads the AMD 4750U CPU supports through simultaneous multi-threading (SMT). One of the possible reasons the increased thread count does not become a bottleneck is that this processor has a 16-way last-level cache (LLC), which prevents cache thrashing to a degree. For instance, ideally, the sort-in-chunks function could cause near 0 conflict misses, if the 16 threads progress at the same pace, since each thread only updates consecutive chunks in memory. In contrast, this was not the case with memory copying that is needed in the last iteration of merge sort to copy the result back to the output buffer. The aggregate throughput of memory copying only peaked at 4 threads only. This can be explained by the fact that *memcpy()* has a much higher memory to compute ratio, and can stress the interaction with the dual channel memory that is present to the system more easily. Future work includes exploring the use of prefetching instructions, and refining the access pattern for better utilisation of the cache [138].

As a conclusion, FLiMS-based sorting is an attractive approach to accelerate merge sort using SIMD intrinsics. Though, according to the input, alternatives to merging could still be better performing, such as with radix sort, which would perform fewer data passes on data of restricted range (e.g. 10-bit integers). This evaluation used a uniform distribution for the random inputs, and alternative sorting algorithms can be optimised towards certain input features [131]. An important note is that although sorting 32-bit (or wider) integers with CPUs can be useful for databases, FLiMS on FPGAs provides more flexibility for databases, including the custom data format, the skewness optimisation, and the ability to seamlessly extend the pipeline with specialised logic.

3.7 Summary

FLiMS is currently the most hardware-efficient 2-way merge block on FPGAs. It can be used to build efficient high-throughput merge trees for sorting completely unsorted input on hardware. It features fewer and/or simpler pipeline stages than the alternatives, while achieving a higher amount of parallelism with less hardware resource utilisation. The skewness optimisation is the equivalent workaround found in an older merger, while maintaining the decentralised nature of the *MAX* entities logic, which is collectively used as a scalable selector logic. A SIMD implementation of FLiMS on a modern processor is also found to outperform popular highly-optimised C++ sort libraries in a relatively thorough evaluation, including a multi-threaded variant. As FLiMS is based on merge sort, it can be more appropriate for big data analytics acceleration, since the same logic can be applied hierarchically to sort arbitrarily long data without keeping growing states. The reconfigurability of FPGAs is still preferred for accelerating big data using FLiMS, due to the flexibility for custom data format, including carrying payloads for column-store databases, and for applying the skewness optimisation or the stable version.

Chapter 4

Parallel round-robin arbiter

In order to achieve faster computation, the need for a higher-bandwidth memory has led to increased data width and wider data units. Processing systems including CPUs, GPUs and FPGAs have adapted to this change by increasing the width of the datapath, rather than the operating frequency. This also translated to wider interconnects, ports for inter-chip communication etc.

There are plenty of examples of this direction in many aspects of computing, from the microarchitecture to different programming models. In the world of FPGAs, the Zynq FPGA architecture which combines an FPGA with some ARM cores, originally offered 64-bit-wide ports between the programmable logic (PS) and the programmable system (PS), but now the Zynq UltraScale+ offers 128-bit-wide ports. Correspondingly but for higher-level programming, in Intel CPUs, the single instruction, multiple data (SIMD) instructions have eventually evolved to include from 64-bit to 512-bit vector registers [109]. In order to benefit from this increased memory bandwidth, clever techniques are responsible to achieve high performance with “slow” but more parallel hardware.

In FPGAs, there are some workarounds to benefit from the increased datapath width or bandwidth and they mainly revolve around multi-tasking [27] and crossing clock domains (or more generally doing more in a single FPGA cycle). For multi-tasking, the hardware designer must think of creative ways to split the data for processing with multiple workers, and aggregate

their output, if required. These workarounds are sometimes unnecessarily complex to implement, and consume a considerable amount of resources. Other shortcomings include logic or data redundancy and also scalability issues, unless the nature of the task is truly parallel and is very suitable for a distributed approach. When crossing clock domains, a simple worker could, for example, run with multiples of the frequency of the datapath. An ASIC example applicable to FPGAs is the priority encoder in network switches, which does not scale well for increasing the number of inputs [139]. Unfortunately, there is relatively little work on clever parallel algorithms that focus on a single general task on FPGAs.

Considering that independent tasks on FPGAs (equivalent to threads for CPUs) could increase the “randomness” of the memory access pattern, they can be challenging to optimise for performance and utilise 100% of the memory bandwidth. Part of the induced hardware complexity of multi-tasking solutions are additional workarounds for the increased memory latency, such as incorporating small caches per task [42]. Therefore, one of the goals of this thesis is to parallelise computation of a single task or thread as much as possible, to yield a linear access pattern (when applicable). Such high-throughput designs are able to achieve high performance with low resource utilisation and power overhead.

This chapter introduces the parallel round-robin arbiter¹, also known as round-robin module [43]. It is a novel non-blocking high-throughput stream processor, with a functionality that has proven useful for accelerating specialised database operators. For direct applications to accelerating different database operators, the chapter 5 introduces accelerators that make use of the parallel round-robin arbiter as a building block.

This chapter also focuses on a high-performance network switch architecture [39], which is based on the concepts of the parallel round-robin arbiter, but is indirectly related to datacenters. A novel FPGA-based design is presented for performing high-bandwidth network switching. The proposed solution replaces the crossbar with the combined parallel round-robin arbiter. Unlike a crossbar, the combined parallel round-robin arbiter is easy to pipeline, and does not require centralised iterative scheduling algorithms that try to fit too many steps in a single or

¹(not to be confused with [140])

a few FPGA cycles. The result is a network switch implementation on FPGAs operating at a high frequency and with a low port-to-port latency. The proposed architecture additionally provides a competitive switching performance approaching output-queued crossbar switches. The implemented designs exhibit a throughput that exceeds 100 Gbps per port for switches of up to 16 ports, reaching an aggregate throughput of around 1.7 Tbps.

While FPGAs have already been adopted by cloud providers as a hardware acceleration platform, FPGA vendors have been increasing the transceiver count on mid to higher-end FPGAs, making them attractive also for single-chip network switch implementations. Xilinx Virtex UltraScale+ FPGAs support up to 48 GTM transceivers (up to 58 Gbps each), and up to 128 GTY transceivers (up to 32.75 Gbps each), depending on the speed grade and device/package combination [141], [142]. In latest devices, the available logic resources have also increased substantially. For example, the recently announced VU19P device features around 9 million logic cells. This leaves a gap in research on FPGA-based switches, which are usually unable to saturate the bandwidth of the transceivers while retaining competitive switching performance.

The presence of FPGAs in network switches can also be beneficial to distributed computing and the datacenter. Support for offloading computation to network switches could bring the computation closer to the data and improve the data movement efficiency in multi-node and big data applications. Section 6.2.7 discusses this idea in high-level as a potential future work for such a high-performance switching approach.

This chapter is partly based on the paper “Accelerating the merge phase of sort-merge join” [43], which first introduces the concept of the parallel round-robin arbiter in the context of databases (see chapter 5). It is also based on the paper “High-Performance FPGA Network Switch Architecture” [39] that elaborates on its application in network switches, as well as on some of its lower-level properties. The co-authors of these papers mostly helped with proofreading and providing general knowledge or supervisorship in the papers. The material presented here is the author’s, with the exception of Jiuxi Meng providing the baseline dataset for comparison in 4.5.2. The chapter extends on the concepts through additional functionality, properties and optimisations.

4.1 Pipelined building blocks

This section provides more information about the smaller building blocks of the proposed solutions, including the parallel round-robin arbiter's extension to a network switch.

Since the target platform for the designs is FPGAs, highly pipelinable circuits are selected as building blocks for the parallel round-robin arbiter and its applications. Such structures are also known to scale well with increasing number of inputs, which ensures high operating frequencies for large designs and future highly-parallel architectures.

The idea is that the logic is divided in parallel steps of no nested computation between the lower-level blocks, which are either compare-and-swap (CAS) units, adders or multiplexers. There is a set of registers between each parallel step that collectively increase the latency of the circuit as a whole, but keep the critical section narrow, so that a high operating frequency can be achieved.

- a) Sorting network: Such parallel algorithms consist of compare-and-swap (CAS) units and are able to sort a list of P elements in $O(\log_2^2(P))$ parallel steps (see section 2.3.1).
- b) Adder tree: It calculates the sum of P inputs (i.e. $out = \sum_{i=0}^{P-1} in_i$) in $\log_2(P)$ parallel steps.
- c) Prefix sum: The parallel version of prefix sum [143] can be seen as a superset of the adder tree. It provides the cumulative sum (i.e. $out_k = \sum_{i=0}^k in_i$ for $k \in \{0, 1, \dots, P-1\}$) in $\log_2(P)$ parallel steps. (It is only used in the *combined* parallel-round robin arbiter.)
- d) Barrel shifter: Here implemented as rotators, they rotate a number of elements by a specified offset. The wiring in the presented pipelined implementation, as shown in figure 4.1d, has a small resemblance to the wiring of prefix sum and consumes $\log_2(P)$ pipeline stages. The select signal gets propagated alongside the data (packets) and the respective bit is fed across multiple 2-to-1 multiplexers in each stage.

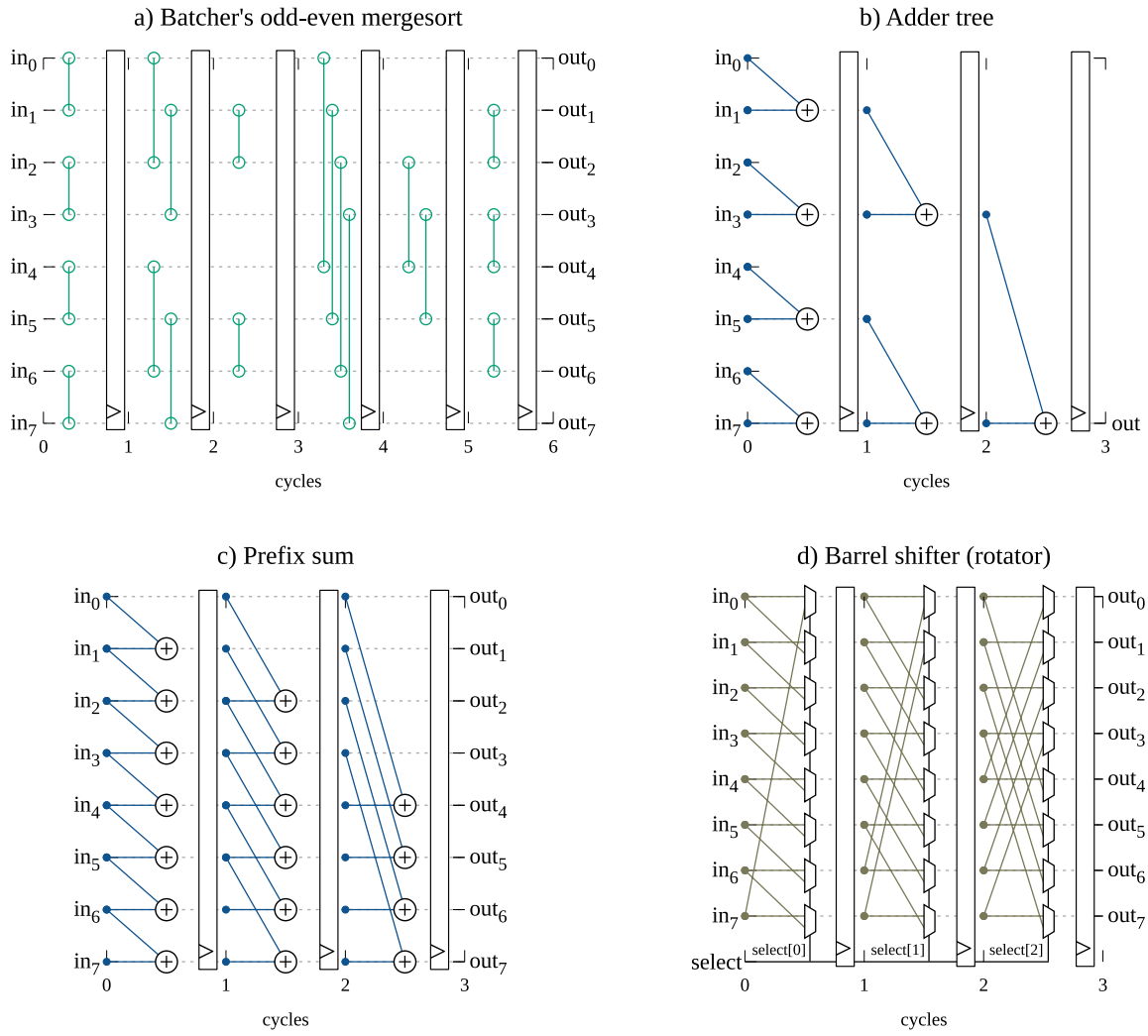


Figure 4.1: Pipelined versions of the building blocks for 8 inputs

4.2 The parallel round-robin arbiter

The parallel round-robin arbiter (PRRA) rearranges incoming packets with a fixed latency, so that the output to a number of banked memory or another processor appears to be written in round-robin fashion, but this is done in parallel. In other words, it removes sparsity (with *Nulls*) from sparse data, and can also be considered a load-balancer for banked memory, as the banks are seemingly written in round-robin fashion.

4.2.1 High-level functionality

The parallel round-robin arbiter receives 0 to P packets or data units per cycle, where P is the arbiter's number of inputs. For each batch of packets it receives, it then permutes the packets and places them at the output after a fixed latency. The performed permutations ensure that the packets arriving at the same cycle will appear consecutively at the output and with the correct rotation offset, such that if it is connected to P banks, the data has no sparsity and is also evenly distributed with a round-robin priority.

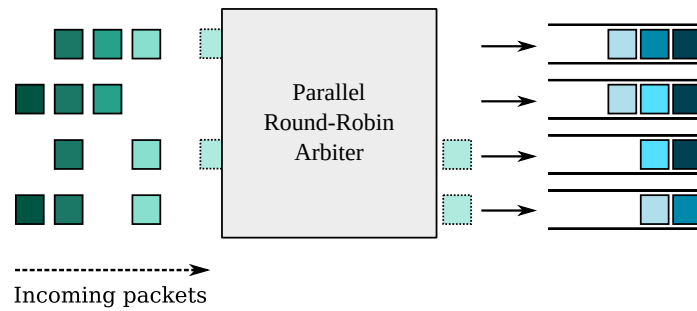


Figure 4.2: Parallel round robin arbiter visualisation

Figure 4.2 visualises an example instance of a parallel round-robin arbiter with four inputs and four outputs. Figure 4.3 illustrates a similar execution but with a waveform, a more technical diagram that shows how signals vary with the passing of the time. In the latter figure, only the valid bits are shown, to represent the existence of a packet in the input or output. Note that the output signals are shifted to the left (equivalent to a processing latency of 0 cycles) to better show how they correspond to the input signals.

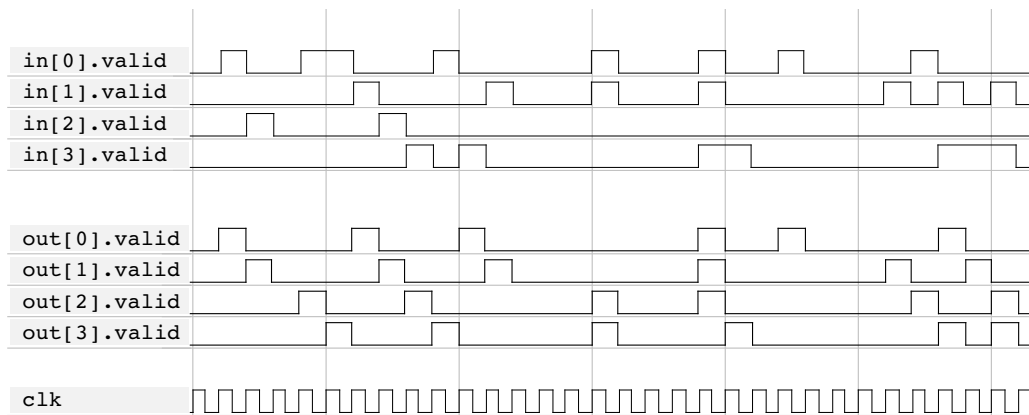


Figure 4.3: Parallel round robin arbiter waveform

As an observation from both example visualisations, the input involves packets arriving in arbitrary positions. The parallel round arbiter processes everything on-the-fly, and no packet is delayed for a different cycle other than the fixed processing latency, which is the same for all inputs.

4.2.2 Proposed implementation

As a baseline implementation, a sorting network such as a bitonic sorter sorts the input packets based mainly on the valid bit, to bring the valid packets on top, in $\Theta(\log_2^2(P))$ parallel steps. During the last $\log_2(P) + 1$ pipeline stages of the sorting network, an adder tree counts the number of incoming packets by performing an addition of the valid bits. After the adder tree and the sorting network, a final adder adds the number of valid packets in the “current” batch to the value of a $\log_2(P)$ -bit register. The value of the register represents the rotation that needs to be performed on the “succeeding” batch of up to P packets. Finally, $\log_2(P)$ additional pipeline stages implement pipelined barrel shifters working as rotators that perform the final rotation. Figure 4.4 shows a high-level view of the parallel round-robin arbiter.

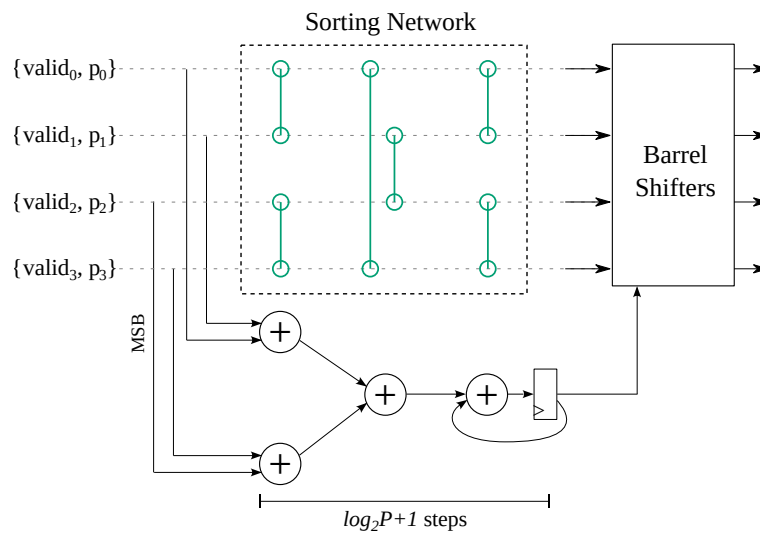


Figure 4.4: A pipelined implementation for the parallel round-robin arbiter, for $P=4$

4.2.3 Adopting for data analytics

With the use of a sorting network, the proposed implementation can be unnecessarily complex for its purpose. For instance, in place of the sorting network, there are “concentrator” circuits or “binary sorters” that can be used for bringing the active input towards the top instead (also see 6.2.2).

The sorting network allows the design choice of the concentrating action to be *stable*, as in stable sort, where equal values must appear in the same order in the output as in the input (but including the required rotation). Here, the term *stable* is adopted for referring to the incoming (valid) packets as opposed to *Nulls*. The output sequence is sorted so that it contains consecutive packets in the same order as in the input, but also rotated by the current offset, so that any sparsity will not affect the integrity of the data, if the order matters.

In order to achieve a *stable* parallel round-robin arbiter, the order of the data must be considered by the compare-and-swap (CAS) units. This is achieved by concatenating the valid bit and the appropriately ordered data field for comparisons, as shown for the fields p_0 to p_3 in the example in figure 4.4, where p_i can contain order information.

An alternative implementation of a *stable* parallel round-robin arbiter is shown in figure 4.5. Instead of comparing the payload (p_i) for extracting the order information, the line number is compared instead. The concatenation of the valid bit and the line number will end up leaving the *Nulls* at the bottom and the valid packets ordered as in input. This does not necessarily inherit the order of the overall order of the input, because if the sparse data come rotated, the line information is not sufficient to distinguish the rotation of the input. However, in the studied applications the batches of sparse data in the input are not rotated. As the whole input batch is released per cycle, of P elements (including *Nulls*), and any rotation will be naturally split in two cycles, resulting in the correct priority given by the line numbers. This solution only requires a comparison of $\log_2(P) + 1$ bits by each compare-and-swap (CAS) unit in total.

Sections 5.2 and 5.4 present accelerators for sort-merge join and distinct count respectively. These solutions are applications of the parallel round-robin arbiter in databases that exploit

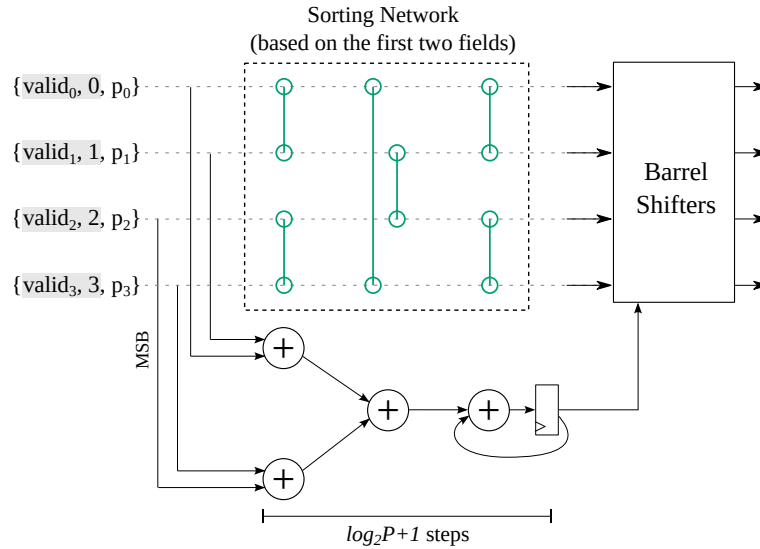


Figure 4.5: A *stable* parallel round-robin arbiter, for $P=4$

the use of the sorting network. The technique to use a sorting network also enables other interesting uses than just concentrating, including the network switch presented in section 4.3.

4.3 The combined parallel round-robin arbiter

The combined parallel round-robin arbiter (CPRRA) is used to build a novel network switch designed, that perform well as an FPGA-based implementation, by not requiring iterative algorithms [144], logic speedup [13] or memory speedup. Independently of the implementation, it is shown that it is also competitive as a switching approach under different traffic patterns. This is something that previous research on FPGA switches gave almost no emphasis, despite its impact on memory utilisation, dropped packet rate and average latency.

FPGA-based network switch implementations are inspired by well-known approaches used for ASICs/ASSPs, usually classified by the position of their buffer memory. Output-queued (OQ) network switches have been shown to perform better algorithmically (lower packet latency) than input-queued (IQ) switches [145], as IQ reduces routability between the queues and the output ports in certain traffic patterns (see figure 4.6). However, the adoption of OQ switches has generally been limited, due to the requirement for output queues to operate at a much higher frequency. To solve this problem, various hybrid approaches emerged, making use of both input

queues and other memories [13]. These present different shortcomings, such as the need to move the speedup to the internal logic. Moreover, for FPGAs such workarounds have a prominent performance overhead, being unable to saturate the available link bandwidth efficiently. Thus, the latest FPGA solution has fallen back to input-queued switches [144], and with bigger flits, in order to achieve a moderately high throughput, inheriting the related limitations.

As a proof of concept, the provided open-source implementation of a 16×16 switch [146] provides an aggregate throughput of 1.7 Tbps on a Xilinx Alveo U280 FPGA, surpassing the last reported, theoretically achievable 819 Gbps [147], which requires an embedded network-on-chip (NoC) [148].

4.3.1 Background

Head-of-line (HOL) blocking is the performance bottleneck in crossbars when multiple inputs request to send to the same output port. It is known to limit the throughput to just 58.6% [149]. The available workarounds try to rearrange the transmissions of the most recently arrived packages using temporary buffer memories.

Virtual Output Queues (VOQs) are widely accepted as a solution to head-of-line blocking. They are used in input-queued (IQ) switches as a buffer space between the inputs and the crossbar. There is one queue for each source/destination combination, totalling $P \times P$ queues, where P is the number of ports. The idea is that a packet will always have a place to go, until its transmission is determined by a *scheduling algorithm*, given that the respective queue is non-empty. As demonstrated in figure 4.6 (and in section 4.5), VOQs have performance limitations. These include buffer fragmentation, causing dropped packets, and poor performance for bursty and nonuniform traffic.

They are still found today in the highest-end switches, such as Arista’s 7800R3, which provides an aggregate throughput of up to 460 Tbps [150]. VOQs also exist in some form in hybrid memory model switches [13], such as in combined input and output queued (CIOQ [145]), combined

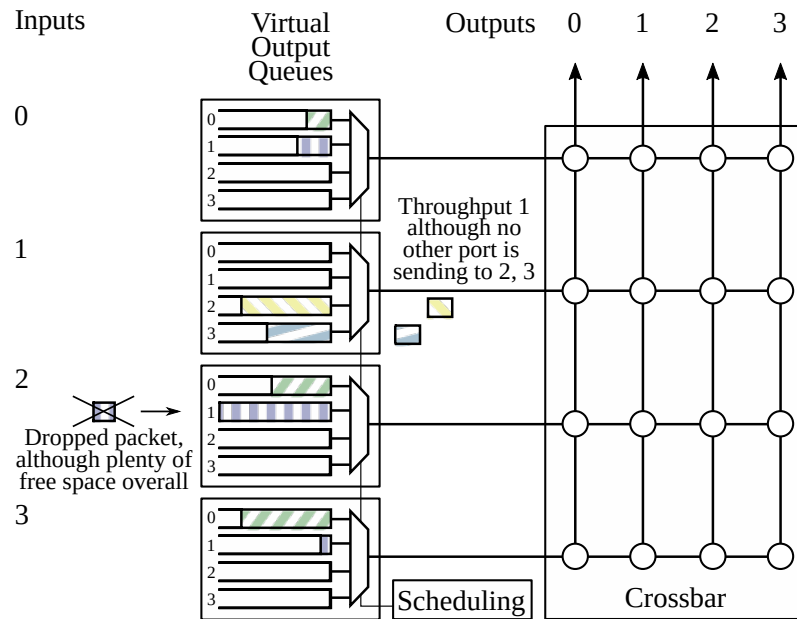


Figure 4.6: An input-queued switch showing two limitations

input and crosspoint queued (CICQ [151]) and hierarchical crossbar (HC [152]) switches.

Scheduling algorithms or matching algorithms are used to make those ‘dequeue’ decisions, by finding a nearly-maximum matching between the available input and output ports [149]. The goal is to maximise throughput, while providing other desirable features, such as preventing starvation, i.e. ensuring no input is getting blocked indefinitely. They require multiple iterations to perform well for various traffic patterns.

Round-robin arbiters are used for input arbitration to achieve fairness to incoming requests [149]. They are commonly applied to dequeue packets from 1 out of P queues, with a round-robin priority. The incoming packets can arrive at any of the P inputs. When there is an input available, a grant signal is enabled, and the arbiter selects the next available queue, based on a priority offset.

In hardware, a round-robin arbiter is usually implemented using a programmable priority encoder (PPE). The PPE can be expensive to implement efficiently and can contribute to the critical path for an increasing number of inputs [139].

In practice, the round-robin arbiter can have small modifications to accommodate the requirements of a scheduling algorithm. One way is to postpone the rotation until a queue is emptied entirely, in order to improve scheduling performance for bursty traffic [153].

4.3.2 High-level functionality

The idea is to use fully-pipelined logic to process the incoming packets at line-rate and insert them in output queues in a balanced way. There are P groups of P output queues, where P is the number of ports. Similar to Virtual Output Queues (VOQs), this requires $P \times P$ queues. However, by moving those queues towards the output, the traffic has the opportunity to get reorganised in such a way that each of the P FIFO groups is filled with packets only destined for that port. In each cycle, up to P packets arrive, which are then grouped according to their destination. These packets are permuted such that the P queues in each of the P groups appear to be filled serially in round-robin fashion, but this is done in parallel, as with a parallel round-robin arbiter. Figure 4.7 outlines this architecture.

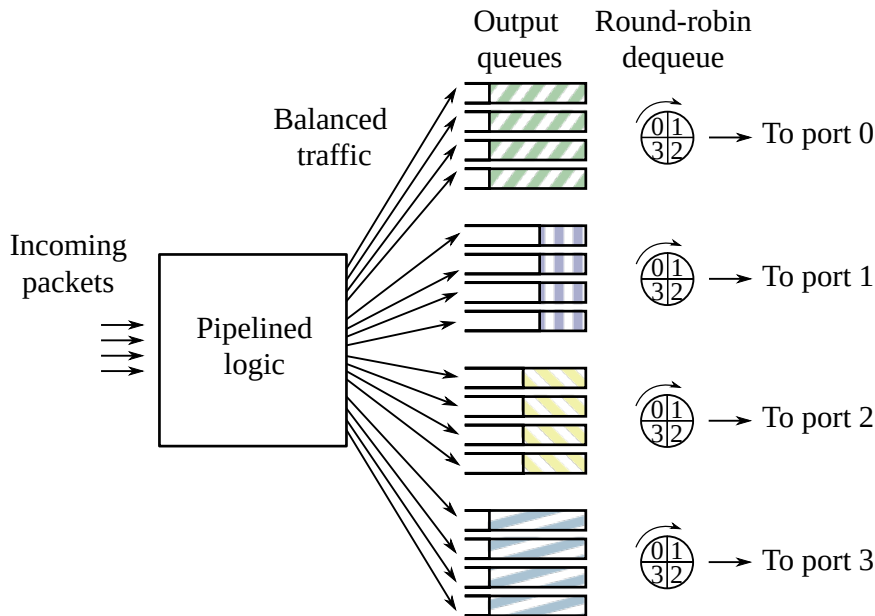


Figure 4.7: Abstracted view of the proposed network switch, $P=4$

The balanced traffic approach achieves better utilisation of the available memory resources. This is because the P queues of the same group are filled at the same rate, leaving no gaps of

unused resources. In contrast, the $P \times P$ queues in traditional VOQs assign only one queue for each $(port_{source}, port_{dest.})$ combination. A packet arriving in the proposed switch has the possibility to land in P queues for $port_{dest.}$, which are virtually unified.

Near the ports, there are P *round-robin arbiters*, which rotate to provide the content of each FIFO group in a serialised manner, with a throughput of 1 per port.

4.3.3 Proposed implementation

The proposed implementation focuses on high performance for FPGAs. As a result, the commonly-used crossbar is replaced in favour of pipeline-friendly building blocks (as described in 4.1). This removes the need to solve bipartite graph problems (or approximations) on-the-fly, which is what scheduling algorithms are for.

It can be seen as a hardware optimisation of the following switching scheme. The switch functionality can be accomplished by P parallel round-robin arbiters, one for each output port, for balancing the occupancy of each FIFO group. Initially, up to P packets received on that cycle are being broadcasted across P independent and identical pipelines, one for each port. For each pipeline, a filter then invalidates the packets going to other ports. Then the parallel round-robin arbiters are responsible for storing the packets in round-robin fashion in each of the FIFO groups. Finally, ordinary round-robin arbiters dequeue the packets from the FIFO groups in a serialised manner. Figure 4.8 depicts this unoptimised network switch design.

The *Combined parallel round-robin arbiter*, as shown in figure 4.9, only uses one sorting network, instead of P sorting networks for P separate *parallel round-robin arbiters*, hence its name.

It consists of two pipelines being merged together, with additional pipeline stages at the end for the final rotation of the packets, before appending them to the FIFO group of their destination port. The first pipeline is for the ‘sorting network’ and the second one, ‘offsets’, is for calculating the amount of rotation each barrel shifter (rotator) needs to perform. The ‘offsets’ pipeline includes everything below the sorting network in figure 4.9. The two pipelines have different lengths. Thus, there are some additional measures for synchronising their output. When the

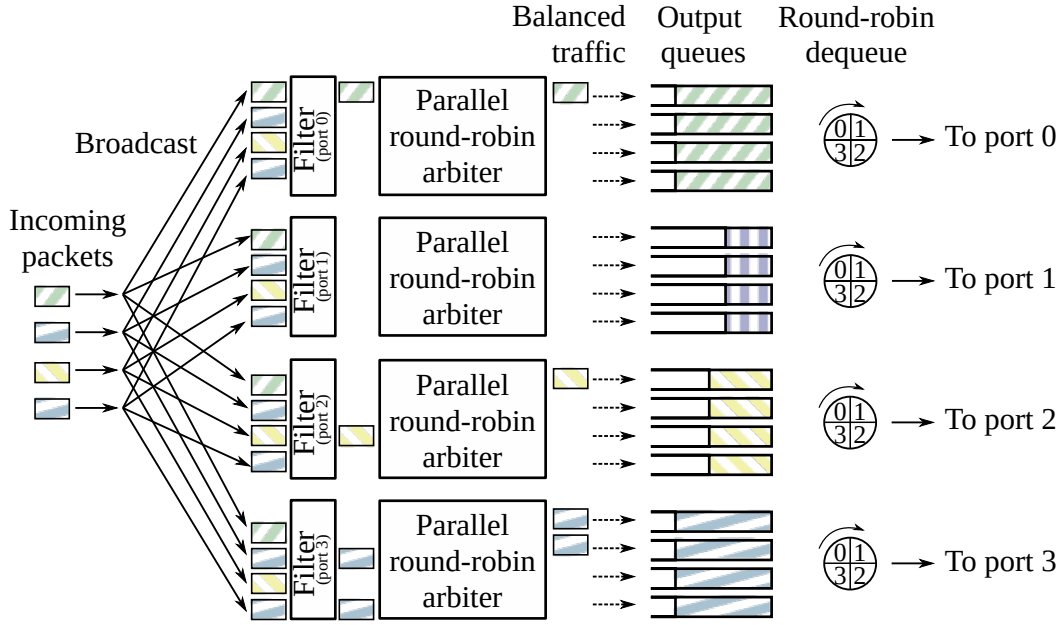


Figure 4.8: Unoptimised network switch based on the parallel round-robin arbiter, $P=4$

offsets pipeline is longer, shift registers are added after the sorting network. Otherwise, if the sorting network is lengthier, the offsets pipeline is simply moved to the right, and the adder trees can read the valid and destination fields from a later stage. As with the parallel round-robin arbiter, this is possible due to the fact that during the sorting network pipeline, a set of up to P packets is only getting permuted, and that the addition operation is commutative and associative.

The arbiter architecture is still fully pipelined and processes up to P packets at line rate. At its output, all inserted packets are distributed into the output queues. The pipeline logic is non-blocking, and any dropped packets can only occur at the queue level.

Following is a high-level description of how the building blocks interact with each other to enable unicast switching:

- The sorting network is used to sort the incoming batch of P packets according to their *valid* and *dest.* fields. The goal is to have all packets going to the same port arranged in consecutive order, before feeding them to the rotators. The sort priority is given to the invalid packets, thus the comparison in the CAS units is done on the concatenation $\{\text{valid}, \text{dest.}\}$, which is $1 + \log_2(P)$ bits wide.

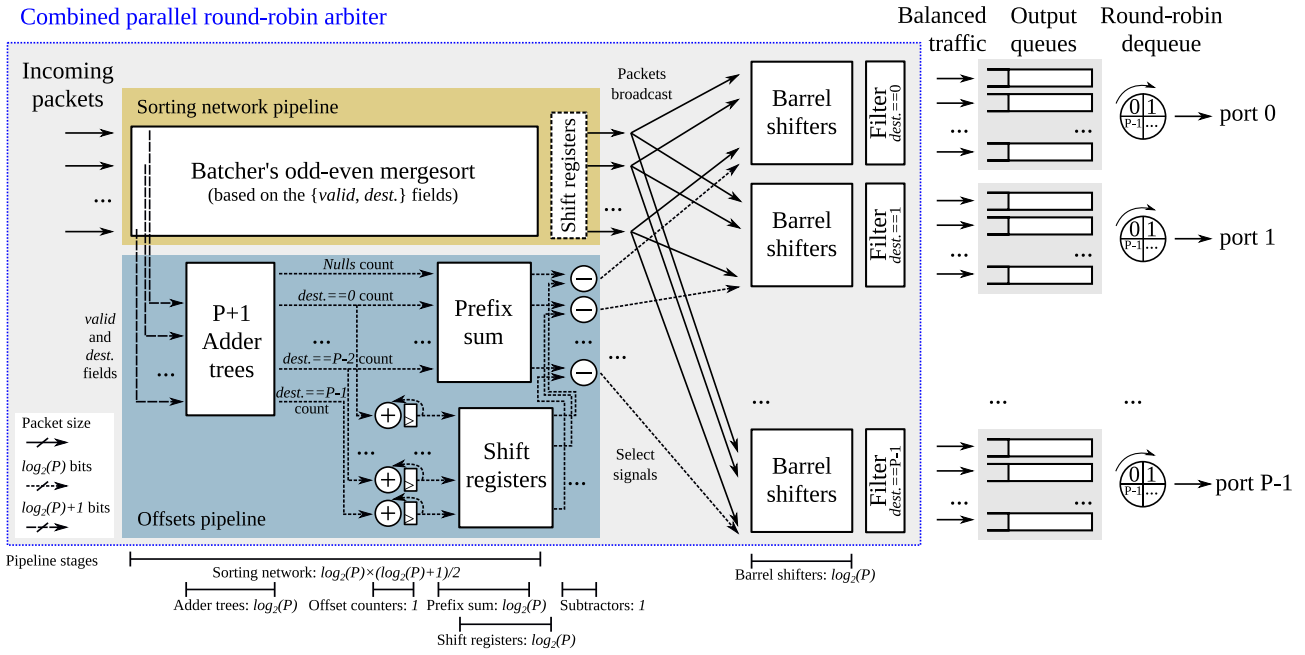


Figure 4.9: The proposed high-performance FPGA network switch

- The adder trees are used to perform popcount, i.e. to count the number of packets coming in a batch of up to P , that satisfy certain criteria. There are $P+1$ adder trees for P inputs. The first tree counts the number of packets with their valid bit unset (*Null* or non-existent). Each one of the other P trees are counting the number of packets going to each of the P ports. This information is later used by the prefix sum, as well as to update the *offset counters* (keeping the ‘current’ write rotation index of the output queues).
- The parallel prefix sum is used here to identify the starting positions of each group of packets going to the same destination port, within the sorted batch of P packets. The inputs of the prefix sum are the P out of the $P+1$ popcounts, starting from the *Null* count for in_0 , the $(dest. == 0)$ count for in_1 , and up to the $(dest. == P - 2)$ count for in_{P-1} . Since the priority is given to *Null* packets in sorting, the out_0 wire of the prefix sum represents the start index of the packets going to $port_0$, or the index just after the *Null* packets. Those starting positions are used to calculate the final amount of rotation each copy of the sorted batch of P packets needs, before writing them to the FIFO groups, in order to achieve the round-robin effect.
- The subtractions occupy one pipeline stage and their purpose is to produce the select signals for the rotators. Each rotator performs a rotation equivalent to two rotations,

one to the left, for bringing the packets going to the respective port to position 0 in the sorted packet batch, and one to the right, to achieve the round-robin effect. In other words, the rotation will follow the “current” rotation offset for correctly balancing the corresponding queues (positive offset), after “normalising” the starting position of the consecutive packets to 0 (negative offset). The subtractor calculates the addition of the positive offset to the negative offset. The positive offset corresponds to the (delayed) queue rotation offset for the port, produced by the *offset counters*. The negative offset corresponds to the respective output value of the prefix sum, i.e. out_0 the start index for packets to $port_0$, etc.

- The barrel shifters working as rotators are useful for performing the final rotation on the batch of 0 to P consecutive packets destined to each port. Note that the rotations are done on all packets in the batch, as it is broadcasted to all rotators. The output of each of the rotators is filtered afterwards to ensure that it only serves the respective destination port.
- The filters check the validity of a packet or whether the destination field equals a specified port number (an integer between 0 and P-1). Therefore, an equality check is performed for only up to $\log_2(P)$ bits, which has little timing slack and no additional pipeline register was added for it.

4.4 Properties and optimisation

Following is a list of properties and optimisation for the high-level functionalities, as well as specifically for the provided implementations.

4.4.1 Non-blocking

A stream processor is called *non-blocking* when it can process everything as it arrives, without delaying the output or applying backpressure on the input. It is commonly used in the context of circuits that perform permutations. *Backpressure* is supported when a stream processor can indicate with a *ready* signal whether it is able to process the input so that its master or producer will wait for it to accept the next data (or retransmit it, according to the system). An alternative to backpressure is to just drop the incoming packets, such as when it is acceptable by the communication protocol. Classic examples of non-blocking circuits are crossbars, sorting networks and various non-blocking permutation networks, including Beneš networks.

The parallel round-robin arbiter does not necessarily implement all possible permutations. For example, the stable version places consecutive packets consecutively in output (or cyclically). Still, it can accept everything as it arrives (being non-blocking), for all permutations of the input, and the output permutation is decided internally (being self-routing) by the compare-and-swap units and the adders.

Note that the definition of the non-blocking property could be specific to the application. For example, crossbars can do more than only permutations, including broadcasting (one-to-all), and for their supported mappings are always considered non-blocking.

On the other hand, in the case of interconnects and network switches, it can be misleading to call them non-blocking, because they support more mappings than permutations. They support multiple inputs going to the same output, as each incoming packet has a designated address field. This is true even if they are implemented with a crossbar internally, because each output port can only accept a single packet per cycle. Though it can be fair to characterise internal components as non-blocking, including the *combined* parallel round-robin arbiter and the crossbar. Therefore, memories are required in network switches, in order to appropriately delay packets and tolerate some levels of uneven traffic for some time. Theoretically, infinite amounts of memory would be required to call a network switch non-blocking for all input traffic patterns.

Unlike the network switches, the parallel round-robin arbiter requires no additional memory or FIFOs to achieve the round-robin effect, as everything can be processed immediately. The same holds for the combined parallel round-robin arbiter, before also adding the queue system near the ports to build a network switch.

In this chapter, it is assumed that the number of input ports is the same as the number of output ports (P). If the number of output ports was lower than the number of input ports, then the parallel round-robin arbiter would not be non-blocking, and would require FIFOs to tolerate short intervals of high utilisation (more incoming packets than the number of output ports). In such cases, if the number of inputs is different to that of the outputs, the parallel round-arbiter would also work as a rate converter.

With respect to the *combined* parallel round-robin arbiter for implementing a network switch, the number of output ports can be lower than the number of input ports while still being non-blocking (not the switch in its entirety). This is true as long as there are enough outputs/FIFOs for the queue groups near the output ports, to be able to forward every packet combination in a single cycle.

4.4.2 Scalability

The proposed implementations have no signal going from a pipeline stage to the input of a previous stage. In other words, they are “feedback-less” pipelines (as with FLiMS from chapter 3), or each pipeline stage can operate independently from the next stages during the FPGA clock cycle. This helps with timing closure, as the critical path does not directly increase by increasing the number of inputs.

An important advantageous property is that both the PRRA and the CPRRA are “self-routing”. This means there is no additional routing algorithm for deciding the 2x2 switch states (equivalent to the multiplexers and CAS units here). This is inherited by the building blocks, such as from the self-routing property of the sorting network, when used as a permutation network. For instance, a Beneš network is more efficient than the sorting networks in terms of num-

ber of pipeline stages for implementing all permutations, but it is known to require a routing algorithm “more complex than the network itself” [154].

A concern for scalability for the combined parallel round-robin arbiter is its hardware resource utilisation. According to the FPGA technology, a high utilisation could negatively affect the timing characteristics, as well as the routability. With higher numbers of ports this becomes a concern, although a high-performing 16-port switch on a high-end FPGA is presented in the evaluation section. It yields higher performance than previous FPGA-based solutions, but reducing the hardware complexity could possibly make it competitive towards commercial switches as well.

4.4.3 Complexities

In terms of hardware resource utilisation for the parallel round-robin arbiter, the fastest growing of all basic building blocks is the sorting network. The hardware complexity of the bitonic sorter or the odd-even merge sorting network is $O(P \times \log^2(P))$ CAS units.

For implementing the proposed network switch, the sorting network, having the highest complexity, highlights the importance of the “combined” aspect of the *combined parallel round-robin arbiter*. Its reduced hardware resource utilisation makes its implementation practical on FPGAs for a fairly high number of ports. The reduction to just one sorting network reduces the overall hardware complexity to that of the P barrel shifters, which sums to $O(P^2 \times \log(P))$ multiplexers.

The pipeline length of the parallel round-robin arbiter ($latency_{PRRA}$) is the sum of that of the sorting network and the one of the barrel shifter, since the adder tree pipeline stages overlap the last $\log_2(P) + 1$ stages of the sorting network.

$$\begin{aligned}
 latency_{PRRA} &= latency_{BS} + latency_{SN} \\
 &= \log_2(P) + \log_2(P) \times (\log_2(P) + 1)/2 \\
 &\in O(\log^2(P))
 \end{aligned} \tag{4.1}$$

For brevity, this formula only considers P values power of 2, and greater than 2, so that the adder tree pipeline can be placed across the sorting network stages. In a PRRA of two inputs, the sorting network is shorter than the adder tree pipeline, and could be implemented with one additional set of pipeline registers after the sorting network. However, for such a small size, it would be more appropriate to use a combinational circuit (i.e. with a truth table), as the focus of the pipelined structure is scalability, especially for the routing characteristics.

For network switches, latency is one of the main contributors to the port-to-port latency, as it is added as a processing latency, even when the output queues are empty. Here, the latency of the combined round-robin arbiter ($latency_{CPRRA}$) is similar to $latency_{PRRA}$ because the multiple barrel shifters operate in parallel, but the offsets pipeline can sometimes surpass the latency of the sorting network pipeline. $latency_{CPRRA}$ is calculated as the sum of the barrel shifter latency ($latency_{BS}$) and the maximum of the two pipelines at the beginning, which are the sorting network ($latency_{SN}$) and the offset calculation logic ($latency_{offsets}$).

$$\begin{aligned}
 latency_{CPRRA} &= latency_{BS} + \max(latency_{SN}, latency_{offsets}) \\
 &= \log_2(P) + \max\left(\frac{\log_2(P) \times (\log_2(P) + 1)}{2}, 2 \times \log_2(P) + 1\right) \quad (4.2) \\
 &\in O(\log^2(P))
 \end{aligned}$$

As is, the resulting $latency_{CPRRA}$ is not competitive for network applications. However, in the following subsection, an optimisation is presented to improve the port-to-port latency (including $latency_{CPRRA}$) significantly, that makes it competitive for network switches.

4.4.4 Reducing the pipeline latency

The logic complexity of each stage, although highly parallel, is relatively lightweight. According to the application of the parallel round robin arbiter, different numbers of bits are required to be compared in the sorting network. In the case of the combined parallel round-robin arbiter, the basic components are comparing, adding or subtracting up to $\log_2(P)+1$ bits in the entire design. Comparing only up to $\log_2(P)+1$ bits is also the case for the stable version of the

parallel round-robin arbiter shown in figure 4.5. This allows combining multiple pipeline stages in one cycle with minimal overhead in the critical path.

The pipeline latency can be reduced by removing the registers from all pipeline stages whose order is not multiples of S , where $S \in \mathbb{N}^*$ can approximately divide the pipeline latency (i.e. $latency_{opt} \approx latency/S$). In this way, although the complexity of the number of pipeline stages is more than that of competing solutions, this approach can still provide lower port-to-port latency for moderately high values of P . An example for dividing a sorter's latency by 3 is shown in figure 4.10.

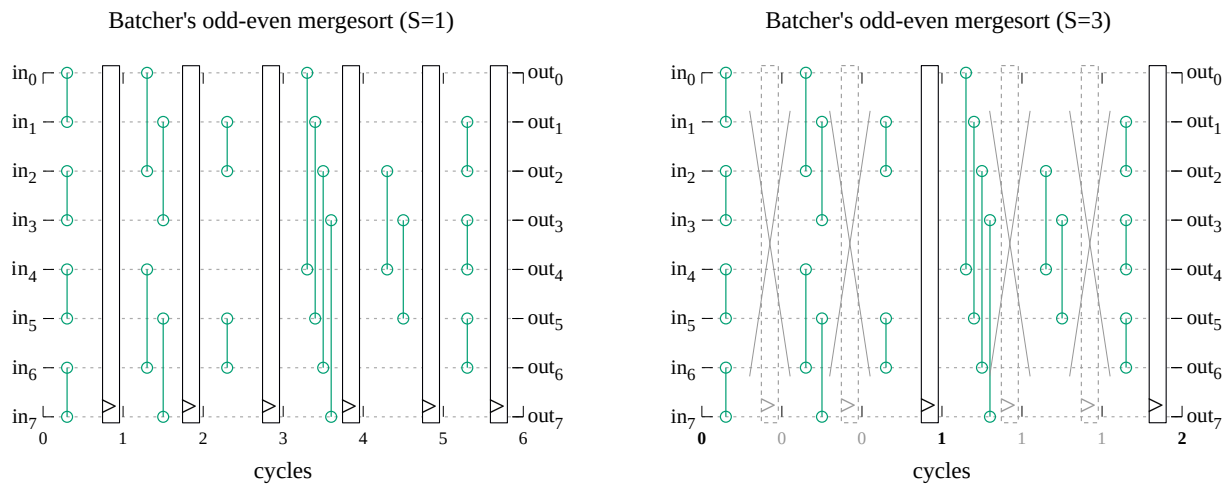


Figure 4.10: Dividing the latency of a sorter by 3, before (left) and after (right)

At the programming level, when there is a pipeline register corresponding to a stage without registers (when $S > 1$), the registers are replaced with wires. In this way, the notion of stages is kept for the sake of simplicity. For instance, a shift register of length 4 and $S=2$ is essentially still implemented as a shift register of length 2.

Note that this optimisation is very similar to an already existing technique in current FPGA tools, which is register balancing throughout retiming [155]. The idea is that the toolchain can explore the potential placement of registers in an automated and more fine-grained way than here. This is not considered, in an attempt to be vendor agnostic, to keep the notion of cycles consistent, and to simplify the design space for providing more in-depth insights on the impact of the registers in the design. For instance, since the comparisons are done on a small subset of the data for the switch (on up to $\log_2(P) + 1$ bits), the option to add more registers than

stages would be redundant and is omitted. Expressing the switch in a combinatorial form and optimising for a specific target with actual transceivers are part of the future work.

4.4.5 Calculating the optimised latency

The only pipeline stage whose registers cannot be removed is where the offset counter is updated. On the example of figure 4.5, this is the third stage. On the combined parallel round robin arbiter, P adders are keeping the queue offsets for the P ports on the same pipeline stage, as shown in figure 4.9. These adders are the only place where a feedback exists in the pipeline, as the output of each adder is added back as a way to update the respective offset. Fortunately, this feedback is always one stage long and does not contribute to the critical path.

In both the PRRA and CPRRA, the placement of the compulsory register moves according to the number of inputs. The compulsory register in the parallel round-robin arbiter is in the last stage of the sorting network. For the combined parallel round-robin arbiter, it is $\log_2(P)$ stages before the common ending of the sorting network and offsets pipeline. The index of the compulsory register stage is denoted as $index_{comp,PRRA}$ and $index_{comp,CPRRA}$ for PRRA and CPRRA respectively, and it represents the index of this pipeline stage starting from 0. These are calculated as:

$$\begin{aligned} index_{comp,PRRA} &= latency_{SN} - 1 \\ &= \frac{\log_2(P) \times (\log_2(P) + 1)}{2} - 1 \end{aligned} \tag{4.3}$$

$$\begin{aligned} index_{comp,CPRRA} &= \max(latency_{SN}, latency_{offsets}) - \log_2(P) - 1 \\ &= \max\left(\frac{\log_2(P) \times (\log_2(P) + 1)}{2}, 2 \times \log_2(P) + 1\right) - \log_2(P) - 1 \end{aligned}$$

Keeping the above in mind, the optimised pipeline latency ($latency_{opt}$) is calculated as follows.

$$latency_{opt} = \lceil \frac{latency - index_{comp}}{S} \rceil + \lceil \frac{1 + index_{comp}}{S} \rceil - 1 \quad (4.4)$$

Table 4.1 illustrates how the latency reduction optimisation significantly reduces the pipeline latency. A typical operating frequency of 150 MHz was selected to enable translating the pipeline latency from cycles to nanoseconds, before the actual place-and-route in the evaluation.

Table 4.1: Indicative pipeline latency after register reduction

	PRRA							CPRRA							
Number of ports (P)	4	8	16	32	64	128	256	2	4	8	16	32	64	128	256
Pipeline length	5	9	14	20	27	35	44	4	7	10	14	20	27	35	44
Compulsory reg. index	2	5	9	14	20	27	35	1	2	3	5	9	14	20	27
Pipeline latency, S=1	5	9	14	20	27	35	44	4	7	10	14	20	27	35	44
S=2	2	5	7	10	13	18	22	2	4	5	7	10	14	18	22
S=4	1	3	4	5	6	9	11	1	2	2	4	5	7	9	11
S=8	1	1	2	2	3	5	6	1	1	1	2	3	3	4	6
Pipeline latency (ns) (<i>indicative</i> , with S=4 and f_{clk} =150 MHz)	6.7	20.0	26.7	33.3	40	60	73.3	6.7	13.3	13.3	26.7	33.3	46.7	60	73.3

4.4.6 Optimality

The proposed implementations are not optimal in terms of hardware complexity, pipeline depth (latency) or with respect to synthesising it for specific technologies. They were designed to facilitate the needs of data analytics, as in subsection 4.2.3, as well as to provide a proof-of-concept for ideas related to the implementation of a high-performance network switch (see section 4.3).

For example, with respect to selecting a sorting network, for a specific number of inputs (P) there are other sorting networks, which feature optimal depth and/or the number of comparators (see section 2.3.1). The depth contributes to the pipeline length, thus the number of registers are affected, which contribute to the hardware utilisation.

A category of additional optimisations would be about variations in implementation of the structures on the target FPGA architecture. For instance, adder trees have been discouraged

for FPGA implementation, at least for certain Xilinx architectures [156]. This is because 6-input LUTs are able to add three numbers together, hence this binary tree approach being considered wasteful. An alternative recommended approach is to use the available DSPs [156]. Additionally, there have already been optimisations on the basic building in other contexts, such as halving the barrel shifter pipeline latency by using 4-to-1 multiplexers in ASICs [157], and combining pairs of compare-and-swap units (CAS) when consecutive pairs operate on 3 wires [74]. Nevertheless, these recommendations would need further experimentation for adoption here, because the adder tree recommendation was not for pipelines, and combining multiplexers and CAS units could break the complexity regularity between the pipeline stages that allowed efficient register reduction optimisation.

Mainly speaking about the parallel round-robin arbiter, the directives behind the proposed implementations were scalability, portability and applicability for the studied use cases. See the future work section 6.2.2 for a discussion on an alternative implementation of the parallel-round robin arbiter (PRRA), where a banyan network-based approach is shown to achieve the required routing with a fraction of the routing logic, but the control logic itself is currently more expensive.

Finally, it would be important to also refine the usage of memories at the output ports of the network switch. The $P \times P$ complexity of the queues could be limiting, such as for a high number of ports and a queue implementation using block RAM, which can quickly reach the limits of the device [13], [144]. Shared-memory techniques for network switches have some shortcomings, such as the requirement for running at multiples of the base frequency in a technique that has also been used in an FPGA implementation [13]. Since CPRRA outputs packets in consecutive positions, this could simplify memory-sharing for inclusion in a future iteration of this switch.

4.5 Evaluation

For the purposes of evaluating the parallel round-robin arbiter, only the network switch implementation (that uses the *combined* aspect) is considered. Evaluating this use case as a network switch is more complex, but yields easily-relatable performance metrics such as the port-to-port latency and the throughput. Due to its small size, the parallel round-robin arbiter itself is less challenging to implement on FPGAs, especially in its use cases for database acceleration, that also use a relatively small number of input ports. The evaluation of its database uses exists in chapter 5.

The novel network switch [39], that uses the combined parallel round-robin arbiter, is evaluated in two ways:

1. As a switching algorithm: This part of the evaluation is conducted exclusively in software using high-level simulations [146] without describing hardware. The purpose of this approach is to focus on the algorithmic aspect of the proposal, such as the time a packet stays in the queues, the queue occupancy and packet losses under different traffic patterns. Any hardware implementation specifics are ignored, such as the operating frequency, the pipeline latency, the port bandwidth and the packet size. A series of Python scripts have been developed from scratch to achieve a comparison with competing approaches.
2. As an FPGA design: The switch is implemented in Verilog HDL [146], with the help of a generator script written in Python to parameterise the switch design. There are two wrappers for the generated Verilog file, one to make it work as an AXI peripheral and one that includes a testbench for debugging in simulation. Originally, the AXI peripheral version was useful for debugging in real hardware, such as a relatively small device in the Xilinx Ultra96 board featuring 154K logic cells. The same design was used for collecting FPGA resource utilisation reports for higher-end devices, as AXI peripherals are relatively portable among the current Xilinx devices.

4.5.1 As a switching algorithm

The combined parallel round-robin arbiter is considered an alternative to scheduling algorithms for crossbar switches, even though it does not use any crossbar. This is because the proposed network switch uses $P \times P$ queues, as in VOQs for input-queued switches. Another similarity is that it requires no speedup of memory or logic (e.g. through multipumping [158]), as the entire design runs at the same operating frequency. These scheduling algorithms are relatively simple, and have been incorporated in FPGA network switch implementations [144]. They are also present in less-traditional switch designs, such as inside the building blocks of hierarchical switches [13].

As this focuses on the algorithmic performance of the proposed approach, all competing scheduling algorithms are assumed to run on a switch using the same operating frequency and bandwidth and with a processing latency equal to zero.

The performance of the proposed design is compared against various scheduling algorithms. Specifically, the comparison is made against the following: parallel iterative matching (PIM [159]), round-robin matching (RRM [149]), iSLIP [149], dual round-robin matching (DRRM [160], [161]) and dual round-robin matching switch with exhaustive service (EDRRM [153]).

The progress in such algorithms has been rather iterative, in the sense that each approach improves on some aspects of a predecessor. The changes focus on improving ASIC implementation efficiency [149], [160], scheduling performance and/or their behaviour under a wider spectrum of traffic patterns [153], [162].

Although all can work as iterative algorithms, some of them are originally presented as 1-iteration approaches, due to their simplicity and relative-efficiency. Nevertheless, both their single-iteration and their multiple ($\log_2(P)$) iteration versions are considered.

With respect to the traffic, the proposed approach is evaluated for the following patterns:

- (a) *Uniform Bernoulli arrivals*: At each input port, in each cycle a packet is sent with a probability r (input rate). The destination port of each packet is any of the P ports, all

appearing with an equal probability $1/P$.

- (b) *Uniform bursty traffic*: The packets at each input are sent based on an independent Markov chain, which has three states: ‘On’, ‘Off’ and ‘New’. In each cycle at ‘Off’, no packet is emitted. At ‘New’, a packet is sent with its output destination selected uniformly among the ports. At ‘On’, a packet is sent to the same destination as the one of the previous packet. The transition probabilities of the Markov chain are shown in figure 4.11, where $r \in [0, 1]$ is the average input rate and $s \in \mathbb{N}$ is the average burst size. The proposed model is almost equivalent to the two-state model [162], but it does not limit the maximum input rate to $1 - \frac{1}{s+1}$. The default value of s is set to 32.

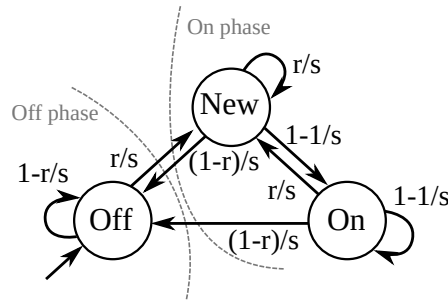


Figure 4.11: A uniform bursty traffic Markov model

- (c) *Nonuniform traffic* [153], [162]: At each input, in each cycle a packet is sent with a probability r . Its destination will be the same as the source with a probability p . All other outputs share an equal probability, i.e. $(1 - p)/(P - 1)$. The default value of p is set to 0.5.

Average latency The first experiment explores the average latency a packet waits inside a queue to be served. In this experiment, it is assumed that the $P \times P$ FIFOs have *infinite depth* and, thus, no dropped packets.

In figure 4.12, the proposed design is observed to consistently outperform the studied scheduling algorithms. For each of the scheduling algorithms a multi-iteration version and a single-iteration version is considered. The latter is denoted by appending “-1” to the name of the corresponding algorithm. For both the traffic patterns (a) and (b), the multi-iteration scheduling algorithms perform almost identically to each other. (It is their single-iteration performance that motivated

most of them). In traffic model (c), the only close competitor to this approach is EDRRM [153], which is selected here as an algorithm designed for bursty/nonuniform traffic. Still, for an input rate of near 1, the proposed approach reduces the average latency by 2x, 1.4x and 6x for the traffic models (a), (b) and (c) over the best alternative for each (PIM, PIM and EDRRM).

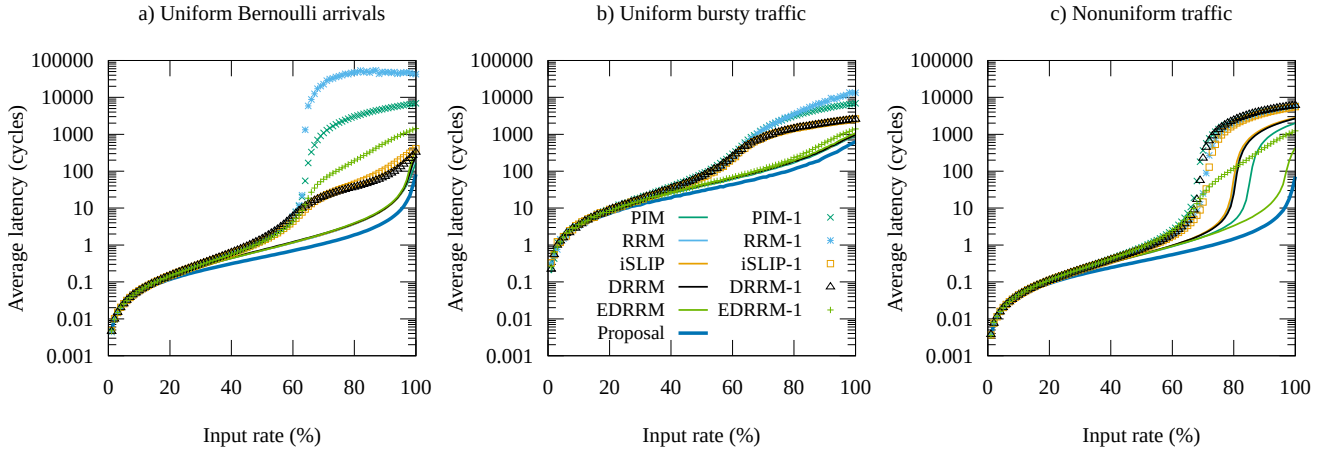


Figure 4.12: Average packet latency using different algorithms and traffic models (16×16 switch, simulation)

Worst-case latency Using a similar experiment, for a 16×16 switch, the proposed approach also yields the lowest average worst-case latencies. The best alternative algorithms remain the same for this metric. Using an input rate of near 100%, the best alternative per traffic model yields a latency of 1535, 7992 and 7212 cycles for the traffic patterns (a), (b) and (c). The corresponding values for the proposed design are 375, 2807 and 310 cycles, which translate to a reduction of the worst-case latency by 4.1, 2.8 and 23.3 times on average respectively. Figure 4.13 summarises the results of the worst-case latency experiment.

Note that this difference in performance is expected and comes from the fact that these scheduling algorithms involve a crossbar with virtual-output queues (VOQs). VOQs limit the possible paths from the queues to the output ports, as illustrated in figure 4.6. This leads to additional delays for higher input rates, and is reflected on the worst and average latency. The proposed approach is not a scheduling algorithm for VOQ-based switches. However, it is included in this comparison, as it combines some of the advantages of this family of switches related to FPGA implementation. Alternative approaches exist and they exhibit a better algorithmic

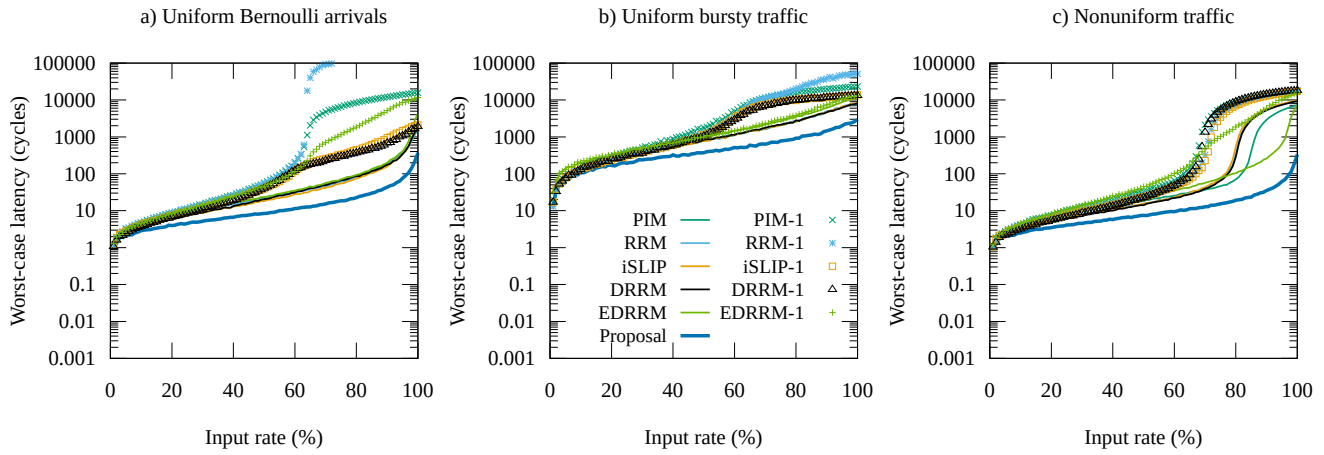


Figure 4.13: Average worst-case packet latency using different algorithms and traffic models (16×16 switch, simulation)

performance than VOQ-based switches, but require a logic or memory speedup [145].

Impact of queue size As a third experiment, the queue size requirements are explored. The best alternative algorithms are still PIM, PIM and EDRRM for the traffic patterns (a), (b) and (c) respectively. The approach is to measure the dropped packet rate using different FIFO lengths. The FIFO depth metric is a consistent way to measure the total buffer size in the simulations, as both virtual output queues (VOQs) and the proposal’s output queues are $P \times P$ buffers.

Figure 4.14 shows the dropped packet rate with respect to the queue size for the proposed approach, as well as for the best alternative for each of the traffic patterns. The proposed approach requires a small fraction of memory to support 100% throughput, i.e. a dropped packet rate equal to zero (represented by the colour white). For the traffic pattern (a) and an input rate near to 1, the proposal achieves a 4.6x reduction in FIFO depth (153 to 33) for supporting 100% throughput, while the dropped packet rate for a FIFO depth of 1, drops from 16.1% to 5.2%. For the traffic patterns (b) and (c) and input rate close to 1, the other approaches require a FIFO depth beyond the design space, which is a sign for a significant performance bottleneck, as the memory’s order of magnitude approaches the one of the number of inserted packets. Nevertheless, the proposed approach exhibits a better behaviour. Moreover, for the traffic pattern (c), it is able to handle nonuniform traffic with marginally less requirements

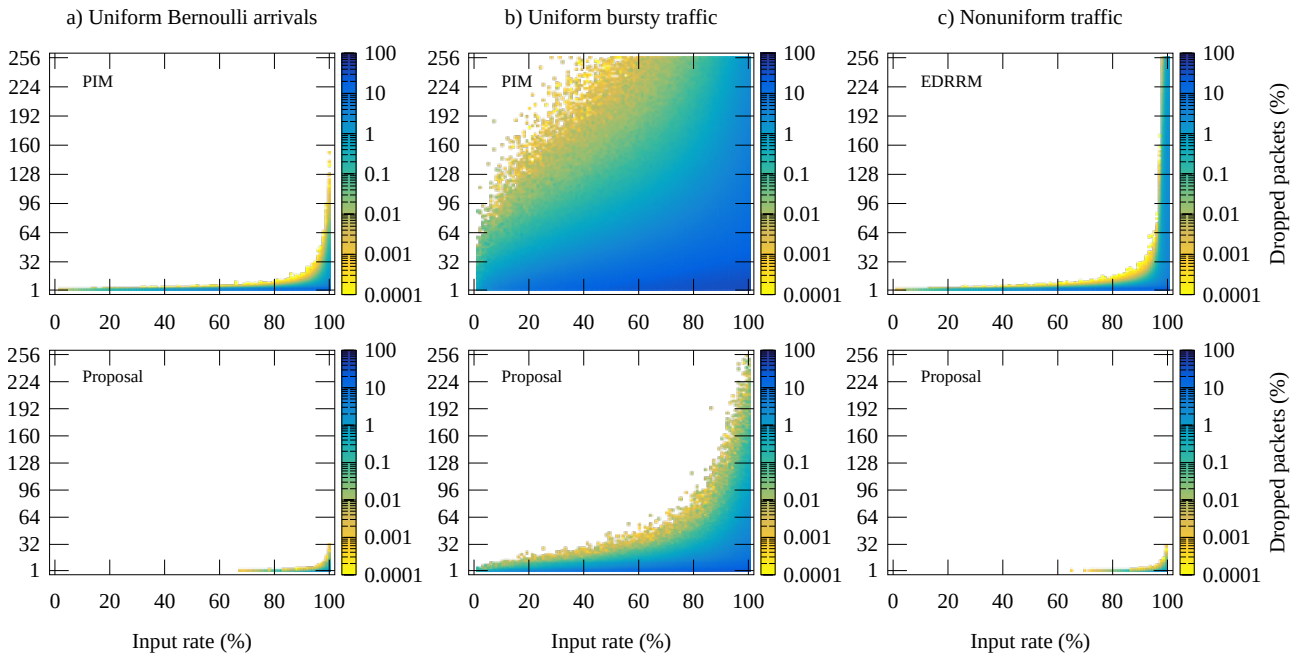


Figure 4.14: Impact of queue size, for the proposal and the best alternative per traffic model (16×16 switch)

than in the Bernoulli case, while this pattern becomes much more challenging for the other approaches.

4.5.2 As an FPGA implementation

This exploration focuses on the performance and timing characteristics of the main logic, which is the combined parallel round-robin arbiter, on a relatively high-end Xilinx FPGA. Starting with the Verilog generator script, a custom network switch encapsulated in an AXI peripheral is produced for a custom number of ports (P), packet size in bits and a custom latency reduction parameter (S). If P , packet size and S are restricted according to the requirements and feasible switch sizes, this results in a relatively narrow design space. The resulting switch configurations are integrated into a simplistic Vivado project for the Xilinx Alveo U280 board, with debugging functionality.

Note that this is an indicative and out-of-context approach, as no transceiver is populated and the aggregate FIFO space is only P packets per port. Thus, this evaluation does not use block RAM, and studying memory implementations and memory sharing on FPGAs is part of future

work.

Resource utilisation: Figure 4.15 introduces the resource utilisation metrics as reported by Vivado for the switch design space. Each data series includes the switch designs for the same port and packet configuration under different S values. The dashed lines represent the LUT and FF capacity of the test platform on the corresponding axis. As can be observed, the number of ports is a big contributor for both the LUT and FF utilisation and can become a limiting factor for a high number of ports. The packet size is shown to impact linearly the LUT and FF utilisation, as the 512-bit versions of the equivalent configurations for 256-bit packets have roughly double the resources. The S parameter for latency reduction does not seem to have a big impact on the LUT utilisation. This is expected as the value of S only determines the number of pipeline registers and does not impact the rest of the logic of the switch. In contrast to LUTs, the number of FF registers can be significantly decreased by the register reduction optimisation (S approximately divides the pipeline stages). For instance, for $P=16$ and 512-bit packets, using approximately one quarter of the pipeline registers (using $S=4$, yielding around 3.5x stages reduction), the FF utilisation drops from 755K to 305K (around 2.5 times FF reduction). Appendix B presents the resulting dataset from this exploration in more detail.

Performance: Figure 4.16 introduces the physical implementation performance results through the throughput and port-to-port latency metrics for the design space up to 16 ports (see appendix B for the complete dataset).

As can be observed from figure 4.16, among all 512-bit packet switches, there is at least one configuration that exceeds 100 Gbps per port for each explored number of ports up to 16. For 256-bit packets, it seems more challenging to get high throughput for a similar range of operating frequencies, but there are still points exceeding 100 Gbps for ports 2, 4 and 8, but not for $P=16$. For $P=16$, one good performing configuration from the design space is for $S=2$ and 512-bit, which achieves an aggregate throughput of 1.7 Tbps and a port-to-port latency of 39 ns. Note that this specific data point required aggressive optimisation to ease routing.

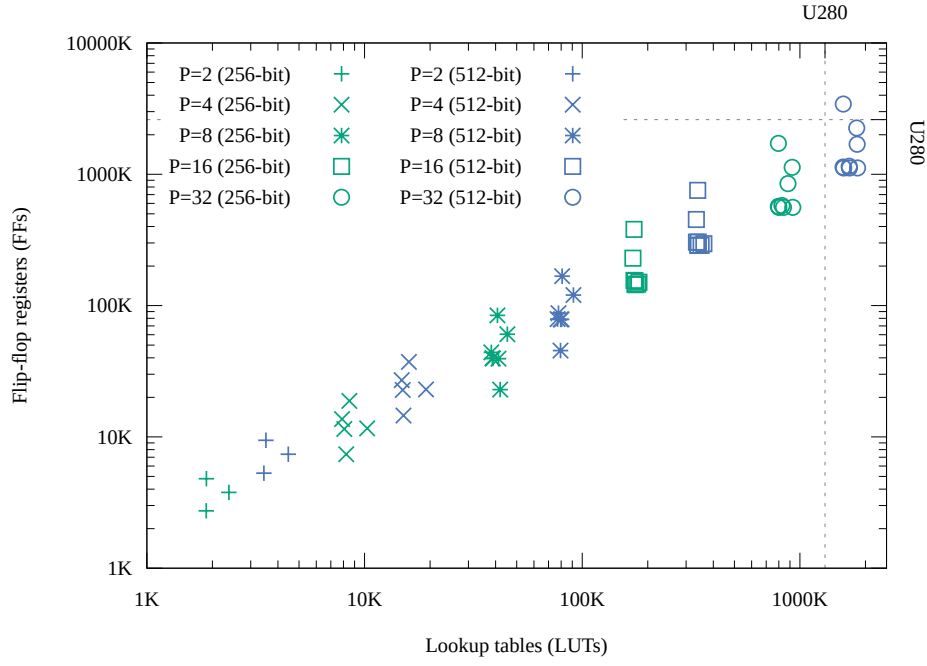


Figure 4.15: Flip-flop register (FF) and lookup table (LUT) utilisation

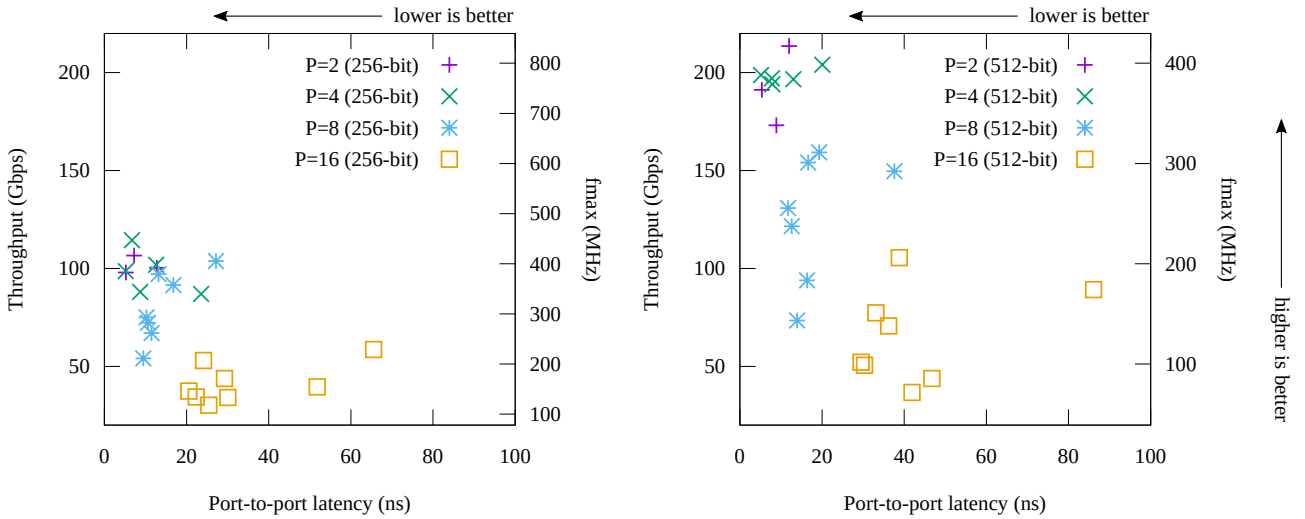


Figure 4.16: Performance for switch designs with 256-bit (left) and 512-bit (right) packets

Comparison with FPGA-based related work

In this subsection the 16×16 switch implementations [146] are selected for comparison to other FPGA-based network switches. The other switch implementations here were redeveloped for the same target FPGA (Xilinx Alveo U280) by Jiuxi Meng. The dataset is updated according to the journal extension [44]. Note that this comparison is mostly indicative, as some assumptions were made for each implementation, such as to be out-of-context and featuring different buffer

lengths and memory types.

An FPGA network switch implementation is GCQ [13]. It improves on a hybrid buffer approach, the hierarchical crossbar switch [152]. The idea is to improve the scalability for a higher number of ports by using smaller switches hierarchically. Due to the fact that it requires a logic speedup, the yielded base frequency was 40 MHz, resulting in an aggregate throughput of 160 Gbps for a 16×16 switch, but was originally evaluated on older architecture (Xilinx Virtex 6-240T).

A more recent switch implementation on an FPGA is SMiSLIP [144]. It is an input-queued switch, implemented as a crossbar with VOQs, and controlled by the iSLIP scheduling algorithm [149]. The focus of this work is the optimisation for FPGAs to achieve buffer-sharing and a better performance. Despite the relatively high operating frequencies, SMiSLIP only produces output once every $3 + \log_2(P)$ cycles. For this reason, the throughput is calculated as $\frac{1}{3 + \log_2(P)}$ of the line speed.

This is expected for iterative matching algorithms, such as iSLIP. One of the original iSLIP-based prototypes [139] is an ASIC operating at 175 MHz in a 32-port switch. Despite its optimisations such as the merging of the 2 out of its 3 phases (request, grant and accept), it is not fully-pipelined. It only produces output once every 9 cycles, reducing the maximum throughput to only 1/9 of the speed the fabric. It does not necessarily mean that the lines are useless during the in-between cycles. In this case, a packet is broken down into 9 pieces and is sent progressively. Thus, higher throughput can be achieved, but with fewer decisions (crossbar configurations) between the same amount of data. In other words, a larger packet size is needed to fulfil higher throughput, but this can quickly become wasteful and inefficient on FPGAs.

Figure 4.17 shows how the design space for the proposed design performs against the related work, based on the throughput and port-to-port latency. For a packet size of 256 bits and $S=4$, the proposal achieves a throughput of 53 Gbps, which is around 8.5 and 4.6 times the throughput of the 256-bit versions of SMiSLIP and GCQ respectively. SMiSLIP sometimes marginally wins for port-to-port latencies, but the design space for the proposal features choices that are better in both the throughput and latency, including the aforementioned data point,

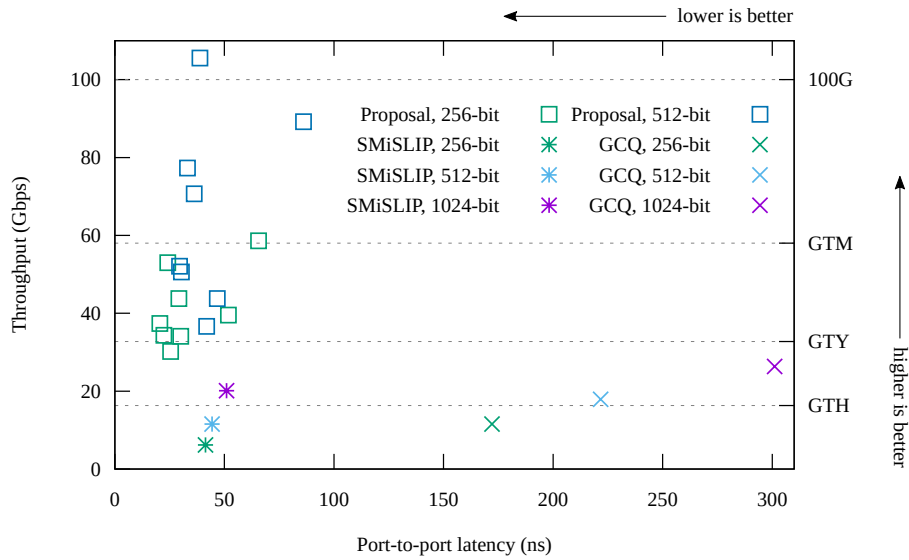


Figure 4.17: Comparison of 16×16 switches implementations on Xilinx Alveo U280

which has a latency of 24.2 ns. Also, since SMiSLIP uses the iSLIP scheduling algorithm, there will be additional latency for high input rates, as demonstrated in simulation. GCQ yields the worst latencies irrespective of packet size. For a packet size of 512 bits and $S=2$, the proposal achieves the highest throughput overall at 105.5 Gbps and a latency of 39 ns, saturating the UltraScale+ GTY and GTM transceiver, as well as for 100 Gbps or equivalent.

In terms of hardware resources utilisation, as a direct comparison with GCQ [13], for a 256-bit 16×16 switch the $S=4$ implementation consumes around 3.5x the LUTs and 4x the FFs. It is important to note that it would be more appropriate for this comparison to be made against a competitor switch of wider packets for achieving similar throughput, as demonstrated. In conclusion, this design provides state-of-the-art performance for unicast, but with a relatively high resource utilisation.

4.6 Summary

The parallel round-robin arbiter is a non-blocking load balancer circuit that is easily pipelined and keeps the order of the input among its output ports in round-robin priority. It has applications in parallel processing, such as for appropriately removing the sparsity for aggregation tasks, including the use cases of chapter 5 in sort-merge joins and distinct counts. The

occurring sparsity can be considered as an intermediate step for accelerating other database applications in streaming processors. The general concept of the pipelinable logic is also applicable to network switches, for building a high-performance FPGA-based network switch. Its variation called the “combined” parallel round-robin arbiter achieves high switching performance by emulating the performance of the output-queued crossbar, achieving better performance than common scheduling algorithms. The pipelined design for the network switch is also shown to achieve efficient FPGA implementations, outperforming relatively recent FPGA-based switch implementations on the same FPGA.

Chapter 5

Accelerating database operators using FPGAs

This chapter presents novel designs for database operators known to consume considerable computation time for the specialised industrial use case, as outlined in section 1.1, as well as more widely. These are for sorting, joins and distinct counts.

FPGAs are increasingly being introduced as a platform to accelerate big data and database analytics [41]. A considerable fraction of such operations relies on sorting [37], [81], or their performance can be improved when operating on sorted data after making the appropriate algorithmic changes [24]. For this reason, the research for efficient FPGA-based sorters has been fairly active recently, with one of the main considerations being the disproportionate advantage of the throughput and latency performance with which the CPU accesses the main memory.

First, an end-to-end FPGA implementation of a sorter design is presented, that improves on the latest research about sorting acceleration on FPGAs [32]. It can also be considered a realistic use case for FLiMS [36] (chapter 3), which is a building block for hardware merge-sorters. This solution introduces some other interesting concepts and designs, such as the merge-capable linear sorter. The proposed sorter works for data that fit on-chip, with the additional functionality to merge sorted sublists recursively, for an input of arbitrary length.

While many-leaf mergers (merging hundreds of sorted lists simultaneously) are conventionally single-rate (a record per FPGA cycle), a novel technique in the presented approach is to use a parallel merge tree only for the latest stages of the merge tree. This achieves merging that is both multi-rate and many-leaf, and in combination with the flexibility of the generator script, it can produce designs for a variety of use cases, while saturating the memory bandwidth for uniformly random data.

Other presented accelerators are for equality joins (equi-joins) in relational databases, as well as for performing distinct count, to accelerate counting the number of distinct values in a column. Also introduced is a more demanding variation of a distinct counter that counts the distinct values in different groups.

The overall goal is to provide alternative designs to address some of the research challenges (section 1.2) to a greater extent than the related work, rather than a direct performance comparison. For instance, while there are alternative approaches and algorithms for performing these database operators in hardware, the presented approaches are based on algorithms yielding linear access patterns for easier performance gains on heterogeneous systems.

The sorter was originally introduced in the paper “An Adaptable High-Throughput FPGA Merge Sorter for Accelerating Database Analytics” [32]. The material relating to sort-merge join is based on the paper “Accelerating the merge phase of sort-merge join” [43]. The accelerator for the distinct count without groups has not been published elsewhere yet. The distinct counter with groups is based on the paper “An Adaptable High-Throughput FPGA Merge Sorter for Accelerating Database Analytics” [32].

5.1 An adaptable high-throughput FPGA merge sorter

This section presents a full system FPGA implementation of a sorter design, that improves on the latest research about mergesort on FPGAs [32].

A challenge a the hardware designer is to maximise performance while adhering to a number

of constraints related to the storage medium and data movement. These include the storage capacity, the data width and number of ports, the latency for different access patterns [42], address segmentation and NUMA-related restrictions. In research, using a representative storage medium is not always the case, such as with the majority of sorters mentioned in the related work, that are BRAM-based even though are destined for large-scale sorting.

When performing mergesort using FPGAs, there currently seems to be a trade-off for supporting either high-throughput/few-leaves or single-rate/many-leaves. For high-throughput mergers, a limitation of the current solutions is their scalability to a high number of inputs, and require a high aggregate bandwidth. The most recent variation has only reported merging for up to 64 inputs [74]. On the other hand, many-leaf mergers can merge a couple of thousand input lists simultaneously, which can have significant benefits due to the reduced number of data passes. A data pass involves reading every data from DRAM and writing them back once and is considered a performance bottleneck. However, an evaluation [42] that demonstrated saturation of the PCI Express bandwidth used 800-bit records, which may not always be the norm.

The presented sorting solution combines features from the state-of-the-art hardware designs for mergesort, while taking into account different architectural constraints. A sorter generator produces a hardware merge sorter in Verilog, with any desired datapath length, data width and payload width. The same logic is also used to perform the first sorting pass efficiently, that is to produce a series of sorted sublists, before merging them together. This functionality, while crucial to applicability, performance and resources utilisation, is absent from most large-scale sorters, which usually assume already-sorted sublists or rely on existing hardware sorter modules.

The provided open-source RTL generator [163] produces sorting peripherals with customisable parallelism and data format. The FPGA design is evaluated as a 128-bit-wide peripheral on an MPSoC platform, with a speedup of up to 49 times over a single A53 core for 32-bit integer sorting (see section 5.1.3 for more information on the evaluation).

5.1.1 Solution

The proposed design works in 3 phases internally, with regards to the data movement. As shown in figure 5.1, initially, the unsorted data are streamed from DRAM directly to the sorter, and the immediate results are stored to BRAM. These data in BRAM are sorted in chunks of k elements.

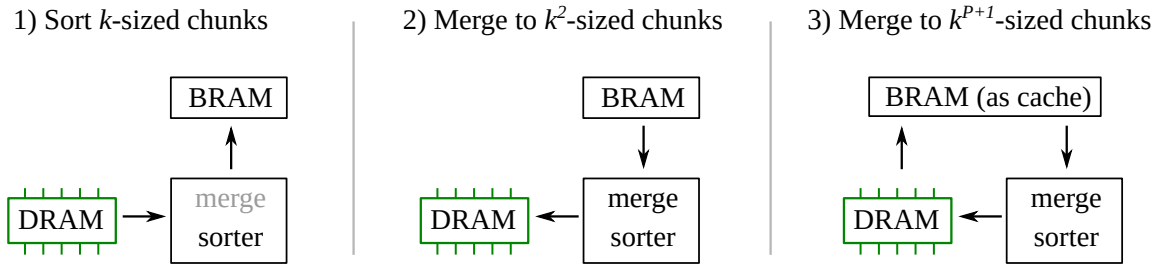


Figure 5.1: Data movement during the three sorting phases

In the second phase, the sorter works as a merge sorter of k sorted lists of length k from BRAM. In other words, the sorter is able to provide a sorted list of size $k \times k$ with a single pass.

The third phase is the merge operation from data in DRAM. The same phase applies to input lists of arbitrary length. Thus, the sorter can sort an input of size k^{P+1} , where P is the total number of passes. During this phase, the merger still reads from BRAM, which works as a cache to DRAM, to hide the latency from accessing with a random access pattern. This memory is already divided in k lists of length k , which now work as buffers and are filled upon request, with a throughput of w records per cycle.

See section 6.2.4 for a proposed future work that expands on this approach to take into account NUMA-related complications in datacenter FPGAs, such as with PCIe data transfers.

Following are detailed descriptions of the building blocks.

Merge-capable linear sorters

The linear sorter is modified such that it can also be used for merging a number of sorted lists. The idea is that a many-leaf merger essentially outputs the minimum head of all queues to be

merged, representing the minimum of all unprocessed input. And this can be emulated by a linear sorter. By appending the source list index to the value as a payload, we can keep track of the source of the extracted head. This is useful for fetching the next head to be processed on the immediately next cycle. In this way, we can keep exactly one head for each of the input queues, and appropriately represent the merge operation.

Figure 5.2 illustrates¹ an example for merging 8 lists (named A through H). The minimum head (1) is extracted as an output, as it represents the minimum of the heads of already sorted lists, which subsequently also represents the minimum of all input. The next head is indicated by the source list (A), and the next head (6 from list A) is dequeued to be processed on the next cycle.

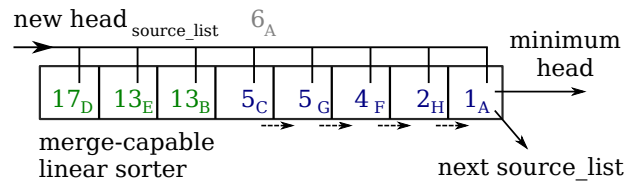


Figure 5.2: Merge-capable linear sorter example

In order to make it practical for general use, some additional functionality is added, such as for supporting list endings:

a) *Enqueue a ‘bubble’*: This is used to adjust merge capacity, as well as to support the end of an input queue.

b) *Flush current values*: Only useful in sorting mode, this is triggered automatically, as soon as the sorter fills up, in order to support full-streaming of multiple chunks for sorting (i.e. no additional latency between the processing of consecutive chunks). The cells with flushed values behave as shift registers to eventually propagate the sorted chunk, while ignoring new requests.

¹See [133] for a more comprehensive visualisation online.

Modified parallel merge tree

The modified linear sorters are appropriate for both sorting and merging, but they are limited to single rate input/output. In this subsection, we propose to use an upgraded parallel merge tree (PMT [69]), to merge the results of multiple merge-capable linear sorters. In this way, we can efficiently support a custom amount of throughput, as well as many-leaf merging.

The parallel merge tree here adopts FLiMS [36] (chapter 3) as a building block, an efficient high-throughput merger for 2 lists.

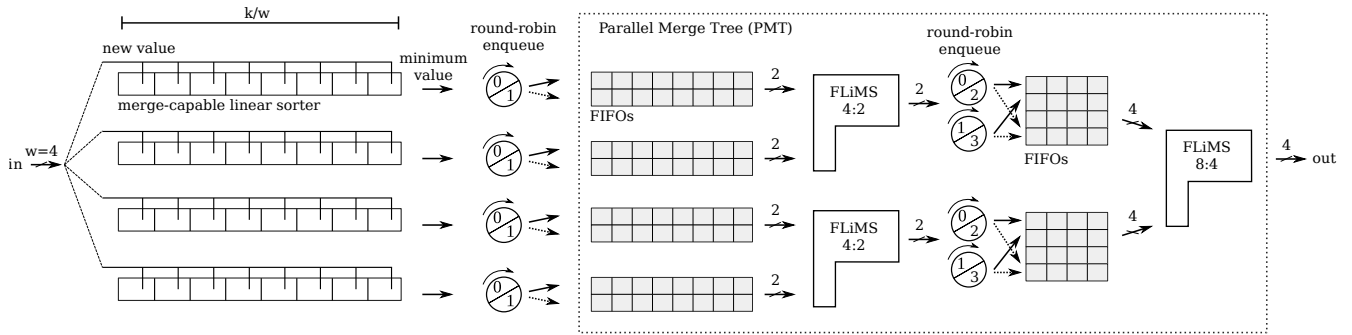


Figure 5.3: The merge-capable high-throughput sorter (for a throughput of 4). Wire widths are multiples of the value width.

Figure 5.3 shows how the parallel merge tree increases the throughput of the proposed merge sorter. First, there are w linear sorters, where w is the degree of parallelism. Each linear sorter has a length of k/w , where k is the total merge capacity or the sorted chunk size. This structure is able to both sort an input of k elements, and merge k sorted lists of arbitrary length.

In sort mode, the value of a two bit counter is appended to the most significant bits of all output of the linear sorters. This 2-bit counter is incremented when a new sorted chunk is flushed to the PMT. It is used by FLiMS to be able to give the correct sorting priority to independently sorted chunks. Note that one bit is not enough, as during the half-cleaners stage (MAX units of figure 3.2), between one value with a chunk order of 0 and one with 1, it is not possible to differentiate easily which came first.

On-chip buffers

This subsection describes each of the buffers required for efficiently merging or sorting using the presented approach. Note that there are many such design choices.

a) PMT FIFOs: In order for the first phase to be able to support bursts of length k , the FIFOs after the linear sorters have a depth of k/w elements. This is required because the linear sorters are flushed automatically when full. A ready signal represents the space availability for a forthcoming burst. With respect to the FIFOs of all levels in the merge tree, each FLiMS module of a previous level (as well as the linear sorters) must stall when there is no space left for its output.

b) Burst buffer: The output to DRAM is written in bursts. A burst buffer is near the output and is enabled from the second phase and beyond. When no space is left, the FLiMS of the last level in the PMT is stalled.

c) List buffers (BRAM): The block memory is divided in $w \times w$ banks. The first dimension is for the w linear sorters, which merge an independent range of k/w lists. The second dimension is for being able to fill a buffer with a throughput w . In the first phase (sort), the output of the PMT is written across all w partitions, with round-robin priority per sorted chunk, to distribute the merging workload evenly across the w linear sorters. In the second phase, each linear sorter reads from its own partition with a throughput of 1, and the merged output is redirected to DRAM.

d) Fill request buffers: In the third phase, the linear sorter functionality remains similar, and each list buffer of size k represents data in DRAM. When a list buffer is halfway empty, a request is stored in a fill request queue. When a list buffer is empty or a fill request buffer is full, a stall signal is triggered for the respective merge-capable linear sorter.

e) Next head buffers: For an efficient merge operation, there needs to be a buffer between the linear sorters and BRAM. This is because the BRAM introduces a latency of 1 cycle and this could potentially halve the overall throughput. The next head buffers prefetch the next head for each of the k lists, as soon as a head is read, so that the BRAM latency is hidden.

5.1.2 Optimisations

The following optimisations are proposed for increasing the throughput of the sorter according to the micro-architecture and capabilities of the target platform. The goal is to be able to consider these optimisations in the sorter design space (see section 5.1.3) to overcome some performance bottlenecks that can be observed when implementing on current FPGAs. Both of the following optimisations were used in the presented designs of the evaluation (section 5.1.3).

Input buffer rate

A design choice for balancing the resources utilisation for a specific device is to use additional BRAM for the 3rd phase (merging from DRAM). Instead of only reusing the BRAM of the 2nd phase (merging from BRAM, totalling $k \times k$ values), the Verilog generator script accepts one more parameter, the buffer rate (r). It represents the ratio of the BRAM to be used as input buffers in the 3rd phase over the BRAM required for the 2nd phase. By increasing r , the input buffers are lengthened accordingly, improving the performance when merging from DRAM. As the input buffers can be filled with longer DMA transfers, a higher throughput can be achieved [43] and the memory access pattern from DRAM becomes more linear.

Throughput

One complication from using a form of linear sorters in this solution could be a reduction to the operating frequency. This is because linear sorters require broadcasting to multiple comparator cells, that becomes expensive for many cells [67], given that the result is needed on the next cycle. In contrast, systolic sorters [68] could scale better, but are not suitable for implementing the proposed merge-capable design, because of the induced latency. The proposal solves this problem partially, due to the arrangements to achieve high-throughput. It essentially splits a sorter of length k to w linear sorters of length k/w . Therefore, each broadcast is more local, as there are w independent linear sorters that are w times shorter, for the same chunk size k .

An optimisation is proposed to further increase the operating frequency of the generated sorters and saturate the bandwidth of the port. Since this approach allows a custom degree of parallelism (w), a sorter with a width multiple of the datapath width can be generated instead. For example, by doubling the value of w and appropriately interleaving the input and output, the sorter logic can operate at half the operating frequency of the base clock. This emulates a sorter of half the value of w and double the frequency (as in the evaluation). This modification is relatively inexpensive, as only w changes in the logic complexity (see table 5.2) and w is relatively a small number. This optimisation can also help in situations where the value width is high and the comparator logic latency contributes to the critical path of the design.

5.1.3 Evaluation

In this section, we evaluate the sorter design with an example use case for sorting 32-bit integers. See the following sections 5.2 and 5.4 for different database applications that could utilise such a sorter. The latter (section 5.4) demonstrates a 64-bit use case based on the same design for distinct count.

A generator script written in Python [163] produces a Verilog sorter for user-specified values for w , buffer rate r (see below) and a flag to enable the FLiMS skewness optimisation (see section 3.2.1). The remaining parameters can also be modified after generation (and before synthesis): total merge capacity or inner chunk size k , value width d , comparison width (to facilitate payloads) and various queue lengths. The combination of w and the data width specifies the throughput of the sorter (e.g. $w = 4$ and 32-bit values for reading and writing at 128 bits per cycle). All integer parameters are in powers of 2, except the comparison width.

The evaluation platform was Avnet’s Ultra96 board, which combines a Xilinx Zynq UltraScale+ ZU3EG device with 2GB LPDDR4 RAM shared between the programmable logic and the four A53 (mid-range) ARM cores, running Linux at a frequency of 1.2 GHz. In comparison with earlier iterations of this FPGA architecture [83], one of its database-related advantages is that it provides 128-bit-wide ports for faster communication between the processing system (PS) and the programmable logic (PL).

The available memory is manually divided into two 1GB regions, one for the Linux environment, and one for the accelerator. The latter is uncached, and for fairness it is not used for evaluating the CPU software baseline. The generated peripherals have a 128-bit-wide datapath and most data transfers are done by a DMA engine. The software part of the sorter accelerator behaves as a callable C function.

The baseline for comparing to CPU sorting is GCC C++ standard library's `std::sort()` implementation (single-core), that is considered a widely-available well-performing baseline for evaluating high performance sorting in software [5]. All binaries were compiled using the `-O3` optimisation flag of GCC, and their execution is pinned to the same core/thread (using `taskset`). The micro-architecture of the A53 cores is advanced enough to include a data prefetcher in hardware [164]. The A53 also supports a basic set of 128-bit SIMD instructions [165]. Though, no manual vectorisation and fine-tuning of the `std::sort()` function has been done for this experiment to ensure that SIMD is used efficiently by the software stack in the default Linux image of Ultra96.

The first selected configuration was a sorter with $k = 128$, $w = 4$, 128-bit datapath and an operating frequency of 250 MHz. This frequency can saturate the achievable bandwidth of that port at 128 bits per cycle [77] and was achieved with the throughput optimisation (internally with $w = 8$). The equivalent design that has not used this optimisation (internally with $w = 4$) achieved an operating frequency of only 150 MHz. This example highlights the impact of the throughput optimisation on the operating frequency, though further discussion of suboptimal datapoints is omitted for brevity.

As shown in figure 5.4, this sorter achieves a speedup of up to around 36 times for data that fit in a single pass ($k^2 = 16384$) and above around 15 times for all other input sizes. A big contributor to performance is the number of passes, especially for the 2nd phase and above, as the memory access pattern becomes less linear. For instance, for 1 pass, the memory access pattern is completely linear, while for 2 passes and beyond, the access pattern is random with large portions of linear sections (at least k^2 -sized). Different combinations of chunk sizes for the same number of passes seems to have a negligible variation in performance (showing the

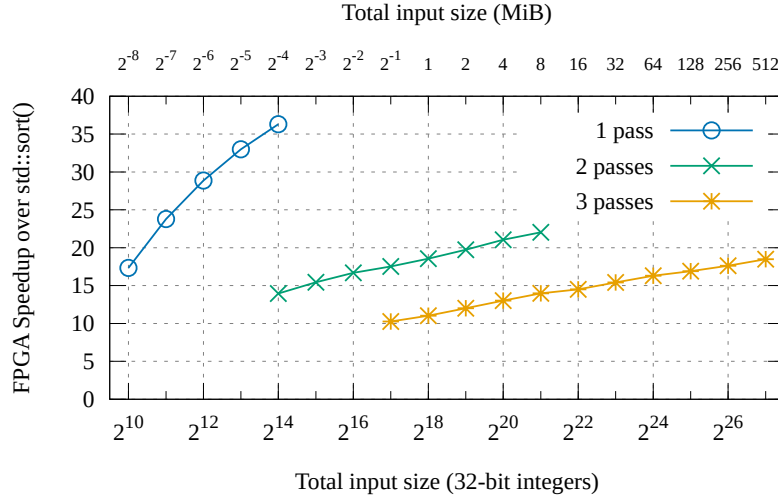


Figure 5.4: Sorting performance

maximum in the figure). This can be explained by the wide-enough chunk size, that achieves to “linearise” the accesses to a considerable degree.

Also evaluated is a sorter configuration optimised for single pass sorting. With k set to 256, it is able to sort 65536 32-bit values in a single pass and with a speedup of 49x over the A53 core and 19x over an Intel i7-8809G core running at 4.2 GHz (in turbo boost). Note that the i7 is on a different system and with a more advanced memory. Table 5.1 illustrates the results of the evaluation, as well as the hardware resource utilisation.

Design configuration ^a				Vivado 2019.2 report				FPGA Speedup over ^c			
k	w	d	r	LUT	FF	BRAM	f_{max} ^b (MHz)	A53		i7	
								Min	Max	Min	Max
128	4(×2)	32-bit	8×	26K	39K	128	250	15	36	6	17
256	4(×2)	32-bit	1×	44K	61K	64	214	6	49	2	19

^aAll designs feature the skewness optimisation (see section 3.2.1) and no payload.

^bNearest lower frequency to f_{max} evenly dividing 1500 MHz.

^cMax. is at input size $k \times k$ (1 pass). Min. only refers to > 1 passes.

k : chunk size of phase 1, w : parallelism degree, d : data width, r : input buffer rate.

Table 5.1: Results summary for full system sorting designs for 32-bit sorting

With respect to the block memory utilisation, which is used for the list buffers, the presented solution can achieve 100% utilisation of the allocated resources. In other words, for phases 1 and 2, all used BRAM can be dedicated to a single fully-sorted list of size $k \times k$, given that k is in multiples of $w \times w$, and the data width is in multiples of the BRAM width of

the target FPGA. For example, for the example configuration with $k = 256$ and 32-bit data, it is able to sort $k^2 = 65536$ integers ($= 256\text{KiB}$) in a single pass with 64 BRAM blocks ($= 64 * 36\text{Kib} = 288\text{KiB}$). In this example, $288 - 256 = 32$ KiB are unused (88.8% utilisation efficiency), only due to the fact that the data width is not 36-bit (or more generally, multiples of 18-bit for mapping directly to RAMB36/RAMB18 primitives).

On the other hand, the input buffer rate optimisation divides the BRAM utilisation efficiency by r for improving the performance of phase 3, if only on-chip sorting is concerned (phases 1 and 2). This is because the $(r - 1)/r$ of the total buffer space is only used for achieving wider bursts on phase 3, and is not used in phases 1 and 2. In conclusion, the implementation efficiency of specific use cases can be improved by fine-tuning the configuration parameters, though if only single-rate many-leaf merging is required, there are superior approaches with respect to the resource utilisation in general (see sections 2.3.4 and 5.1.4 on many-leaf merging).

5.1.4 Properties and related work

As a small-scale sorter

Small-scale sorting, also known as internal sorting, refers to sorting on-chip data (see section 2.3.1). The proposed approach provides a good combination of logic, storage and time requirements (challenges **C3** and **C4**). Table 5.2, compares the complexities of a series of small-scale sorters. The proposed sorter combines the minimum for storage and time, that are in $O(n)$ and $O(n/w)$, where n is the input size and w the throughput. The only exception in time is that of sorting networks, which are only practical for small n .

With respect to the logic, there are two competing streaming sorting networks [167]. The first, although with relatively similar complexity, the impact of the input size is rather limiting for its storage requirements, which are in $O(n \log^2 n)$. The second one has the least logic, but as its time is not linear, its overall throughput as a stream processor is reduced considerably.

The proposed approach requires a logic complexity of $O(w \log^2 w + \sqrt{n})$, which is the sum of

Approach	Throughput	Logic	Storage	Time	Storage type	Fully-streaming
Linear sorter [65]	1	$O(n)$	$O(n)$	$O(n)$	registers	Yes
Bitonic and odd-even mergesort [61]	n	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(\log^2 n)$	registers	Yes
Interleaved linear sorter [166]	$1 \leq w \leq n$	$O(wn)$	$O(wn)$	$O(n/w)$	registers	Yes
Streaming sorting networks [167]	$2 \leq w \leq n$	$O(w \log^2 n)$	$O(n \log^2 n)$	$O(n/w)$	BRAM	Yes
Folded streaming sorting networks [167]	$2 \leq w \leq n$	$O(w)$	$O(n)$	$O(n \log^2 n/w)$	BRAM	No
Partially ordered set linear sorter [168]	$1 \leq w \leq n/2$	$O(w^2 + wn)$	$O(n)$	$O(n/w)$	BRAM	No
Proposed design	$1 \leq w \leq \sqrt{n}$	$O(w \log^2 w + \sqrt{n})$	$O(n)$	$O(n/w)$	BRAM/ DRAM	No

Table 5.2: Comparison with various small-scale sorters, inspired by [167], [168]

the complexity for the FLiMS-based merge tree ($CAS_{tree} \in O(w \log^2 w)$ (or $O(w^2 \log w)$ for the “unoptimised” tree of figure 2.5)) and the merge-capable linear sorters ($O(\sqrt{n})$, totalling k sorting cells, for an input length $n = k^2$). The complexity of the merger tree (CAS_{tree}) is calculated as the sum of the number of CAS units (CAS_{FLiMS}) for each merger in the tree, i.e.:

$$\begin{aligned}
CAS_{tree}(w) &= \sum_{i=1}^{\log_2 w} (2^{\log_2(w)-i} \times CAS_{FLiMS}(2^i)) \\
&= w \sum_{i=1}^{\log_2 w} (2^{-i} \times CAS_{FLiMS}(2^i)) \\
&= w \sum_{i=1}^{\log_2 w} (2^{-i} \times ((2^i) + \frac{1}{2}(2^i) \times \log_2(2^i))) \\
&= w \sum_{i=1}^{\log_2 w} (2^0) + \frac{w}{2} \sum_{i=1}^{\log_2 w} (\log_2(2^i)) \\
&= w \log_2 w + \frac{w}{2} \sum_{i=1}^{\log_2 w} i \\
&= w \log_2 w + \frac{w}{4} (\log_2^2 w + \log_2 w) \\
&= \frac{5w}{4} \log_2 w + \frac{w}{4} \log_2^2 w \in O(w \log^2 w)
\end{aligned} \tag{5.1}$$

A sorter is fully-streaming when processes consecutive chunks without the need to wait in-between [167]. The proposed design is fully-streaming for chunks of size $k = \sqrt{n}$ (phase 1). During phase 2, the logic is being reused for merging, unless we double the logic (see section 6.2.4). When processing multiple chunks separately, the overall throughput is getting halved.

For data that fit on-chip (k^2), in contrast to [168], there is virtually zero latency between reading all the unsorted data and writing the result (challenge **C1**).

Finally, this proposal is the only studied small-scale sorter to provide the merge functionality. It can be used recursively to sort an input of arbitrary size. This makes this solution applicable to the following sorter classification as well (challenge **C4**).

As a large-scale sorter

Large-scale sorting, also known as external sorting, refers to sorting of off-chip memories by using external interfaces (see background sections 2.3.4 and 2.3.5). The proposed sorter achieves both many-leaf sorting for minimising data passes, and high-throughput for saturating the available bandwidth (challenge **C1**). In terms of logic complexity, it is inferior to other many-leaf mergers. The number of comparator cells in many-leaf mergers [42], [76] is $\log_2 k$, whereas the proposal requires a comparator per list. However, the related work (2.3.2 to 2.3.4) require and assume pre-sorted sequences, making them more appropriate for merging only. In many-leaf sorting a small-scale sorter is required to be present anyway, and for small-scale sorting, the proposal offers improvements over the related work.

With respect to alternative non-merge-based end-to-end large-scale approaches, there are only a few examples, including samplesort [8]. There are some considerations when using such a non-merge based sort, such as the sampling accuracy and the applicability of the data format, stable sorting etc. It is also not appropriate for data with many duplicates. Nevertheless, samplesort and other solutions requiring sorting in chunks as a first step, can still benefit from this work as an efficient small-scale sorter.

5.2 Accelerating the merge phase of sort-merge join

An efficient, high-throughput and scalable hardware design is presented for accelerating the merge phase of the sort-merge joins [43]. Sort-merge join is one of the fundamental join algo-

rithms and among the most frequently executed operations in relational databases. It has been the focus in various pieces of research on FPGA use, each having their own different shortcomings and usually only focusing on the sort phase. A new parallel sort-merge join architecture is developed that provides data streaming functionality and high throughput. The key idea is the use of a novel design for a co-grouping engine, with which the input data are summarised on-the-fly. In this way, the operation is performed on streams of data, preserving the linear access pattern for a more efficient movement of the data, and it also eliminates the need for a replay buffer/cache. In contrast to related work, this approach does not make assumptions about the value distribution of the input data and applies to any input size and width.

The result is a fully-streaming sort-merge join approach, where the merge phase is on the programmable logic. The design decisions allows merge-joins to be performed on data of arbitrary length, width and key distribution.

The two most widely applied (equi-)join algorithms are sort-merge join and hash-join (in various forms). Neither of these algorithms strictly dominates the other in terms of performance, and this is especially true when considering implementation on different architectures [37]. The usual approach in data management research is to apply hash-join when one table is much larger than the other and the data are unsorted. Sort-merge joins are applied when the table sizes are in the same order of magnitude or when the input tables are sorted. In this case, the dominant performance factor is the merge phase. While there is published research on the hardware-acceleration of the sort phase (see section 2.3), there is no significant progress on the merge phase.

Equijoin is the task to join two tables by combining all rows that share the same value in the join attributes. If keys appear more than once in the two tables, then the join operation would need to create a Cartesian product of the matching rows from the two tables. As an example, the tables A and B , containing a set of $(key, value)$ pairs, $\{(2, A1), (3, A2), (3, A3)\}$ in A and $\{(3, B1), (3, B2)\}$ in B , their join would be $\{(3, A2, B1), (3, A2, B2), (3, A3, B1), (3, A3, B2)\}$. As this is essentially a product for rows with a matching key, supporting arbitrary product lengths is challenging to implement efficiently on static designs on FPGAs.

On the other hand, hash-join implementations require some form of a hash-table [19]. The input data do not need to be sorted, and for this reason they produce a random access pattern. CPUs are an attractive platform for hash-joins, since they are equipped with mechanisms to cope with the additional latency of accessing data in a non-linear pattern, and their main memory system is also designed for lower latency.

The goal is to accelerate equijoin using sort-merge join on FPGAs, to benefit from the algorithm's streaming nature, while overcoming a number of related challenges found in current research.

5.2.1 Solution

This work considers equi-joins (with the equality predicate) for a single-attribute join, which are one of the key operations in relational database systems.

In high-level, the solution takes two sorted input streams, A and B , each containing rows of the form $\langle index, key \rangle$. The resulting output contains rows of the form $\langle i_{start,A}, i_{end,A}, i_{start,B}, i_{end,B}, key \rangle$, where $i_{start,A}$, $i_{end,A}$ and $i_{start,B}$, $i_{end,B}$ represent the index ranges for A and B respectively, where the value of the key is the same. Figure 5.5 presents the inputs and outputs of the merge-phase accelerator in high-level, where w is the degree of parallelism, and the wire widths are multiples of the respective tuple width.

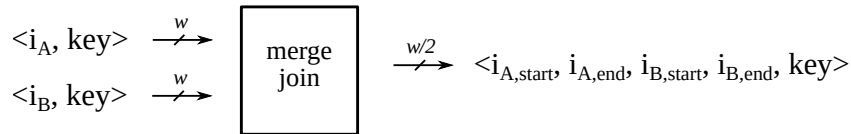


Figure 5.5: High-level view of the merge-phase accelerator

Figure 5.6 presents an overview of the architecture. w denotes the degree of parallelism, or the maximum number of elements that can be processed per cycle. This design receives either 0 or w elements per cycle from each of the 2 inputs, while the number of rows exported in output ranges from 0 to $w/2$ per cycle.

The merge-join accelerator is a modular design mainly consisting of two co-grouping engines

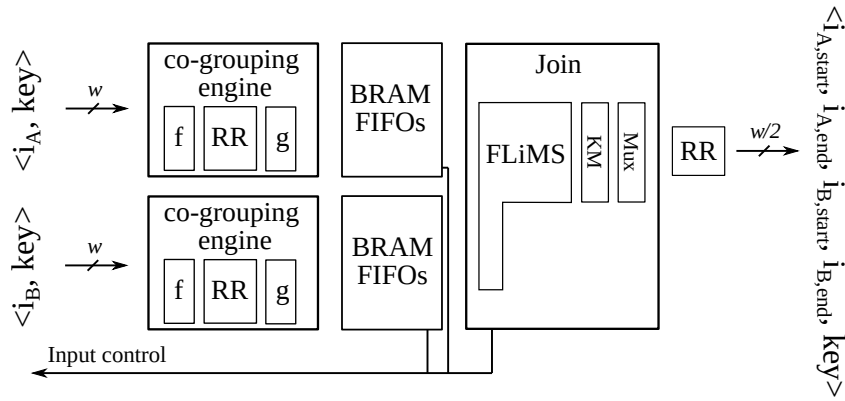


Figure 5.6: Building blocks of the proposed solution

(CG), for compacting multiple tuples with same keys into row ranges, a join module, to match ranges with same keys in the two streams, two circular queue buffers, to synchronize the co-grouping engines with the join module, and finally multiple parallel round-robin arbiters (RR). The latter building block is responsible for removing invalid (*Null*) entries while keeping the data correctly aligned in banked memories for high-throughput processing (see chapter 4). Following is a detailed description of the novel building blocks of the approach.

Co-grouping engine

First, the data pass through the co-grouping engine. There are two co-grouping engines, one for each input stream. They are responsible for summarising multiple consecutive entries sharing the same key (duplicates). The input stream is a series of $\langle index, key \rangle$ pairs and the output is a series of $\langle index_{start}, index_{end}, key \rangle$ tuples.

The idea is to be able to provide a high-throughput by processing multiple $\langle index, key \rangle$ pairs per cycle, without underutilising the datapath width. For these purposes, we assume that the data are already organised in a round-robin fashion, such as when loading consecutive main memory locations to banked memory, or directly to this stream processor, if the platform supports it. In order to achieve this, we use the parallel architecture as illustrated in figure 5.7.

In the first pipeline stage, there are w entities, denoted with f , that invalidate any tuples whose key does not appear for the first time. The resulting tuples are of the form $\langle index, key, valid \rangle$ where the $\langle index, key \rangle$ part is the same as the input (one cycle delayed). The *valid* bit is

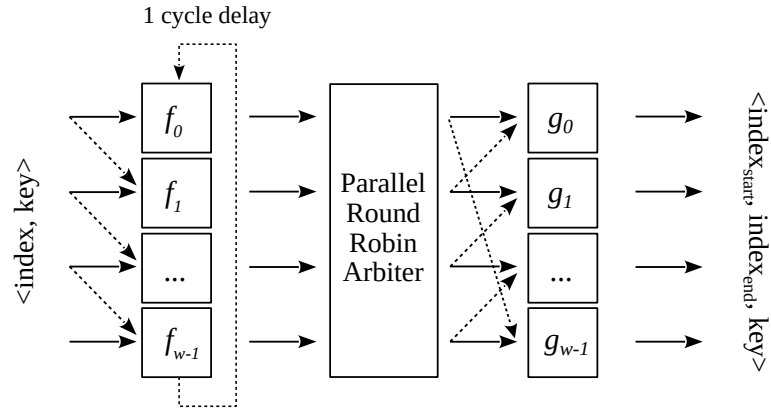


Figure 5.7: Co-grouping engine

the result of the condition ($key \neq key_{Left}$), where key_{Left} is the immediately previous key. In order to provide the correct previous key to the top entity (f_0), the last one (f_{w-1}) also works as a shift register, to delay the last key of the current batch by a cycle, for comparing it with the top of the next batch. The behaviour of f is summarised in distributed algorithm 5. Figure 5.8 (left) provides a high-level illustration to associate the wire names with the ones used in the co-grouping engine of figure 5.7.

```

1 input  $index, key, key_{Left}$ ;
2 output( $\langle index, key \rangle$ );           ▷ function representing valid output on current cycle
3 while forever do
4   receive (positive clock edge);
5   if  $key \neq key_{Left}$  then
6     | output  $\langle index, key \rangle$ ;
7   end
8 end

```

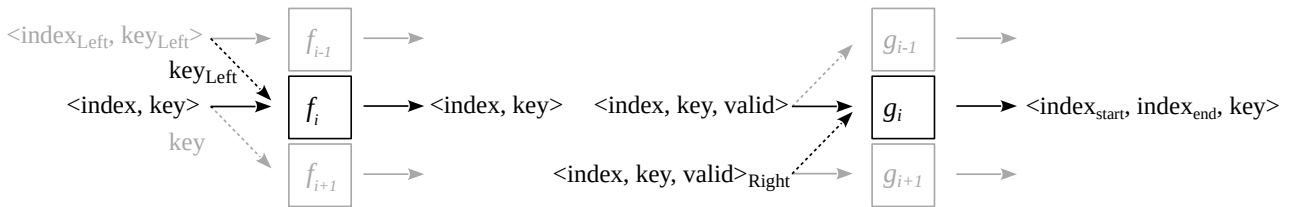
Algorithm 5: Entity f_i algorithm pseudocode

Figure 5.8: Complementary figures for algorithms 5 (left) and 6 (right)

After the first pipeline stage, the parallel round-robin arbiter permutes the $\langle index, key, valid \rangle$ tuples according to their valid bit and the last offset, so that the valid tuples will appear as they are produced in order. For more details on the parallel sound-robin arbiter see chapter 4,

and especially section 4.2.3 that describes its stable version for database use.

During the last pipeline stage, there are w entities responsible for resolving the $index_{end}$ field of the ranges for the duplicates and finalising the output of the co-grouping engine. The output tuples are of the form $\langle index_{start}, index_{end}, key \rangle$. Each one of the g_i entities receives both the tuples in positions i and $(i + 1)\%w$. Note that this does not require delayed input, but there are internal state registers. The actions taken based on input and states are summarised by algorithm 6 and figure 5.8 (right).

```

1 registers  $index'$ ,  $key'$ ,  $valid'$ ;                                ▷ local registers
2 input  $\langle index, key, valid \rangle$ ;
3 input  $\langle index, key, valid \rangle_{Right}$ ;
4 output( $\langle index_{start}, index_{end}, key \rangle$ );    ▷ function representing valid output on current cycle
5 while forever do
6     receive (positive clock edge);
7     if ( $valid \ \&\& \ valid_{Right}$ ) then                                ▷ this and the next key valid
8         if  $key > key_{Right}$  then                                ▷ the following key is smaller → no room for duplicates
9             | output  $\langle index, index_{Right} - 1, key \rangle$ ;
10        else
11            if  $valid'$  then
12                | output  $\langle index', index_{Right} - 1, key' \rangle$ ;    ▷ commit the last saved
13            end
14             $index', key', valid' \leftarrow index, key, True$ ;    ▷ save new values
15        end
16    else if ( $!valid \ \&\& \ valid_{Right}$ )  $\&\& \ valid'$  then    ▷ no more keys equal to register  $key'$ 
17        | output  $\langle index', index_{Right} - 1, key' \rangle$ ;
18        |  $valid' \leftarrow False$ ;
19    else if ( $valid \ \&\& \ !valid_{Right}$ ) then
20        | ▷ key is the lowest available in the batch → may not be the last with this value
21        |  $index', key', valid' \leftarrow index, key, True$ ;    ▷ save new values
22    end
23 end

```

Algorithm 6: Entity g_i algorithm pseudocode

Join module

The purpose of the join module is to get the $\langle index_{start}, index_{end}, key \rangle$ tuples from the two co-groupers, one for each input stream A and B, and encode the Cartesian product whenever the key matches. This results in the output stream, which consists of tuples of the form $\langle i_{start,A}, i_{end,A}, i_{start,B}, i_{end,B}, key \rangle$.

The join module is based on FLiMS [36] of chapter 3, a high-throughput 2-way merger for building hardware merge sorter trees [69]. FLiMS merges 2 sorted streams into one sorted

stream, with a steady output rate of w entries per cycle. The idea of using an efficient merger of 2 streams is based on the fact that, since we essentially eliminate the duplicate keys, we will have a maximum of two identical keys consecutively, after we merge the two input lists together. Then, we can match the grouped keys by just comparing consecutive entries in the sorted-by-key output. Figure 5.9 provides an overview.

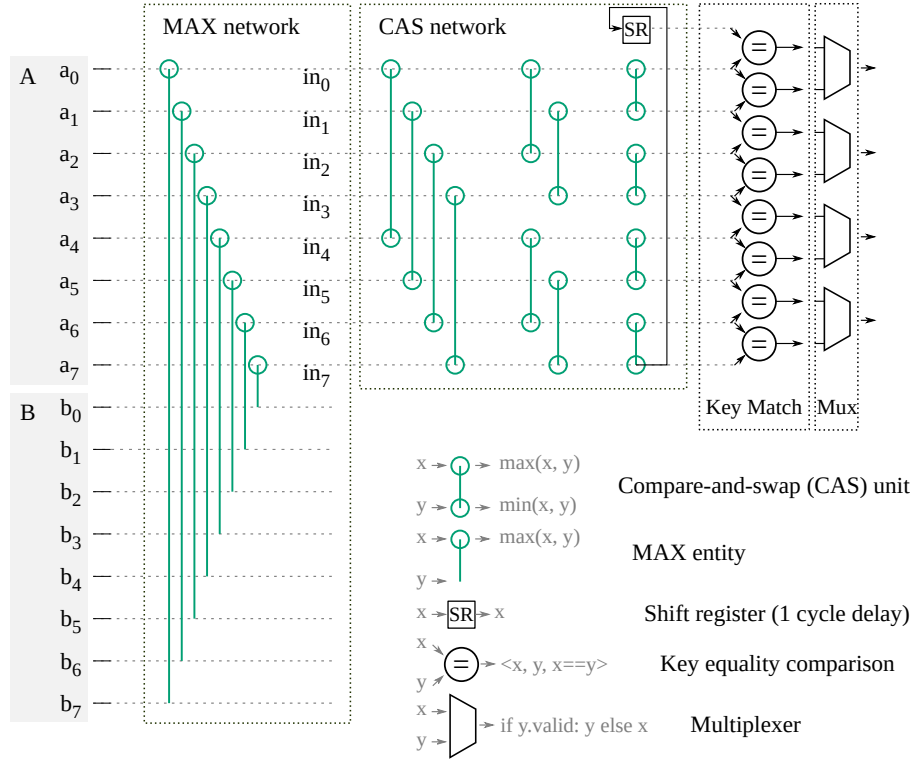


Figure 5.9: Join module, for $w = 8$

Beforehand, the two data streams of $\langle index_{start}, index_{end}, key \rangle$ tuples are stored in two circular queue buffers, one for input A and one for input B. The circular queue buffers are divided into w banks each, to provide a degree of parallelism w . On each cycle, FLiMS dequeues a total of w heads, denoted a_i and b_i from inputs A and B respectively. The MAX entities allow only the top w out of the current $2w$ values residing in the internal registers cA_i and cB_i to pass through. The functionality of these entities is described in algorithm 7. The results of these comparisons are also used for input control, as they are used to calculate the empty space in the circular queue buffers.

We modify FLiMS in the following ways: 1) Every data unit needs to carry the payload, which are the $index_{start}$ and $index_{end}$ fields, as well as an extra bit to remember each tuple's source,

```

1 int  $i$ ;
2 registers  $cA_i, cB_i, in_i$ ;
3 while forever do
4   receive (positive clock edge);
5   if  $cA_i.key > cB_i.key$  then
6      $in_i \leftarrow \langle 0, cA_i \rangle$ ;
7      $cA_i \leftarrow \text{dequeue}(a_i)$ ;
8   else
9      $in_i \leftarrow \langle 1, cB_i \rangle$ ;
10     $cB_i \leftarrow \text{dequeue}(b_{w-i})$ ;
11  end
12 end

```

▷ i is the entity tag

▷ 0 for source A

▷ 1 for source B

Algorithm 7: MAX_i pseudocode, modified for equi-joins

for inputs A and B. 2) The comparators inside the CAS and MAX_i entities now operate on the *key* fields only. Note that the skewness optimisation presented in chapter 3 is not useful here because the duplicates in each input stream are already eliminated by the co-grouping engine.

After FLiMS, there is the key matching stage (KM), where each consecutive pair of tuples are compared based on their keys. The last entry/tuple is also delayed for an additional comparison to the top of the next batch, in the cases where a matching pair is halfway through. When there is a match, the resulting $\langle i_{start,A}, i_{end,A}, i_{start,B}, i_{end,B}, key \rangle$ tuple is produced, and the position of the i fields are arranged according to their source bit.

Next, there are $w/2$ multiplexers (Mux) to reduce the output width to the maximum output rate ($w/2$), based on the valid bits. This is because there should not be three consecutive identical keys, and therefore no two consecutive matches. Finally, the join module output goes through another round-robin module (parallel round-robin arbiter) for $w/2$ inputs (see figure 5.6) to remove the newly occurring sparsity and the results are saved in a banked output buffer.

5.2.2 Evaluation

The evaluation was done on Avnet's Ultra96 development board (see section 5.1.3 for more information on the evaluation platform). It is shown that there is up to 4.1 times speedup over the host processor, even without accelerating the sort phase, while still taking into account the data offloading from and to main memory in Linux.

The current evaluation focuses on comparing single-thread (software) and one-port (hardware) performance. Note that for the sake of simplicity and the focus on evaluating different the input distributions, this specific implementation does not support processing during offloading.

A program written in C [169] has two functions for the same task, one is a software implementation and the other is hardware-accelerated. The code for the latter is rather simple. It initiates some DMA copies and busy-waits when necessary. The input of the program is a text file containing two integers per line, which represent keys from the inputs A and B. These data are saved twice, once in a memory-allocated (malloc) space and once in a memory-mapped (mmap) space. The first performs better for the software implementation, since it is cached, for a fair comparison. The latter is a continuous DRAM memory location outside the kernel space, where the Direct Memory Access (DMA) engine moves data from and into the programmable logic. After calling the two functions individually, the expected result is the same, an array of tuples, each carrying the 5 32-bit values. The C program is compiled using *gcc -O3 -march=native*.

The implementation of the merge phase accelerator is for $w = 8$ (256-bit/cycle), and has a clock frequency of 136 MHz. One basic performance factor is the input size. Longer DMA transfers are more efficient in such a heterogeneous system [77]. Here, the FPGA speedup peaks when the DMA throughput reaches 2.1 GB/s at an input size of around 2^{14} 32-bit values, using this operating frequency.

The performance of both software and hardware implementations also depends on the key value distributions. In figure 5.10, there are experimental results on how the FPGA speedup varies across different random inputs. The data are automatically generated to have some specified characteristics. We vary the percentage of distinct keys in both inputs together, as well as the number of key matches. The input size is fixed at 2^{14} 32-bit keys per table, to approach saturation of the available throughput. In order to minimize any runtime noise, measures that were taken include pinning processes, running in isolation with high priority and averaging tens of runs per data point. As an observation, FPGA favours inputs that yield more ‘internal’ workload, i.e. many co-groupings of duplicate keys, with fewer rows in output for less inter-chip communication.

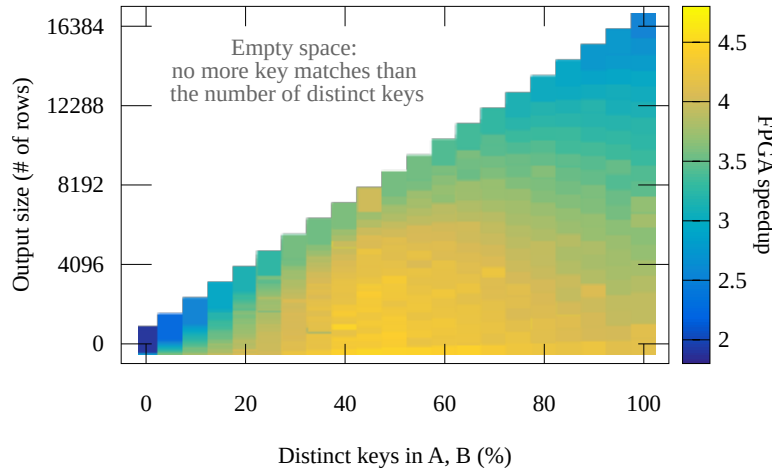


Figure 5.10: FPGA speedup for different key distributions

The FPGA speedup for this input design space varied from 1.9 to 4.8, with 3.4, 3.8 and 4.1 being the lower quartile, the median and the upper quartile respectively. This peripheral consumes about 19K LUTs and 22K FFs, which take 27% and 15% of the respective capacity for this device, also including the AXI logic and input and output buffers.

5.2.3 Properties and related work

Regarding the inputs and output, the approach to use indices instead of passing full rows of tables minimises the inter-chip data movement. Additionally, the co-grouping approach is consistent with a strategy known as “late materialization” that is implemented in most modern in-memory database systems [170]. This reduces the bandwidth needs (challenge **C1**), rather than carrying entire rows through the accelerator. Not carrying entire row data also makes this applicable to arbitrarily wide tables. According to the application’s needs the expansion could be done on demand. Additionally, this fully-streaming functionality is agnostic to the input length (challenge **C3**).

With respect to edge cases, the only unaddressed case is when the number of records with the same key exceeds the range of the index counter (such as with 2^{32}). For instance, the width of the index-related counters and wires need to be increased to the desired count, to support such a case, according to the workload characteristics. Note that due to the late materialisation

approach, the expansion of the results does not take place on the FPGA design at the moment, hence eliminating edge cases relating to the use of buffers in related work [23].

With respect to performance, a consideration about the input could be the data skewness problem, which is traditionally found in hardware merge sorters, as introduced in chapter 3. It arises when the input queues are not long enough and the merger may underutilise the bandwidth of one or more of the inputs because of the key distribution. In this design, this is already solved efficiently by the co-grouping engine, and the queues afterwards hold co-grouped tuples sharing the same key in single entries. Thus, the skewness optimisation of FLiMS (section 3.2.1) is not required for this use of FLiMS in the join module.

The total pipeline length is equal to:

$$\begin{aligned}
& d_{CG}(w) + d_{Join}(w) + d_{RR}(w/2) + 1 \\
&= (d_f(w) + d_{PRRA}(w) + d_g(w)) + (d_{FLiMS}(w) + d_{KM}(w) + d_{Mux}(w)) + d_{PRRA}(w/2) + 1 \\
&= (1 + (\log_2(w) \cdot (\log_2(w) + 1)/2 + \log_2(w)) + 1) + (\log_2(w) + 1 + 1) + (\log_2(w/2) \cdot (\log_2(w/2) + 1)/2 + \log_2(w/2)) + 1 \\
&= \log_2^2(w) + 2 \cdot \log_2(w) + 4 \in O(\log^2(w))
\end{aligned}$$

where $d_M(w)$ is the latency of the module named M as a function of w , the degree of parallelism, a power of 2. This is if we ignore the circular queue buffer between each co-grouping engine and the join module. Hence, the +1 is used here to represent the minimum time a tuple stays in the queues. The tuples can stay for longer in the pipeline, according to the values in both inputs.

With scaling the degree of parallelism w , the building blocks remain ‘feedback-less’, as with FLiMS of chapter 3. Further optimisation of the implementation, such as to achieve a better maximal operating frequency, is part of the future work. This includes applying the throughput optimisation and the processing-during-offloading functionality of section *propere*.

In terms of hardware resources, it requires 2 sets of w queue buffers of data width $3 \cdot 32$ -bit (for the fields in $\langle index_{start}, index_{end}, key \rangle$) and of depth greater than $1 + (\log_2(w) \cdot (\log_2(w) + 1)/2 + \log_2(w)) + 1 \in O(\log^2(w))$, the latency of the co-grouping engine. Regarding the LUT/FF/routing complexity, the fastest growing resources are in the CAS network of the

parallel round-robin arbiters (PRRA). This is in $O(w \cdot \log^2(w))$ and is an achievement over related work on sort-merge joins [23]. Section 6.2.2 discusses a possible optimisation of the PRRA that could reduce this complexity to only $O(w \cdot \log(w))$.

The hardware complexity with respect to the key-size is linear. However, if the key size is hundreds of bits wide, the CAS units might not be able to fit under a narrow FPGA cycle. In such cases, there are solutions for scaling arithmetic operations on wider fields [171], or breaking the comparison in multiple pipeline stages [42], which could be evaluated in this context.

With respect to related work, there are relatively few hardware-based sort-merge join accelerators, as described in the background section about joins on FPGAs 2.4.1. There is a notable advantage in terms of applicability and scalability (challenges **C3** and **C4**), as the competitor sort-merge join accelerator required N^2 comparators for matching keys from N rows in each input at a time [23]. The proposed approach only needs $O(N \cdot \log^2(N))$ comparators, also while operating on co-grouped entries. Additionally, it removes the need for a “replay buffer” which was a bottleneck in inputs with many matches [23], due to the irregular access patterns.

As the proposed solution is the first to use late materialisation, it cannot be directly compared to other approaches in terms of performance, unless it is compared in a realistic DBMS environment (see section 6.2.3). Also, there are fundamental architectural differences in the related work, such as with multi-FPGA servers [23] and experimental platforms. While a similar discussion can be made for hash-join based approaches as well, it is actually useful to have many algorithmic options available. This is because different algorithms may be more appropriate in different situations and input attributes, such as according to the ratio between the size of the input tables. For sort-merge joins on FPGAs, the overall performance can still come down to the sorter implementation, as well as the main memory throughput bottleneck, and this approach achieves high-throughput joining as an efficient sorter add-on or for pre-sorted data.

5.3 Accelerating distinct count

In contrast to most related work on distinct counts on FPGAs (section 2.4.4), this chapter focuses on non-approximate methods. There are applications that having a zero margin of error is desired, such as when specified by the requirements of the project, or to simplify the contracting process. The solutions presented here are, thus, immune to statistical errors, as there are no design choices concerning accuracy losses and their relationship with different input data distributions.

We focus on 2 specialised use-cases for FPGAs: distinct count of keys with a restricted range (section 5.3.1), and distinct count of keys with a wider range (section 5.3.2).

5.3.1 Distinct count of keys with a restricted range

When the range of a key is known and sufficiently restricted, it is possible to perform distinct count in $O(N)$ time, where N is the input size, by keeping a bitmap, with one bit per possible value. A “test-and-set” algorithm could check if a key was observed for the first time by setting the appropriate flag and incrementing a counter on new occurrences.

In software, the considerations would relate to the main memory, as well as all architectural and memory hierarchy implications. First, the size of the bitmap must be able to fit inside the main memory. As bit manipulation is not that efficient on modern CPUs, the indexing is found to be more efficient on bytes, which increases the storage by 8 (8 bits per flag). Another consideration is the latency that comes with the random access pattern. If the bitmap is greater than the cache size, there will be a significant miss rate, which will result in excessive DRAM accesses and a performance overhead. The source code of a relatively straightforward implementation in C is shown in algorithm 8.

In FPGAs however, addressing by bit can be done very efficiently. By using block RAM, which is conventionally counted in millions of bits (Mbits), a single-cycle latency can be achieved for a key range fitting on-chip. For the same key range, on CPUs it would require 8 times the

```

1  char* flags = (char*) malloc(Range*sizeof(char));
2
3  int distinct_count_cpu(unsigned int* array, int array_size){
4
5      int i, counter=0;
6      for (i=0; i<array_size; i++){
7          int index=array[i];
8          if (flags[index]==0)
9              counter++;
10             flags[index]=1;
11         }
12     return counter;
13 }

```

Algorithm 8: An implementation of distinct count in C

Last-Level-Cache size for a performance in the same order of magnitude, considering that the CPU has tens of times higher clock speed but tens of cycles latency for accessing cache contents. As an example of a rough analogy, the ZU3EG UltraScale+ device, a relatively small FPGA can hold around 8 Million bits, which could compete with CPUs with 8 MB of cache.

One challenge for an FPGA implementation would be to be able to introduce parallelism to this simplistic approach of using the BRAM as a bit array to answer memberships. Without any parallelism, it would be able to process 32-bits per cycle, regardless of the available port width. Another challenge would be to parallelise without introducing any redundancy in BRAM space, as BRAM natively allows 1 write and 2 reads per cycle (per block, and on popular architectures such as Xilinx UltraScale+).

Figure 5.11 presents a high-level overview of a proposed solution as a pipeline, for an input consisting of four 32-bit keys per cycle, totalling to 128-bit in the datapath as an example. First, each of the four keys are sent to a group of four FIFO queues, with a condition that the last two bits of the key should be equal to the ID of the FIFO in the group. Then, each round-robin arbiter dequeues the next available key from four FIFOs consisting of keys of the same last-two-bit combination using a round-robin priority. The selected four keys from the round-robin arbiters now have different last two bits, one combination for each of the four BRAM partitions. A “test-and-set” arrangement tests for the first occurrence of each key, and a counter is incremented on each bit flip. Finally, an adder provides the distinct count of all partitions, by adding all four immediate distinct counts.

In this way, when the data is close to evenly distributed with regards to the last two bits of the

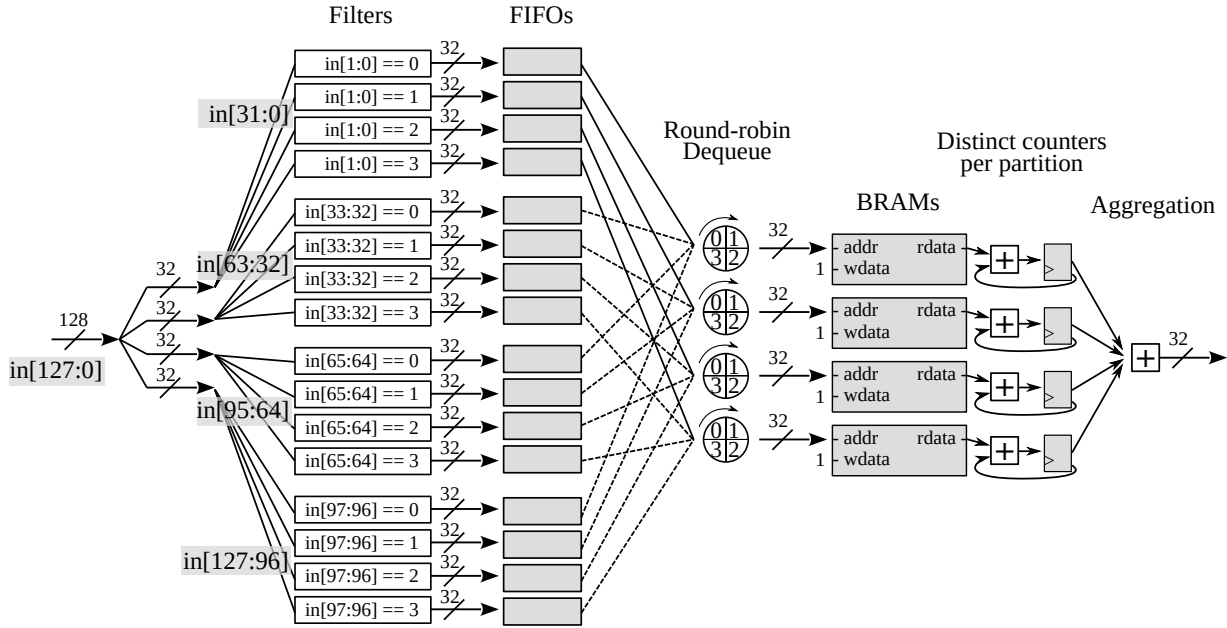


Figure 5.11: High-throughput design of distinct count on FPGAs, $w = 4$

keys, this design is able to process 4 keys per cycle, using a BRAM space of *exactly as many bits as the range of the keys*. Not exchanging BRAM space for *throughput* can be crucial to the applicability to the task, as the BRAM of the FPGA can be seen as a precious resource in such specialised uses with a restricted key range.

We evaluate this design on Ultra96, working as an AXI peripheral controlled by a C callable function on a dedicated address space, with a direct memory access (DMA) engine facilitating the transfers to the design. A stall signal is generated so that it goes high when the FIFOs cannot fit an additional AXI burst, as AXI bursts cannot be terminated early [172]. This design has an operating frequency of 125 MHz and around 2 GiB/s theoretical peak bandwidth of the 128-bit port. We compare the results to the single-core implementation in C, as shown in algorithm 8. The FPGA results are compared to a system with an Intel i7-8809G, a CPU with a high single-thread performance, running at 4.2 GHz and with a more sophisticated memory hierarchy. With respect to the input, we vary the number of inputs, as well as the range of the keys.

Figure 5.12 illustrates the results of the design exploration of the input as a way to compare the performance of the FPGA design and the software alternative. The first two colormaps

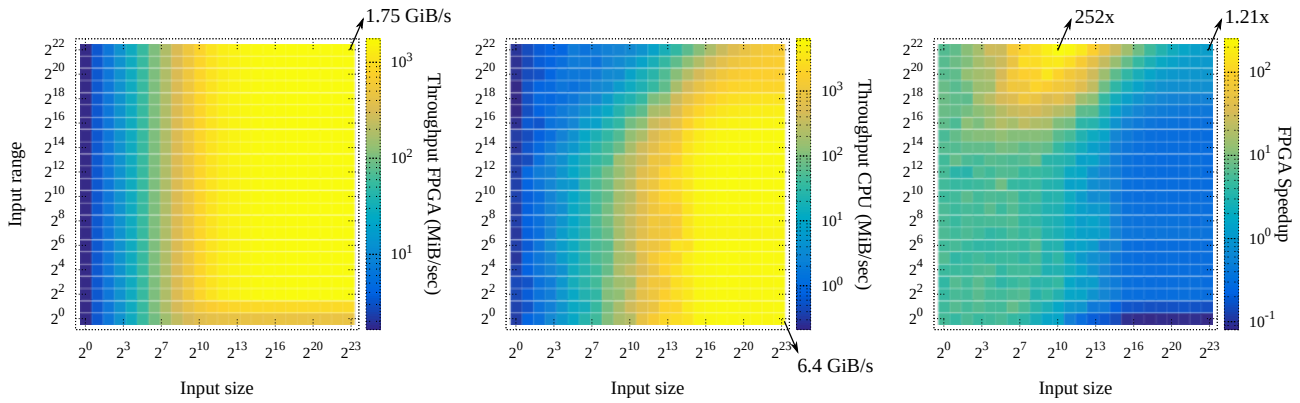


Figure 5.12: High-throughput distinct counter performance for different inputs

show how the achieved throughput varies according to the input size and key range. As an observation, the FPGA throughput is highly influenced by the input size, and this is because the DMA engine is most efficient on longer transfers. At the same time it is also not dependent on the input range, as the entire BRAM is accessible in under one cycle latency, with the exception of key ranges 1 and 2, which only utilise a quarter and a half of the partitions respectively. These exceptions also represent the worst case performance, when the last two bits are not evenly distributed. On the other hand, the CPU is similarly affected by the input size, but longer input probably also improves the performance of its adaptive mechanisms such as prefetching etc. The highest achievable throughput in the FPGA design is 1.75 GiB/s, while on the CPU it is 6.4 GiB/s. Notably, unlike the FPGA, the cache hierarchy limitations are reflected (roughly) on the throughput values through the key ranges. As a flag is represented by a byte, 32 KiB for L1D cache translates to a degradation near past 2^{15} , 128 KiB for L2 cache near 2^{17} , and 8 MiB for Last-Level cache (LLC) near 2^{23} , if we extrapolate the y axis. Finally, the third colormap is a combination of the first two colormaps to show the speedup of the FPGA. For longer inputs, the performance of the FPGA is close to the one of the CPU, but is much higher on medium input sizes, such as 252x for 1024 elements, and for lower power.

A higher-end FPGA is seemingly going to show further benefit by having more BRAM than the equivalent space available in the last-level cache of the CPU. One consideration for this case would be the scalability to a wider key range with more BRAM. While addressing by bit was feasible in the evaluation on this FPGA and key range, the additional addressing logic for a wider range could lengthen the critical path, resulting in a timing overhead. However, in

terms of scalability to the problem size, the proposed design supports arbitrary input sizes, and the restricted range aspect still applies to big data. The main disadvantage of the proposed approach would be the need to reprogram the FPGA per query, in order to initialise all flags in BRAM to 0, which takes a few hundred milliseconds.

Note that both the CPU baseline and the proposed approach do not need a flag counting step at the end, as the counting is done progressively. Otherwise, following a final flag counting step would be disproportionately expensive for smaller problem sizes for a fixed key range (yielding a time complexity of $O(N + M)$, where M is the key range). On CPUs, there are bit manipulation intrinsics that could be used to accelerate this final phase [109], such as with Hamming weight (popcount) instructions. Future work could elaborate on fine-tuning a bit-manipulation-based approach on CPUs, as well as with multi-threaded implementations, for improving the software baseline and cache efficiency.

As a conclusion, fully exploiting the BRAM on the FPGA can make it very competitive for this specialised big data task, with some potential shortcomings such as the reconfiguration time. This approach directly addresses the inter-chip communication challenge (**C1**), although it benefits from a reduced applicability (**C3**) through this specialised use case.

5.3.2 Distinct count of keys with a wider range

A solution for performing distinct count on input with a wider range is to first sort the data and then process the sorted data. When using sorting, the main complexity would be that of sorting, and this also applies to an equivalent CPU implementation. Essentially, the main focus for performance comparison would shift back to sorting, which is used as a means to linearise the access pattern for both the FPGA and CPU implementation, as the post-processing (distinct count) is found to be done efficiently enough. The post-processing step on the FPGA can be "hidden" by including it in the pipeline.

This specific task would be a rather trivial sorter add-on to implement and evaluate, and comparing it with differently optimised software is outside of the scope of the thesis. Thus, it is

omitted in favour of a variation that is closer to real-world applications and is more demanding computationally. The following section presents a variation of distinct count, also with keys of a wider range. It also demonstrates how task-pipelining on the FPGA can eliminate the need to wait between sort and distinct count, and achieves to turn the spotlight back to sorting.

5.4 Accelerating distinct count with groups

An emerging expensive operation is to find the distinct count of keys per group, as in the industrial setting of section 1.1. The input is a list of $\langle group_id, key \rangle$ pairs, with each field being 32-bit wide, totalling 64-bits per pair. The resulting data are 64-bit-wide pairs as well, with the last 32-bit value representing the distinct count of the respective group. For instance, the distinct count of the list $(A1, A2, B2, B2, C3, G4, G5, G7)$ is $(A2, B1, C1, G3)$.

5.4.1 Design

The proposed hardware accelerator, illustrated in figure 5.13, uses parallel round-robin arbiters (chapter 4), plus some adopted building blocks from the co-grouping engine [43] (section 5.2.1) to perform this task as a non-blocking high-throughput stream processor. The output of the merge sorter (section 5.1) bypasses the distinct counter, until the last pass takes place, when the sorted output gets its final form to be processed on-the-fly.

The functionality of this pipeline goes as follows. First, the f'_i modules invalidate duplicate occurrences of the same $\langle group_id, key \rangle$ combination. This is achieved by comparing each entry with its immediate predecessor, as the stream is already sorted. Similarly, the f''_i modules only allow unique $group_ids$ to pass, but with an index appended, representing their order just before removing duplicate $group_ids$. Distributed algorithms 9 and 10 summarise the functionality of modules f'_i and f''_i respectively.

Finally, the g'_i modules subtract the indices of consecutive $\langle group_id, index \rangle$ entries to provide the final distinct count per $group_id$. The algorithm of g'_i entities is very similar with

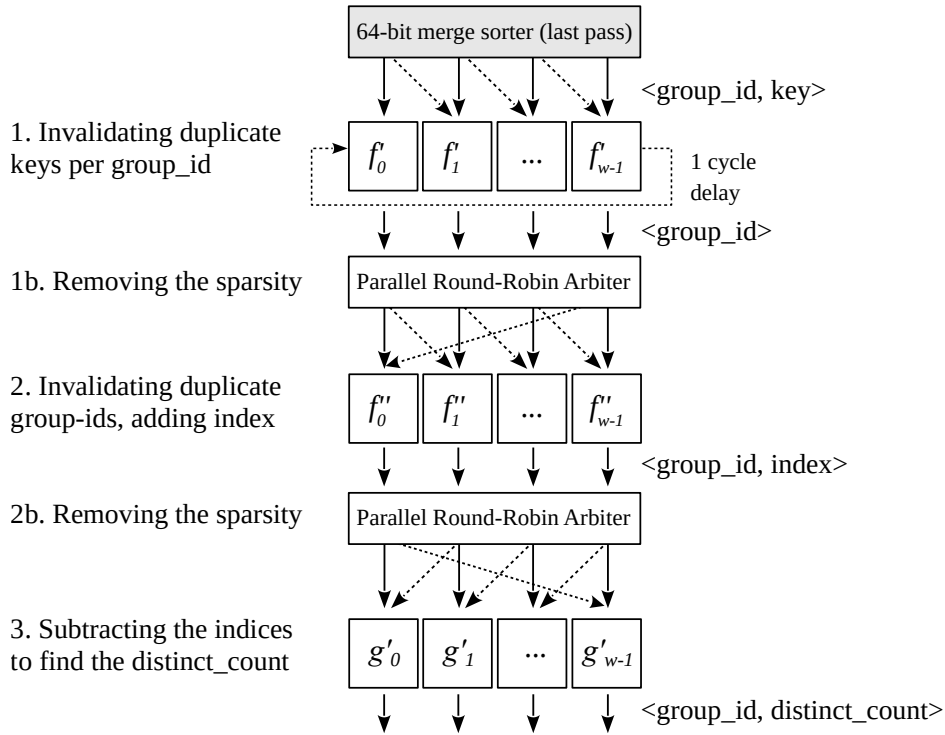


Figure 5.13: Distinct counter engine

```

1 input group_id, key, group_idLeft, keyLeft;
2 output(< group_id >);           ▷ function representing valid output on current cycle
3 while forever do
4   receive (positive clock edge);
5   if < group_id, key > ≠ < group_idLeft, keyLeft > then
6     | output < group_id >;
7   end
8 end

```

Algorithm 9: Entity f'_i algorithm pseudocode

```

1 int i;                               ▷ i is the entity tag
2 register index=i;
3 input group_id, group_idLeft;
4 output(< group_id, index >);       ▷ function representing valid output on current cycle
5 while forever do
6   receive (positive clock edge);
7   if < group_id > ≠ < group_idLeft > then
8     | output < group_id, index >;
9     | index += w;
10  end
11 end

```

Algorithm 10: Entity f''_i algorithm pseudocode

the algorithm 6 of g_i (now with *keys* referring to *group_ids*). The only difference is in the output function, where now instead of providing both the $index_{start}$, $index_{end}$, the $index_{start}$

field is now subtracted from the $index_{end}$ field, for providing the distinct count per $group_{id}$. Any occurring sparsity is removed using parallel round-robin arbiters. The parallel round-robin arbiters (chapter 4) rearrange arriving entries to achieve a round-robin arbitration effect to the output, but with full throughput.

5.4.2 Evaluation

The evaluation methodology is the same as with the proposed hardware sorter (section 5.1). The design is an add-on of the provided merge sorter, enabled on the last pass of merging, for 64-bit entries.

In order to estimate the performance under different inputs, we average multiple runs of two opposite cases. One with 100% unique groups, that consumes the most time, as the number of lines in output is the same as sorting. The second type of input is with near 0% unique groups (all duplicates), which outputs a couple of lines in the last pass, maximising the FPGA workload ratio to data movement. The performance variation between the two types is relatively small, as sorting still takes the majority of time. This is especially true for inputs that require more than one pass, because the data movement savings actually happen only during the last pass, and also the CPU performance improves for many duplicate values as well.

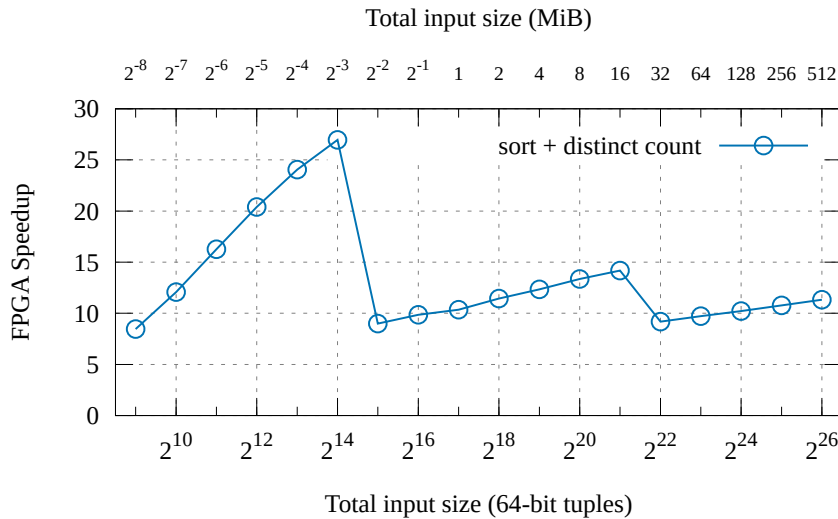


Figure 5.14: Distinct count performance

Figure 5.14 and table 5.3 summarise the results of this exploration. The steps are from the

sorter requiring multiple passes, such as after 2^{14} (as in figure 5.4 for a stand-alone sorter). The average speedup of the selected configuration over `std::sort()` plus distinct count on the A53 core reaches up to 27 times, for 16384 tuples, and over 9 times for the other input lengths.

Sorter configuration ^a				Vivado 2019.2 report				FPGA Speedup over ^c			
k	w	d	r	LUT	FF	BRAM	$f_{max}'^b$ (MHz)	A53		i7	
								Min	Max	Min	Max
128	2($\times$ 2)	64-bit	4 $\times$	35K	50K	120	250	9	27	3	12
(with distinct counter)											

^aFeaturing the skewness optimisation (see section 3.2.1) and no payload.

^bNearest lower frequency to f_{max} evenly dividing 1500 MHz.

^cMax. is at input size $k \times k$ (1 pass). Min. only refers to > 1 passes.

k : chunk size of phase 1, w : parallelism degree, d : data width, r : input buffer rate.

Table 5.3: Results summary for full system 64-bit sorting with distinct counter

Not shown in the figure are the performance variations between the different input types. Most of the variation happens at 16384 (single pass), where the 27 times speedup is actually the average of around 24 and 30 times for the inputs with 100% unique groups and near 0% unique groups respectively. For a lengthier input, this sorter performs more than one data pass, and the variation of input becomes less noticeable. This is because the transfer-time savings from removing duplicates in the output only happen during the last pass.

5.4.3 Comparison to related work

While there are many works focusing on approximate solutions, there are only few concerning exact results, including in a database context (see section 2.4.4).

This section presented an alternative approach to building custom cache hierarchies using content addressable tables (CATs), as in [29]. The idea behind CAT-based approaches is to perform “GROUP BY”s and/or answer set membership queries using hashes, that induce a random memory access pattern. The reduced memory performance is then dealt with multithreading on the FPGA, to be able to saturate the bandwidth. Such approaches are more prone to the data distributions, in the same way a working set might be larger than CPU caches [14]. The proposed design is analogous to sort-merge joins as opposed to hash joins [27], as the most

expensive part of the computation here is dealt by the sorter accelerator.

Advantages include more deterministic performance for different distributions (but relying more on the sorter performance), simpler designs (no need for caches), scanning access patterns, effortless high-throughput (no need for multi-threading), and modularity. Thus, the main challenges that are addressed are the inter-chip communication (**C1**) and development (**C2**), as well as applicability (**C3**). This approach is especially useful when a sorter or sorted data are already present, such as through using sort-merge joins, and the overhead of including such a sorter add-on would be minimal. A disadvantage would be the requirement to sort first. Special use cases such as in section 5.3, can still yield a working set small enough that all the required data fit on-chip, that would perform better without the need to sort for linearising the access pattern.

5.5 Summary

This chapter introduces novel designs for full-system hardware sorting, equi-joins (using sort-merge join), distinct count and distinct count with groups.

It presents a novel hardware sorter [32] (section 5.1) adaptable to different input-sizes, bandwidth and use cases. As a small-scale sorter, it has a competitive combination of logic, storage and time requirements, but also works as a merger of multiple input lists. The merging capability was embedded to the linear sorter, to enable scalability to arbitrary problem sizes. The approach to use a parallel merge tree to combine single-rate mergers provides a way to saturate the bandwidth while still performing many-leaf merge. An open-source script [163] generates Verilog code for a sorter of a custom degree of parallelism, merge capacity, data and payload width. The performance on the evaluation platform was up to a 49x speedup over `std::sort()` and up to 19x over `std::sort()` when running on a different system with an i7 processor.

A merge join approach is introduced [43] (section 5.2), applicable to data of arbitrary length, depth and distribution. The open-source implementation [169] yields a median speedup of

3.8x times on the evaluation platform. Though, it is here evaluated out-of-context, with more potential for larger devices, where the sorter can also reside for achieving direct task-pipelining.

The range-restricted distinct counter (section 5.3.1) introduces a BRAM organisation inspired by network switches to achieve high-throughput, by being able to process all data in streaming fashion. This achieved a speedup of up to 252x over the Intel i7-8809G in a system with more advanced memory, though the performance advantage of this approach depends heavily on the input length and range size.

Finally, a novel design is presented for implementing the “distinct count with groups” task (section 5.4) for arbitrarily long data, that utilises the “parallel round-robin arbiter” for achieving non-blocking and high-throughput computation. This accelerator is designed to extend the merge sorter pipeline, and is activated on the last pass of the data, when the data will be sorted in their entirety. This approach achieved 9x to 27x speedup on the evaluation platform, and 3x to 12x over the i7 for 64-bit input, mostly depending on the sorter implementation.

The presented techniques for processing data in streaming fashion with high-throughput, as well as the approach to implement sorter add-ons, can be used more widely to linearise the access pattern of other data aggregation problems and achieve high-performance in heterogeneous systems. It is demonstrated with two use-cases how the “parallel round-robin arbiter” (see chapter 4) is useful for such join or aggregation tasks as found in relational databases.

Chapter 6

Conclusion

This PhD introduces novel parallel algorithms and FPGA designs for big data analytics acceleration that take into consideration common platform limitations. The presented designs accelerate time-consuming database operators such as sorting, sort-merge join and distinct count, with notable improvements over the state-of-the-art where applicable, including on resource and time complexities.

These accelerators make use of two novel fundamental building blocks, the FLiMS lightweight merge block and the parallel round-robin arbiter. They are shown to have a wide applicability to different problems, including in databases, network switches and single-instruction multiple-data (SIMD) algorithms for CPUs. The preferred methodology on acceleration was to use algorithms that stream sorted data through a wide datapath, for being scalable to input of arbitrary size and future high-bandwidth datapaths. The presented techniques for building high-throughput stream processors can be used to build novel scalable designs for different aggregation tasks.

Finally, embedding small FPGAs as reconfigurable SIMD instructions in CPUs is predicted to be a viable future path for big data acceleration, by solving the inter-chip communication challenges. Communication is one of the most important bottlenecks for accelerating big data analytics using FPGAs, that also expands into development challenges, high deployment cost and scalability, portability and applicability limitations. Simodense, a novel RISC-V softcore is

presented in appendix C, that is optimised for exploring custom SIMD instructions. The goal of this softcore was to help improve the research for novel SIMD instructions, though section 6.2.5 elaborates on the future work required to support reconfigurable instructions in high-end CPUs.

6.1 Summary of contributions

This section briefly lists the contributions of the thesis, alongside their respective addressed research challenges (as introduced in section 1.2): **C1** communication, **C2** development, **C3** portability, applicability and **C4** scalability.

- FLiMS, the high-throughput 2-way merger that is the most hardware-efficient building block to date for building merge sorters using merge trees. (**C3**, **C4**)
- Variations of FLiMS for optimising it for skewed datasets or supporting stable sort, to widen its applicability for different big data applications. (**C3**)
- Correctness proofs for non-trivial aspects of the novel designs, and theoretical and experimental comparison with the state-of-the-art.
- The SIMD application of FLiMS for accelerating merge sort in modern CPUs. (**C3**)
- The linear sorter modification to become merge-capable that allows using it both as a small scale sorter, as well as a many-input merge sorter. (**C3**)
- The approach to combine single-rate many-leaf sorters, with a smaller parallel merge-tree, to overcome the many-leaf vs high-throughput trade-off found in related work. (**C1**, **C4**)
- An end-to-end sorter design with better combination of asymptotic complexities than the related work. (**C1**, **C3**)
- The parallel round-robin arbiter, which provides a load balancing scheme assisting high-throughput computation, without relying on multi-tasking or data partitioning. (**C1**, **C2**, **C4**)

- The combined variation of parallel round-robin arbiter, which yielded 1.7 Tbps aggregate throughput for a 16-port FPGA-based network switch, also with better algorithmic performance than traditional scheduling algorithms. (**C1**)
- Novel designs for accelerating the merge phase of sort-merge join using late materialisation, making it efficient for inputs of arbitrary length, data distributions and format. (**C3, C4**)
- Non-approximate methods for computing distinct count under different problem specifications on FPGAs and comparison with CPU performance. (**C4**)
- Insights on supporting reconfigurable SIMD instructions in future generations of general purpose processors (CPUs) through the discussions of section 6.2.5 and appendix C. (**C1, C2**)

6.2 Future work

There are many ways the discussed topics can be expanded. The following ideas and preliminary experiments also have as a goal to further address the aforementioned research challenges **C1-C4**.

6.2.1 Further optimisation

One interesting way the current material could be expanded is through additional optimisation, mainly targeting challenges **C3** and **C4**. Finding ways to reduce the logic requirements while keeping high timing performance would be ideal to increase the applicability and scalability of such designs to a broader range of FPGAs and ASICs.

In some cases it may be easier to see that there is a form of redundancy in the logic or data. Such case is with the combined parallel-round robin arbiter (CPRRA), where the same batch of packets is broadcasted to multiple sets of barrel shifters, for rotating them in different amounts.

The barrel shifters are currently the biggest contributor to the hardware complexity of CPRRA, and would be desirable to reduce.

Optimisation can be challenging, such as by having an exponentially growing design space for exploration, where brute force is not practical. Thus, other exploration techniques could be followed, including evolutionary algorithms. It would also be interesting if a commercialisation of quantum computing would lead to a revolution in logic optimisation and place-and-route tools for FPGA development, or replacing FPGAs entirely in some domains.

6.2.2 Optimising the parallel round-robin arbiter

One example for further logic optimisation is about the parallel-round robin arbiter (chapter 4), mainly concerning challenge **C3**. This section provides an alternative implementation for the (non-combined) parallel round-robin arbiter that has less routing complexity, but its control logic requirements seem too high for a practical use on FPGAs (at least when scaling to higher numbers of ports). If the control logic complexity is ignored, parallel round-robins arbiters can be implemented with much fewer 2x2 switches, by using an inverse butterfly network (butterfly networks are also referred to as banyan or omega networks in different contexts [173], and the inverse prefix is sometimes called reverse).

Butterfly networks have $\log_2(N)$ parallel stages, where N is the number of inputs. They can perform (an interesting) subset of all possible permutations of the input. Therefore, they are known as blocking networks, and this is easy to prove (the number of possible switch configurations is smaller than the number of possible permutations of the input, e.g. $2^{3*8/2} = 4096 < 8! = 40320$ for $N = 8$). The network topology of the butterfly network can also be found in the bitonic sorter [61] as a building block. It is also the CAS-network found in FLiMS (chapter 3), which is able to sort any bitonic sequence.

Figure 6.1 shows a butterfly network, as found in the bitonic sorter, as well as a reverse butterfly network. The yellow box on the bitonic sorter indicates one of its two equally-sized butterfly networks near its output. The preferred visualisation of the butterfly network in the figure is

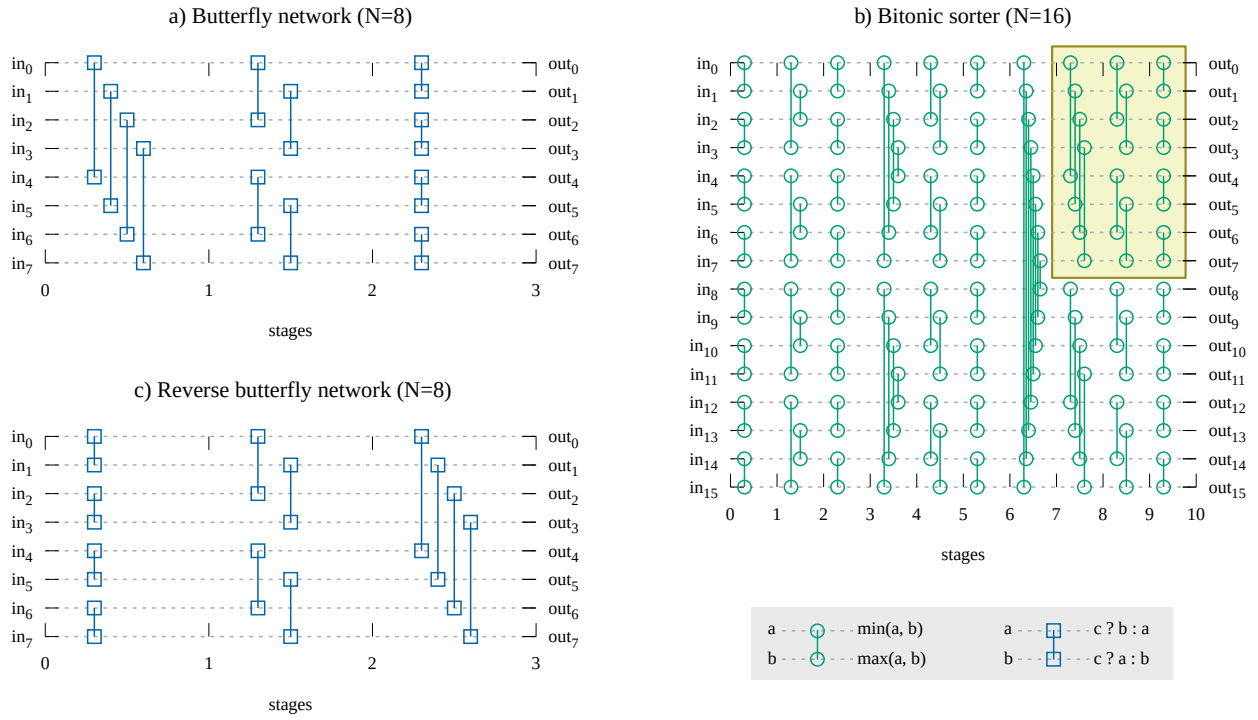


Figure 6.1: Butterfly network (a) as found in a bitonic sorter (b), and reverse butterfly (c)

not a conventional one that uses 2x2 switches [173], and this is to focus on the relevance to the bitonic sorter. Still, instead of compare-and-swap (CAS) units of the sorting networks, that operate on the magnitude of the input, the butterfly network uses 2x2 switches (visualised with connected squares instead of circles). There is a single control bit for every 2x2 switch, with a high value denoting a swap, and a low value denoting simple forwarding.

In the butterfly network case, a widely used algorithm [174] for the input of the control bits is a bit from the value. More specifically, the MSB of the values are for the switches of the first stage, MSB-1 for the switches of the second stage, and so on. If the inputs are a permutation of indexes (0 to N-1), this becomes a permutation network, but not for all possible permutations. In the cases of the inverted butterfly networks, in the studied use cases, the input of the control bits of each switch is more complex.

An already-studied use case for the inverse butterfly network is for implementing rotations [175]. The control logic is more complex than the barrel shifters, but the resulting circuit is close to that of barrel shifters, with $\log_2(N)$ steps. However, the implementation of barrel shifters is found to be more lightweight on ASICs, due to the optimisation to combine multiplexers [157].

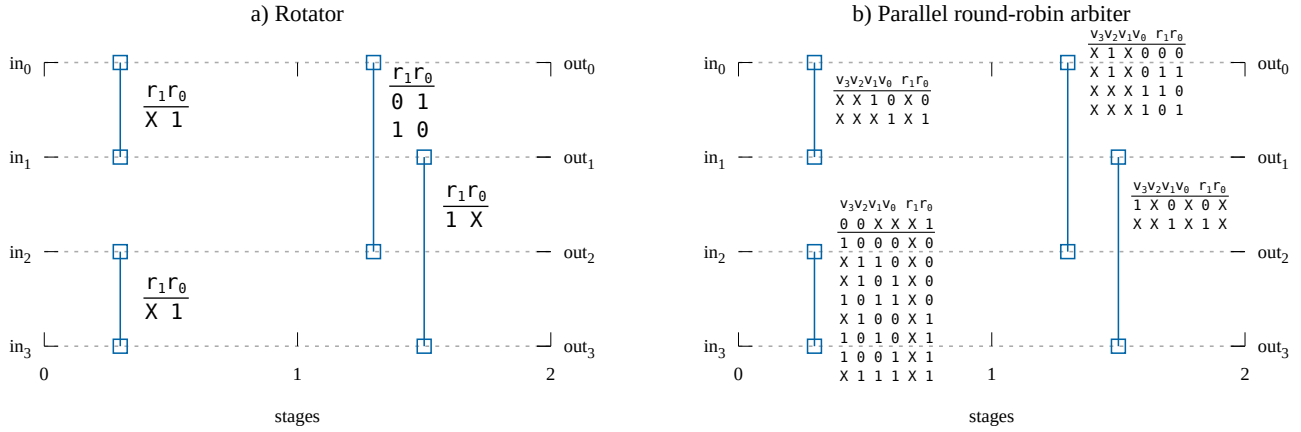


Figure 6.2: Reverse butterfly working as (a) a rotator, (b) a parallel round-robin arbiter

Figure shows the switch control logic required to implement a (a) rotator and (b) a stable parallel round-robin arbiter for four inputs. Since espresso [176] was used to derive the truth tables after exhaustive search in custom python code, the control logic is here expressed in disjunctive normal form (DNF) with don't cares (X). Each of the swap units in (a) reads the rotation bits (r_1, r_0) as shown in the corresponding table, and when a row matches the rotation, the swap is activated. For example, rotating ABCD by 1, would trigger both swaps in the first layer, resulting in BADC, and then the first swap of the 2nd layer, resulting in DABC. This is equivalent to the related work [157], [175], and is provided only for comparison.

In the implementation of the stable parallel round-robin arbiter (b), the logic of each swap unit also involves the valid bits of the inputs ($v_3, \dots, v_0$) from the previous stage. An example execution is provided for the input A_BC, where “_” is here for in_1 not being valid, and a rotation offset of 2. In the first layer the second swap unit is enabled through the rule X101 X0, resulting in A_CB. Then the first swap of the second stage is activated through the line XXX1 10, resulting in C_AB at the output, which is a concentration of the input starting from a desired rotation offset.

In order to prove that the reverse butterfly network routes all desired input combinations (passable permutations), it is enough to show that its output can be considered a bitonic sequence with respect to the order of its input (up to one local minimum and up to one local maximum). This is enough because regular (non-reverse) butterfly is shown to sort all bitonic

sequences in the bitonic sorter [61], and any scheduling that would allow routing of bitonic sequences, would also enable non-blocking routing for all bitonic sequences for sorted input, using the inverted circuit (with its inputs and outputs swapped).

The order/magnitude of the inputs of the reverse butterfly as a PRRA can be considered to be the port number. The possible order of the outputs can be seen as all rotations (according to the rotation offset) of the concatenation of the valid bit and the local order (there are different orders among the two subsets for valid and non-valid elements). For the stable version of the parallel round-robin arbiter, the local order of the valid subset must be the same as in the input port number. With respect to the local order of the non-valid set, the order does not matter for the requirements of the parallel round-robin arbiter. Therefore, in order for this to be a bitonic sequence, the local order of elements in the non-valid set is conveniently considered to be the negative of the respective port number in input. For no rotation, the resulting output is a bitonic sequence. This is because its form is $e_0, \dots, e_{n-1}, e_n, \dots, e_{P-1}$, where e_i is the local order of the i th element, and n is the number of valid inputs. The sublist $e_0, \dots, e_{n-1}$ is already sorted according to the port number, and the sublist $e_n, \dots, e_{P-1}$ is reversely sorted and every of its elements are below all elements in the first sublist. This only yields up to one local maximum e_{n-1} (or e_0 when all inputs are invalid), and up to one local minimum e_{P-1} (or e_0 when all inputs are valid), and thus is a bitonic sequence. For all other rotations, the output will still be a bitonic sequence, as a rotated bitonic sequence is still a bitonic sequence [134], finalising the proof that the required permutations to implement a stable PRRA are passable inside the reverse butterfly.

This alternative approach would reduce the amount of 2x2 switches from $P/2 \times (\log_2(P) + \log_2(P) \times (\log_2(P) + 1)/2)$ to only $P/2 \times (\log_2(P))$ for the parallel round-robin arbiter, although the logic complexity of the control bits is not very practical for implementation, at least in its current form. As this only considers the network complexity, it also does not calculate the rotation offset, but it would be an interesting starting point for further optimisation as future work.

6.2.3 Integration to real database management systems (DBMS)

Most of the proposed designs have been evaluated in small-scale systems, including on embedded platforms as a proof-of-concept. Integrating those designs to high-end systems for data center use would be an appropriate next step, bringing them one more step closer to production. Similarly, most of the benchmarking was done using micro-benchmarks, and would be appropriate to evaluate in a realistic environment, where the workload is representative of datacenter workload. Such work would address the challenges **C1**, **C3** and **C4**.

Note that due to the commercial sensitivity of the methodologies deployed in dunnhumby's tools, an out-of-context approach was preferred, and one of the goals of the thesis was to present framework-agnostic solutions for a potentially wider adoption.

6.2.4 A fully-streaming merge sorter on datacenter FPGAs

Similarly to integrating to DBMS, it could be also appropriate to integrate the proposed sorter of section 5.1 in larger-scale systems. This subsection presents a variation on the use of the merge-capable high-throughput sorter (figure 5.3) for today's datacenter FPGA, including a Xilinx Alveo U280 [117], which features its own DRAM. The idea behind the proposed solution is to use two merge-capable high-throughput sorters (here referenced as merge/sorters), to avoid stalling between merging k -sized chunks to k^2 -sized chunks from BRAM to DRAM.

Figure 6.3 presents the three different phases of such an arrangement. The phases of this approach with regard to the data movement start from 4 to distinguish from the internal phases of the merge/sorters (figure 5.1)). In phase 4, the merge/sorter A works as a fully-streaming k -size chunk sorter for an input of k^2 -sized chunk (internally as phase 1), while the merge/sorter B merges k different k -sized sorted chunks from BRAM to FPGA's DRAM (phase 2 internally). Phases 5 swaps the roles of the merge/sorters A and B, so that A now gets to merge its k -size chunks, and while A is busy merging, B will be accepting the new data. The phases 4 and 5 are interleaved on every k^2 elements, resulting in k^2 -sized sorted chunks in FPGA's DRAM. Finally, in phase 6, the BRAM works as a cache, and the merge/sorters work

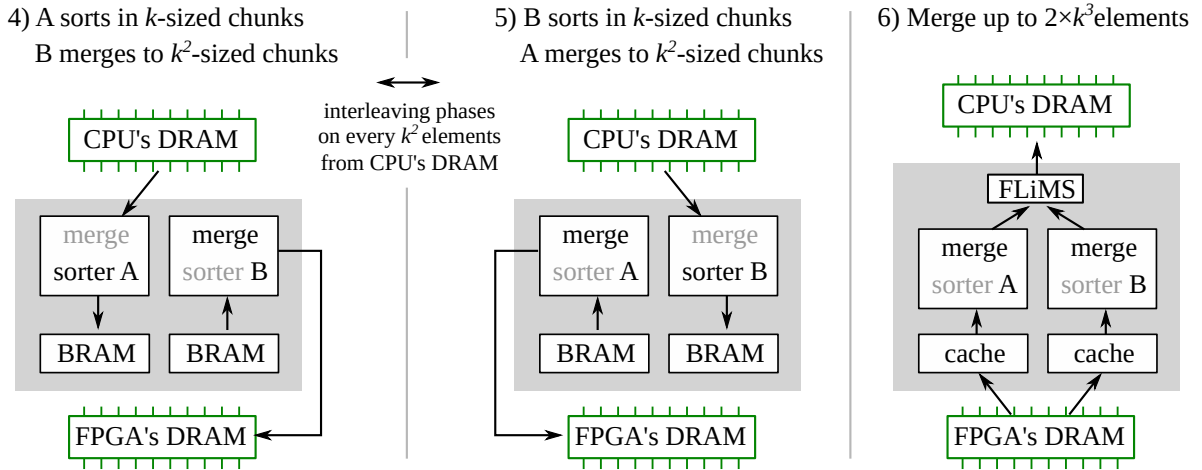


Figure 6.3: Proposed arrangement for fully-streaming merge sorting

both as mergers from FPGA's DRAM (internally with phase 3). Each of the merge/sorters will produce a sorted chunk of up to k^3 elements, and an additional FLiMS merger can be activated in the end to produce a sorted list of up to $2 \times k^3$ elements.

An example configuration would be for 64-bit elements and $k = 512$. By using this approach, it would be able to produce a sorted list of 2 GiB, only with offloading to the FPGA and reading the data back to the host. This can be achieved without waiting in-between the transfers, assuming non-special case data distributions, such as already-sorted. Experimental evaluation of this design as future work would help further with scalability **C4**.

6.2.5 Supporting reconfigurable SIMD instructions in high-end processors

Appendix C introduces Simodense [45], an open-source high-performance RISC-V softcore, in an attempt to facilitate future research on novel SIMD instructions. The general idea is that, by having small FPGAs working as custom instructions inside CPUs, all of the aforementioned challenges can be addressed to a degree. Note that this concept also targets the development challenge **C2** more directly than other designs in the thesis, by only requiring a few low-level lines of code to implement advanced accelerators [46]. Figure 6.4 shows a high-level example of a hypothetical architecture that enables a simple streaming application such as prefix sum

(see section C.3.3) to use a custom instruction implementation for computation and the CPU’s facilities for data movement.

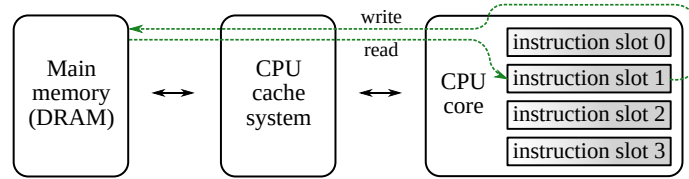


Figure 6.4: Reconfigurable instructions in CPUs: example data movement

Since Simodense is an FPGA-optimised softcore, it does not expose the challenges that would need to be resolved for applying the concept to higher-end processors. Most of the challenges for supporting reconfigurable instructions in high-end processors relate to the presence of internal states in the instruction designs. As modern operating systems need to support context-switching for multi-processing, a thread’s state needs to be able to be resumed. Another case of the need to access the states of the FPGA-based instruction is when wrong-path execution is supported, during which the processor’s pipelines need to be flushed from the wrongly-predicted data. Resuming the state of FPGA designs can be costly, as with the provided flexibility a global access to the registers of the design is not assumed. Otherwise, a serial storing and resuming approach would need to be followed, which would resemble bitstream loading. Alternatively, the required states could be stored inside vector registers, for example, but this would reduce the flexibility of the custom instructions. This could also complicate the processor design for achieving seamless access to a high number of states in the form of data through the instruction operands.

The higher the amount of internal states an instruction can hold, the more likely it is to achieve accelerator-like levels of performance in CPUs. A simple analogy to support this argument is with the design space exploration of numerous FPGA-based accelerators that exchange FPGA resources for performance, as with the sorter design space of section 5.1.3. Avoiding internal states in instructions altogether is possible, such as through combinatorial descriptions of instructions, but this would eliminate the flexibility of the instruction implementations. A related concern is with the size of the reconfigurable areas on the CPU, which would need further exploration to be decided. This is because larger areas would allow more complex designs, but

could introduce timing bottlenecks, also with respect to the reconfiguration time.

Another consideration with this approach is that CPUs operate at higher frequencies than FPGAs. For instance, this is the reason why NEON-based (SIMD) `memcpy()` implementations can achieve high bandwidth on ARM, even though a big portion of the instructions are not directly related to the data movement itself [177]. Given that isolated or out-of-context FPGA designs can run much faster than when integrated in bigger systems [178], the hardening of all communication in future processors with reconfigurable instructions could further close this gap.

Future research on using small FPGAs as instructions would be necessary to enable reconfigurable instructions in higher-end CPUs. This could involve novel programming models, instruction set architectures, verification techniques and other hardware and software mechanisms to abstract and handle internal states correctly and securely.

6.2.6 FPGA-friendly cache subsystem

As discussed, the nature of the memory access patterns heavily impacts design decisions on FPGAs. Appendix C introduces a purpose-built cache hierarchy for facilitating high-performance SIMD instructions on a softcore. A similar set of concepts could be applied into a compartmentalised IP, available for a wider range of FPGA applications. Linearising the access-pattern and introducing additional adaptive mechanisms, such as prefetching, as plug-ins, could be very useful for novel applications on FPGAs.

It is generally accepted that when switching the target application from a CPU to an FPGA, there might be the need to use an entirely different algorithm as a baseline for development, due to this limitation. For this reason, providing an easy means to efficient memory accesses, including optimisations to the inter-connection protocol, would be an alternative path to the more “radical” future architectural approach supported in the thesis.

Such a general cache module, could provide alternative forms of addressing, such as with content addressable tables (CATs), which have already been used in specialised cache hierarchies [29].

Combining advanced adaptive algorithms, such as Markov chain-based prefetching [15] in a general purpose cache module could help accelerate more traditional software-inspired designs for arbitrary problem sizes on FPGAs. This would ease the development of novel algorithms on FPGAs (challenges **C1** and **C2**).

6.2.7 ASIC equivalents and their placement in systems

One unexplored aspect of the provided database solutions, including the sorters and Simodense (appendix C), is how they would perform as hardened circuits. While it could be relatively straightforward to get an initial understanding of the performance, power and area requirements using a standard cell library, it would be appropriate to also study their method of integration in a real system.

In this scenario, a question would be whether the flexibility of FPGAs can also be of any use other than for the proposed functionalities in the thesis. For example, a hardened (combined) parallel round-robin arbiter could be reused in a similar way a NoCs is used in Versal FPGAs [126] for resource sharing and other applications. A high-performance network switch with reconfigurable regions could also prove beneficial for use in the datacenter. The latter could be used to offer computation offloading to network switches, in order to perform aggregation tasks, including implementing a Map Reduce model [179] in multi-node clusters and distributed algorithms.

This future work on partly hardening FPGA designs focuses more on the data movement side (challenge **C1**).

6.3 Concluding remarks

Fundamentally, FPGAs have a high potential for accelerating big data analytics and database systems, due to their high amounts of internal bandwidth. Their usual placement as a co-processor for accelerating specialised workloads has an overhead in development, deployment

and performance, which is more apparent in big data analytics. However, FPGAs offer unparalleled amounts of flexibility internally that can sometimes mitigate the shortcomings of inter-chip communication. Modular designs (distributed/parallel algorithms) such as FLiMS and the parallel round-robin arbiter as building blocks are demonstrated to accelerate workloads for arbitrarily long data with high-throughput. As the latest trend in achieving high-bandwidth in memories and interconnects is to increase the amount of information that is passed per cycle, the proposed designs aim to benefit from arbitrarily wide datapaths, rather than resorting to multi-tasking and data partitioning. Such design choices and optimisations help increase the number of different acceleration methodologies to overcome different restrictions of various acceleration platforms. The approach to first sort the data in different aggregation tasks achieves scalability to larger problems more easily, though for different problem requirements hash or cache-based algorithms can be more appropriate. Finally, it is predicted that future system architectures can benefit from bringing FPGAs closer to CPUs, and potentially reduce the amount of the silicon and computing cycles required to perform big data analytics.

Bibliography

- [1] M. Davis, *The undecidable: Basic papers on undecidable propositions, unsolvable problems and computable functions*. Courier Corporation, 2004.
- [2] N. A. Lynch and M. R. Tuttle, “Hierarchical correctness proofs for distributed algorithms,” in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, 1987, pp. 137–151.
- [3] A. Kottolli, “The economics of structured-and standard-cell-ASIC designs,” *Electronic News*, vol. 16, p. 2006, 2006.
- [4] T. Becker, O. Mencer, S. Weston, and G. Gaydadjiev, “Maxeler data-flow in computational finance,” in *FPGA Based Accelerators for Financial Applications*, Springer, 2015, pp. 243–266.
- [5] B. Bramas, “A novel hybrid quicksort algorithm vectorized using AVX-512 on Intel Skylake,” *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 8, no. 10, pp. 337–344, 2017.
- [6] “RISC-V ”V” Vector Extension, Version 0.9,” 2020.
- [7] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, *et al.*, “A reconfigurable fabric for accelerating large-scale datacenter services,” in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, IEEE, 2014, pp. 13–24.
- [8] H. Chen, S. Madaminov, M. Ferdman, and P. Milder, “FPGA-Accelerated Samplesort for Large Data Sets,” in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 222–232.

- [9] X. Inc., *Zynq UltraScale+ FPGA Product Tables and Product Selection Guide*, 2016–2021.
- [10] V. Kathail, “Xilinx vitis unified software platform,” in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 173–174.
- [11] K. Tatsumura, A. Dixon, and H. Goto, “FPGA-Based Simulated Bifurcation Machine,” in *29th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2019, pp. 59–66.
- [12] A. De Mauro, M. Greco, and M. Grimaldi, “A formal definition of big data based on its essential features,” *Library Review*, 2016.
- [13] Z. Dai and J. Zhu, “Saturating the transceiver bandwidth: Switch fabric design on fpgas,” in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, ACM, 2012, pp. 67–76.
- [14] A. Jaleel, H. H. Najaf-Abadi, S. Subramaniam, S. C. Steely, and J. Emer, “Cruise: Cache replacement and utility-aware scheduling,” in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 249–260.
- [15] P. Papaphilippou, P. H. J. Kelly, and W. Luk, “Pangloss: a novel Markov chain prefetcher,” in *The 3rd Data Prefetching Championship, held in conjunction with ISCA 2019*, 2019.
- [16] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, “High performance cache replacement using re-reference interval prediction (rrip),” *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 60–71, 2010.
- [17] W. Qiao, J. Oh, L. Guo, M.-C. F. Chang, and J. Cong, “Fans: Fpga-accelerated near-storage sorting,” in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2021, pp. 106–114.
- [18] M. Zhang, L. Li, H. Wang, Y. Liu, H. Qin, and W. Zhao, “Optimized compression for implementing convolutional neural networks on fpga,” *Electronics*, vol. 8, no. 3, p. 295, 2019.

- [19] H. Zhang, B. Zhao, W.-J. Li, Z.-G. Ma, and F. Yu, “Resource-Efficient Parallel Tree-Based Join Architecture on FPGA,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 66, no. 1, pp. 111–115, 2019.
- [20] *dunnhumby, source files*, <https://www.dunnhumby.com/source-files/>, [Online; accessed 14-April-2021].
- [21] M. H. Chotard, “Analyzing the effects of coupons and promotion in the grocery retail sector,” 2008.
- [22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, “Scikit-learn: Machine learning in python,” *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.
- [23] J. Casper and K. Olukotun, “Hardware acceleration of database operations,” in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, ACM, 2014, pp. 151–160.
- [24] M. Roozmeh and L. Lavagno, “Implementation of a performance optimized database join operation on FPGA-GPU platforms using OpenCL,” in *Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC), 2017 IEEE*, IEEE, 2017, pp. 1–6.
- [25] G. Alonso, Z. Istvan, K. Kara, M. Owaida, and D. Sidler, “doppioDB 1.0: Machine learning inside a relational engine,” *IEEE DE Bull*, vol. 42, no. 2, 2019.
- [26] M. Owaida, D. Sidler, K. Kara, and G. Alonso, “Centaur: A framework for hybrid CPU-FPGA databases,” in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, IEEE, 2017, pp. 211–218.
- [27] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras, “FPGA-based Multi-threading for In-Memory Hash Joins,” in *CIDR*, 2015.
- [28] R. Mueller, J. Teubner, and G. Alonso, “Sorting networks on fpgas,” *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 21, no. 1, pp. 1–23, 2012.

- [29] I. Absalyamov, P. Budhkar, S. Windh, R. J. Halstead, W. A. Najjar, and V. J. Tsotras, “Fpga-accelerated group-by aggregation using synchronizing caches,” in *Proceedings of the 12th International Workshop on Data Management on New Hardware*, 2016, pp. 1–9.
- [30] H.-J. Yang, K. Fleming, M. Adler, and J. Emer, “Optimizing under abstraction: Using prefetching to improve fpga performance,” in *2013 23rd International Conference on Field programmable Logic and Applications*, IEEE, 2013, pp. 1–8.
- [31] Q. Xiong, R. Patel, C. Yang, T. Geng, A. Skjellum, and M. C. Herbordt, “Ghosts: A transparent fpga-accelerated lossy compression framework,” in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2019, pp. 258–266.
- [32] P. Papaphilippou, C. Brooks, and W. Luk, “An Adaptable High-Throughput FPGA Merge Sorter for Accelerating Database Analytics,” in *2020 30th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2020, pp. 65–72.
- [33] K. Manev, A. Vaishnav, C. Kritikakis, and D. Koch, “Scalable filtering modules for database acceleration on fpgas,” in *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2019, pp. 1–6.
- [34] N. Samardzic, W. Qiao, V. Aggarwal, M.-C. F. Chang, and J. Cong, “Bonsai: High-performance adaptive merge tree sorting,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2020, pp. 282–294.
- [35] M. Saitoh, E. A. Elsayed, T. Van Chu, S. Mashimo, and K. Kise, “A high-performance and cost-effective hardware merge sorter without feedback datapath,” in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2018, pp. 197–204.
- [36] P. Papaphilippou, C. Brooks, and W. Luk, “FLiMS: Fast Lightweight Merge Sorter,” in *2018 International Conference on Field-Programmable Technology (FPT)*, IEEE, 2018, pp. 78–85.

- [37] J. Casper and K. Olukotun, “Hardware acceleration of database operations,” in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, ACM, 2014, pp. 151–160.
- [38] W. Zhang, Y. Wang, and K. A. Ross, “Parallel Prefix Sum with SIMD,” *Algorithms*, vol. 5, p. 31,
- [39] P. Papaphilippou, J. Meng, and W. Luk, “High-Performance FPGA Network Switch Architecture,” in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’20, Seaside, CA, USA: Association for Computing Machinery, 2020, pp. 76–85, ISBN: 9781450370998. DOI: 10.1145/3373087.3375299.
- [40] J. Barnes and P. Hut, “A hierarchical $O(n \log n)$ force-calculation algorithm,” *nature*, vol. 324, no. 6096, pp. 446–449, 1986.
- [41] P. Papaphilippou and W. Luk, “Accelerating database systems using FPGAs: A survey,” in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2018, pp. 125–130.
- [42] K. Manev and D. Koch, “Large Utility Sorting on FPGAs,” in *2018 International Conference on Field-Programmable Technology (FPT)*, IEEE, 2018.
- [43] P. Papaphilippou, H. Pirk, and W. Luk, “Accelerating the merge phase of sort-merge join,” in *29th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2019, pp. 100–105.
- [44] P. Papaphilippou, J. Meng, N. Gebara, and W. Luk, “Hipernetch: High-Performance FPGA Network Switch (in-press),” *ACM Transactions on Reconfigurable Technology and Systems*, 2021. DOI: 10.1145/3477054.
- [45] P. Papaphilippou, P. H. J. Kelly, and W. Luk, “Simodense: a RISC-V softcore optimised for exploring custom SIMD instructions,” in *2021 31st International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2021.
- [46] P. Papaphilippou, P. H. J. Kelly, and W. Luk, “Extending the RISC-V ISA for exploring advanced reconfigurable SIMD instructions,” in *The Fifth Workshop on Computer*

- Architecture Research with RISC-V (CARRV 2021)*, held in conjunction with ISCA 2021, 2021.
- [47] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten, “Monetdb: Two decades of research in column-oriented database,” *IEEE Data Engineering Bulletin*, 2012.
 - [48] D. Sidler, Z. István, M. Owaida, K. Kara, and G. Alonso, “doppioDB: A hardware accelerated database,” in *Proceedings of the 2017 ACM International Conference on Management of Data*, ACM, 2017, pp. 1659–1662.
 - [49] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang, “FPGA-accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-off,” in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, IEEE, 2017, pp. 160–167.
 - [50] M. Owaida, H. Zhang, C. Zhang, and G. Alonso, “Scalable inference of decision tree ensembles: Flexible design for cpu-fpga platforms,” in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2017, pp. 1–8.
 - [51] K. Kara, Z. Wang, C. Zhang, and A. Gustavo, “doppioDB 2.0: hardware techniques for improved integration of machine learning into databases,” *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 1818–1821, 2019.
 - [52] K. Kara, K. Eguro, C. Zhang, and G. Alonso, “ColumnML: Column-store machine learning with on-the-fly data transformation,” *Proceedings of the VLDB Endowment*, vol. 12, no. 4, pp. 348–361, 2018.
 - [53] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T.-H. Lee, *et al.*, “Oracle database in-memory: A dual format in-memory database,” in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, IEEE, 2015, pp. 1253–1258.
 - [54] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, “FPGAs in the cloud: Booting virtualized hardware accelerators with openstack,” in *Field-Programmable Cus-*

- tom Computing Machines (FCCM)*, 2014 IEEE 22nd Annual International Symposium on, IEEE, 2014, pp. 109–116.
- [55] C. Beckhoff, D. Koch, and J. Torresen, “Go ahead: A partial reconfiguration framework,” in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, IEEE, 2012, pp. 37–44.
- [56] C. Beckhoff, D. Koch, and J. Torresen, “Automatic floorplanning and interface synthesis of island style reconfigurable systems with goahead,” in *International Conference on Architecture of Computing Systems*, Springer, 2013, pp. 303–316.
- [57] K. Matas, T. La, N. Grunchevski, K. Pham, and D. Koch, “Invited tutorial: Fpga hardware security for datacenters and beyond,” in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 11–20.
- [58] I. K. Ganusov, M. A. Iyer, N. Cheng, and A. Meisler, “Agilex™ generation of intel® fpgas,” in *2020 IEEE Hot Chips 32 Symposium (HCS)*, IEEE Computer Society, 2020, pp. 1–26.
- [59] J. Harkins, T. El-Ghazawi, E. El-Araby, and M. Huang, “Performance of sorting algorithms on the SRC 6 reconfigurable computer,” in *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, IEEE, 2005, pp. 295–296.
- [60] R. Neininger and J. Straub, “Probabilistic analysis of the dual-pivot quicksort “count”,” in *2018 Proceedings of the Fifteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, SIAM, 2018, pp. 1–7.
- [61] K. E. Batchier, “Sorting networks and their applications,” in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, ACM, 1968, pp. 307–314.
- [62] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, “Efficient implementation of sorting on multi-core SIMD CPU architecture,” *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1313–1324, 2008.
- [63] D. Bundala and J. Závodný, “Optimal sorting networks,” in *International Conference on Language and Automata Theory and Applications*, Springer, 2014, pp. 236–247.

- [64] M. Codish, L. Cruz-Filipe, T. Ehlers, M. Müller, and P. Schneider-Kamp, “Sorting networks: To the end and back again,” *Journal of Computer and System Sciences*, vol. 104, pp. 184–201, 2019.
- [65] C.-Y. Lee and J.-M. Tsai, “A shift register architecture for high-speed data sorting,” *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 11, no. 3, pp. 273–280, 1995.
- [66] R. Perez-Andrade, R. Cumplido, C. Feregrino-Urbe, and F. M. Del Campo, “A versatile linear insertion sorter based on an FIFO scheme,” *Microelectronics Journal*, vol. 40, no. 12, pp. 1705–1713, 2009.
- [67] D. Koch and J. Torresen, “FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting,” in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, 2011, pp. 45–54.
- [68] W. Luk, V. Lok, and I. Page, “Hardware acceleration of divide-and-conquer paradigms: A case study,” in *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, 1993, pp. 192–201.
- [69] W. Song, D. Koch, M. Luján, and J. Garside, “Parallel hardware merge sorter,” in *24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2016, pp. 95–102.
- [70] R. Kobayashi and K. Kise, “Face: Fast and customizable sorting accelerator for heterogeneous many-core systems,” in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, IEEE, 2015, pp. 49–56.
- [71] A. Farmahini-Farahani, H. J. Duwe III, M. J. Schulte, and K. Compton, “Modular design of high-throughput, low-latency sorting units,” *IEEE Transactions on Computers*, vol. 62, no. 7, pp. 1389–1402, 2012.
- [72] S. Mashimo, T. Van Chu, and K. Kise, “High-performance hardware merge sorter,” in *25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2017, pp. 1–8.

- [73] M. Saitoh and K. Kise, “Very massive hardware merge sorter,” in *2018 International Conference on Field-Programmable Technology (FPT)*, IEEE, 2018, pp. 86–93.
- [74] E. A. Elsayed and K. Kise, “Towards an efficient hardware architecture for odd-even based merge sorter,” in *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc)*, IEEE, 2019, pp. 249–256.
- [75] E. A. Elsayed and K. Kise, “High-performance and hardware-efficient odd-even based merge sorter,” *IEICE Transactions on Information and Systems*, vol. 103, no. 12, pp. 2504–2517, 2020.
- [76] T. Usui, T. Van Chu, and K. Kise, “A cost-effective and scalable merge sorter tree on FPGAs,” in *2016 Fourth International Symposium on Computing and Networking (CANDAR)*, IEEE, 2016, pp. 47–56.
- [77] K. Manev, A. Vaishnav, and D. Koch, “Unexpected Diversity: Quantitative Memory Analysis for Zynq UltraScale+ Systems,” in *International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2019, pp. 179–187.
- [78] M. Zuluaga, P. Milder, and M. Püschel, “Computer generation of streaming sorting networks,” in *DAC Design Automation Conference 2012*, IEEE, 2012, pp. 1241–1249.
- [79] R. Chen, S. Siriyal, and V. Prasanna, “Energy and memory efficient mapping of bitonic sorting on FPGA,” in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2015, pp. 240–249.
- [80] A. Srivastava, R. Chen, V. K. Prasanna, and C. Chelmiss, “A hybrid design for high performance large-scale sorting on FPGA,” in *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on*, IEEE, 2015, pp. 1–6.
- [81] S.-W. Jun, S. Xu, *et al.*, “Terabyte sort on FPGA-accelerated flash storage,” in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2017, pp. 17–24.
- [82] S. Salamat, A. Haj Aboutaleb, B. Khaleghi, J. H. Lee, Y. S. Ki, and T. Rosing, “Nascent: Near-storage acceleration of database sort on smartssd,” in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 262–272.

- [83] R. Chen and V. K. Prasanna, “Accelerating Equi-join on a CPU-FPGA heterogeneous platform,” in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2016, pp. 212–219.
- [84] K. Kara, J. Giceva, and G. Alonso, “FPGA-based data partitioning,” in *Proceedings of the 2017 ACM International Conference on Management of Data*, ACM, 2017, pp. 433–445.
- [85] D. Sidler, Z. István, M. Owaida, and G. Alonso, “Accelerating pattern matching queries in hybrid CPU-FPGA architectures,” in *Proceedings of the 2017 ACM International Conference on Management of Data*, ACM, 2017, pp. 403–415.
- [86] R. Sidhu and V. K. Prasanna, “Fast regular expression matching using FPGAs,” in *Field-Programmable Custom Computing Machines, 2001. FCCM’01. The 9th Annual IEEE Symposium on*, IEEE, 2001, pp. 227–238.
- [87] M. Poess and C. Floyd, “New tpc benchmarks for decision support and web commerce,” *ACM Sigmod Record*, vol. 29, no. 4, pp. 64–71, 2000.
- [88] D. Ziener, F. Bauer, A. Becher, C. Dendl, K. Meyer-Wegener, U. Schürfeld, J. Teich, J.-S. Vogt, and H. Weber, “Fpga-based dynamically reconfigurable sql query processing,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 9, no. 4, pp. 1–24, 2016.
- [89] X.-T. Nguyen, H.-T. Nguyen, T.-T. Hoang, K. Inoue, O. Shimojo, T. Murayama, K. Tominaga, and C.-K. Pham, “An efficient FPGA-based database processor for fast database analytics,” in *Circuits and Systems (ISCAS), 2016 IEEE International Symposium on*, IEEE, 2016, pp. 1758–1761.
- [90] L. Woods, G. Alonso, and J. Teubner, “Parallel computation of skyline queries,” in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, IEEE, 2013, pp. 1–8.
- [91] R. Mueller, J. Teubner, and G. Alonso, “Glacier: A query-to-hardware compiler,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 1159–1162.

- [92] M. A. Sarkisla and A. Yurdakul, “Simdify: Framework for simd-processing with risc-v scalar instruction set,” in *2021 Australasian Computer Science Week Multiconference*, 2021, pp. 1–10.
- [93] C. Dennl, D. Ziener, and J. Teich, “On-the-fly composition of fpga-based sql query accelerators using a partially reconfigurable module library,” in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, IEEE, 2012, pp. 45–52.
- [94] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, “The evicted-address filter: A unified mechanism to address both cache pollution and thrashing,” in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, IEEE, 2012, pp. 355–366.
- [95] E. Szabo-Wexler, “Approximate membership of sets: A survey,”
- [96] A. Kirsch and M. Mitzenmacher, “Less hashing, same performance: Building a better bloom filter,” in *European Symposium on Algorithms*, Springer, 2006, pp. 456–467.
- [97] D. Kim, D. Oh, and W. W. Ro, “Design of power-efficient parallel pipelined bloom filter,” *Electronics letters*, vol. 48, no. 7, pp. 367–369, 2012.
- [98] T. Kocak and I. Kaya, “Low-power bloom filter architecture for deep packet inspection,” *IEEE Communications Letters*, vol. 10, no. 3, pp. 210–212, 2006.
- [99] M. Durand and P. Flajolet, “Loglog counting of large cardinalities,” in *European Symposium on Algorithms*, Springer, 2003, pp. 605–617.
- [100] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, “Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm,” 2007.
- [101] A. Kulkarni, M. Chiosa, T. B. Preußner, K. Kara, D. Sidler, and G. Alonso, “Hyperloglog sketch acceleration on fpga,” in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, 2020, pp. 47–56. DOI: 10.1109/FPL50879.2020.00019.

- [102] G. Cormode, “Data sketching,” *Commun. ACM*, vol. 60, no. 9, 48–55, Aug. 2017, ISSN: 0001-0782. DOI: 10.1145/3080008. [Online]. Available: <https://doi.org/10.1145/3080008>.
- [103] N. Bandi, A. Metwally, D. Agrawal, and A. El Abbadi, “Fast data stream algorithms using associative memories,” in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007, pp. 247–256.
- [104] D. Foley and J. Danskin, “Ultra-performance Pascal GPU and NVLink interconnect,” *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.
- [105] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, “Dissecting the nvidia volta gpu architecture via microbenchmarking,” *arXiv preprint arXiv:1804.06826*, 2018.
- [106] J. Choquette, O. Giroux, and D. Foley, “Volta: Performance and programmability,” *Ieee Micro*, vol. 38, no. 2, pp. 42–52, 2018.
- [107] AMD, *AMD EPYC(TM) 7003 series processors*, <https://www.amd.com/system/files/documents/amd-epyc-7003-series-datasheet.pdf>, [Online; accessed 14-April-2021], 2021.
- [108] Intel, *Intel(R) Xeon(R) Platinum 8380 Processor (60M Cache, 2.30 GHz) Specification*, <https://www.intel.com/content/www/us/en/products/sku/212287/intel-xeon-platinum-8380-processor-60m-cache-2-30-ghz/specifications.html>, [Online; accessed 14-April-2021], 2021.
- [109] Intel (R), *Intel intrinsics guide*. [Online]. Available: <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>.
- [110] X. Cheng, B. He, X. Du, and C. T. Lau, “A study of main-memory hash joins on many-core processor: A case with intel knights landing architecture,” in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, ACM, 2017, pp. 657–666.
- [111] D. Bates, A. Chadwick, and R. Mullins, “Configurable memory systems for embedded many-core processors,” *arXiv preprint arXiv:1601.00894*, 2016.

- [112] D. Bates, A. Bradbury, A. Koltes, and R. Mullins, “Exploiting tightly-coupled cores,” *Journal of Signal Processing Systems*, vol. 80, no. 1, pp. 103–120, 2015.
- [113] A. Olofsson, “Epiphany-v: A 1024 processor 64-bit RISC system-on-chip,” *CoRR*, vol. abs/1610.01832, 2016. arXiv: 1610.01832. [Online]. Available: <http://arxiv.org/abs/1610.01832>.
- [114] Ian Cutress and Anton Shilov, *The larrabee chapter closes: Intel’s final xeon phi processors now in eol*, [Online; accessed 14-April-2021], 2020. [Online]. Available: <https://www.anandtech.com/show/14305/intel-xeon-phi-knights-mill-now-eol>.
- [115] M. Sato, “The supercomputer “fugaku” and arm-sve enabled a64fx processor for energy-efficiency and sustained application performance,” in *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*, IEEE, 2020, pp. 1–5.
- [116] J. Stuecheli, W. J. Starke, J. D. Irish, L. B. Arimilli, D Dreps, B. Blaner, C. Wollbrink, and B. Allison, “IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI,” *IBM Journal of Research and Development*, vol. 62, no. 4/5, pp. 8–1, 2018.
- [117] S. Erusalagandi, *Alveo U280 Data Center Accelerator Card Data Sheet*, https://www.xilinx.com/support/documentation/data_sheets/ds963-u280.pdf, DS963 (v1.3), 2020.
- [118] R. Moussalli, “Tradeoffs and considerations in the design of accelerators for database applications,” in *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, IEEE, 2017, pp. 1615–1615.
- [119] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel, “CAPI: A coherent accelerator processor interface,” *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7–1, 2015.
- [120] T. P. Morgan, *Opening Up The Server Bus For Coherent Acceleration*, <https://www.nextplatform.com/2016/10/17/opening-server-bus-coherent-acceleration/>, [Online; accessed 30-March-2018], 2016.
- [121] A. B. Caldeira, “Ibm power system ac922 introduction and technical overview. ibm,” *International Technical Support Organization*, 2018.
- [122] K. Huang, “Multi-way Hash Join Based on FPGAs,” 2018.

- [123] C. Consortium *et al.*, “Cache coherent interconnect for accelerators (ccix),” *Online*. <http://www.ccixconsortium.com>, 2017.
- [124] J. Fang, J. Chen, J. Lee, Z. Al-Ars, and H. P. Hofstee, “An efficient high-throughput lz77-based decompressor in reconfigurable logic,” *Journal of Signal Processing Systems*, vol. 92, pp. 931–947, 2020.
- [125] A. Kumar, S. Fernando, Y. Ha, B. Mesman, and H. Corporaal, “Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 13, no. 3, p. 40, 2008.
- [126] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, “Xilinx adaptive compute acceleration platform: Versaltm architecture,” in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 84–93.
- [127] N. Hemsoth, *Intel Marrying FPGA, Beefy Broadwell for Open Compute Future*, <https://www.nextplatform.com/2016/03/14/intel-marrying-fpga-beefy-broadwell-open-compute-future/>, [Online; accessed 12-March-2018], 2016.
- [128] R. Balasubramonian and B. Grot, “Near-data processing,” *IEEE Micro*, vol. 36, no. 1, pp. 4–5, 2016.
- [129] B. Sukhwani, T. Roewer, C. L. Haymes, K.-H. Kim, A. J. McPadden, D. M. Dreps, D. Sanner, J. Van Lunteren, and S. Asaad, “Contutto: a novel FPGA-based prototyping platform enabling innovation in the memory subsystem of a server class processor,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2017, pp. 15–26.
- [130] P. Mehra, “Samsung smartssd: Accelerating data-rich applications,” *Proceedings of the Flash Memory Summit*, 2019.
- [131] N. Auger, C. Nicaud, and C. Pivoteau, “Merge strategies: From merge sort to tim-sort,” *working paper or preprint*, 2015. [Online]. Available: <https://hal-upec-upem.archives-ouvertes.fr/hal-01212839>.
- [132] P. Papaphilippou, *Source code for the paper ‘FLiMS: Fast Lightweight Merge Sorter’*, <http://philippos.info/merge>, [Online; accessed 28-April-2021], 2018.

- [133] P. Papaphilippou, *FPGA sorting algorithms visualisations*, http://philippos.info/sort_visual, [Online; accessed 28-April-2021], 2020.
- [134] G. Zachmann, “Adaptive bitonic sorting,” *Encyclopedia of Parallel Computing*, David Padua, Ed, pp. 146–157, 2013.
- [135] J. Gray, “Graysort benchmark,” *Sort Benchmark Home Page*—<http://sortbenchmark.org>, 2013.
- [136] A. Farmahini-Farahani, A. Gregerson, M. Schulte, and K. Compton, “Modular high-throughput and low-latency sorting units for FPGAs in the large hadron collider,” in *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*, IEEE, 2011, pp. 38–45.
- [137] *Boost.sort 3.- parallel algorithms*, [Online; accessed 9-April-2021]. [Online]. Available: <https://www.boost.org/doc/libs/develop/libs/sort/doc/html/sort/parallel.html>.
- [138] H. Inoue and K. Taura, “Simd-and cache-friendly algorithm for sorting an array of structures,” *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1274–1285, 2015.
- [139] P. Gupta and N. McKeown, “Designing and implementing a fast crossbar scheduler,” *IEEE micro*, vol. 19, no. 1, pp. 20–28, 1999.
- [140] S. Q. Zheng, M. Yang, J. Blanton, P. Golla, and D. Verchere, “A simple and fast parallel round-robin arbiter for high-speed switch control and scheduling,” in *The 2002 45th Midwest Symposium on Circuits and Systems, 2002. MWSCAS-2002.*, IEEE, vol. 2, 2002, pp. II–II.
- [141] X. Inc., *UltraScale+ FPGA Product Tables and Product Selection Guide*, 2015-2019.
- [142] X. Inc, *Virtex UltraScale+ FPGA Data Sheet: DC and AC Switching Characteristics (DS923)*, 2019.
- [143] W. D. Hillis and G. L. Steele Jr, “Data parallel algorithms,” *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.

- [144] J. Meng, N. Gebara, H.-C. Ng, P. Costa, and W. Luk, "Investigating the feasibility of fpga-based network switches," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE, 2019.
- [145] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar, "Matching output queueing with a combined input/output-queued switch," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1030–1039, 1999.
- [146] P. Papaphilippou, *Source code for the paper 'High-Performance FPGA Network Switch Architecture'*, <http://philippos.info/switch>, [Online; accessed 28-April-2021], 2020.
- [147] M. S. Abdelfattah, A. Bitar, and V. Betz, "Take the highway: Design for embedded nocs on fpgas," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2015, pp. 98–107.
- [148] A. Bitar, J. Cassidy, N. Enright Jerger, and V. Betz, "Efficient and programmable ethernet switching with a noc-enhanced fpga," in *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems*, ACM, 2014, pp. 89–100.
- [149] N. McKeown, "The islip scheduling algorithm for input-queued switches," *IEEE/ACM transactions on networking*, no. 2, pp. 188–201, 1999.
- [150] Arista Networks, Inc, *7800R3 Series Data Center Switch Router Data Sheet*, 2019.
- [151] Y. Kanizo, D. Hay, and I. Keslassy, "The crosspoint-queued switch," in *IEEE INFOCOM 2009*, IEEE, 2009, pp. 729–737.
- [152] J. Kim, W. J. Dally, B. Towles, and A. K. Gupta, "Microarchitecture of a high-radix router," in *ACM SIGARCH Computer Architecture News*, IEEE Computer Society, vol. 33, 2005, pp. 420–431.
- [153] Y. Li, S. Panwar, and H. J. Chao, "The dual round robin matching switch with exhaustive service," in *Workshop on High Performance Switching and Routing, Merging Optical and IP Technologie*, IEEE, 2002, pp. 58–63.

- [154] T. Jain and K. Schneider, "Routing partial permutations in interconnection networks based on radix sorting," in *2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, IEEE, 2018, pp. 1–10.
- [155] X. U. G. UG612, *Timing closure user guide*, 2012.
- [156] S. Erusalagandi, *UltraFast Design Methodology Guide for the Vivado Design Suite*, UG949 (v2019.2), 2019.
- [157] N. Burgess, "Assessment of butterfly network vlsi shifter circuit," in *2010 Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers*, IEEE, 2010, pp. 92–96.
- [158] A. Canis, J. H. Anderson, and S. D. Brown, "Multi-pumping for resource reduction in fpga high-level synthesis," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 194–197. DOI: 10.7873/DATE.2013.053.
- [159] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker, "High-speed switch scheduling for local-area networks," *ACM Transactions on Computer Systems (TOCS)*, vol. 11, no. 4, pp. 319–352, 1993.
- [160] Y. Li, S. Panwar, and H. J. Chao, "On the performance of a dual round-robin switch," in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, IEEE, vol. 3, 2001, pp. 1688–1697.
- [161] J. Chao, "Saturn: A terabit packet switch using dual round robin," *IEEE Communications Magazine*, vol. 38, no. 12, pp. 78–84, 2000.
- [162] C. He and K. L. Yeung, "D-lqf: An efficient distributed scheduling algorithm for input-queued switches," in *2011 IEEE International Conference on Communications (ICC)*, IEEE, 2011, pp. 1–5.
- [163] P. Papaphilippou, *Source code for the paper 'An Adaptable High-Throughput FPGA Merge Sorter for Accelerating Database Analytics'*, <http://philippos.info/sorter>, [Online; accessed 28-April-2021], 2020.

- [164] A. Holdings, *Arm cortex-a53 mpcore processor, technical reference manual: Data prefetching and monitoring*, 2014. [Online]. Available: <https://developer.arm.com/documentation/ddi0500/e/level-1-memory-system/data-prefetching/data-prefetching-and-monitoring>.
- [165] A. Holdings, *ARM Cortex-A Series Programmer's Guide for ARMv8-A*.
- [166] J. Ortiz and D. Andrews, "A streaming high-throughput linear sorter system with contention buffering," *International Journal of Reconfigurable Computing*, 2011.
- [167] M. Zuluaga, P. Milder, and M. Püschel, "Streaming sorting networks," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 21, no. 4, pp. 1–30, 2016.
- [168] D. Li, L. Huang, T. Gao, Y. Feng, A. Tavares, and K. Wang, "An Extended Non-Strict Partially Ordered Set Based Configurable Linear Sorter on FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.
- [169] P. Papaphilippou, *Source code for the paper 'Accelerating the merge phase of sort-merge join'*, <http://philippos.info/mergejoin>, [Online; accessed 28-April-2021], 2019.
- [170] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden, "Materialization strategies in a column-oriented DBMS," in *2007 IEEE 23rd International Conference on Data Engineering*, IEEE, 2007, pp. 466–475.
- [171] H. D. Nguyen, B. Pasca, and T. B. Preußner, "FPGA-specific arithmetic optimizations of short-latency adders," in *2011 21st International Conference on Field Programmable Logic and Applications*, IEEE, 2011, pp. 232–237.
- [172] A Holdings, "Amba axi and ace protocol specification," *Tech. rep. 2011*, 2013.
- [173] M. J. Narasimha, "The batcher-banyan self-routing network: Universality and simplification," *IEEE Transactions on Communications*, vol. 36, no. 10, pp. 1175–1178, 1988.
- [174] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Transactions on Computers*, vol. 100, no. 12, pp. 1145–1155, 1975.

- [175] Y. Hilewitz and R. B. Lee, “Performing advanced bit manipulations efficiently in general-purpose processors,” in *18th IEEE Symposium on Computer Arithmetic (ARITH’07)*, IEEE, 2007, pp. 251–260.
- [176] R. Brayton, G. Hachtel, L. Hemachandra, A. Newton, and A. Sangiovanni-Vincentelli, “A comparison of logic minimization strategies using espresso: An apl program package for partitioned logic minimization,” in *Proceedings of the International Symposium on Circuits and Systems*, 1982, pp. 42–48.
- [177] “A.3.2. Cortex-A8 Data memory access,” in *NEON Programmer’s Guide*, arm, 2013, pp. 5–7. [Online]. Available: <https://developer.arm.com/documentation/den0018/a>.
- [178] M. A. Islam, H. Miyazaki, and K. Kise, “RVCoreP-32IM: An effective architecture to implement mul/div instructions for five stage RISC-V soft processors,” *arXiv preprint arXiv:2010.16171*, 2020.
- [179] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [180] N. Dao, A. Attwood, B. Healy, and D. Koch, “Flexbex: A risc-v with a reconfigurable instruction extension,” in *2020 International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2020, pp. 190–195.
- [181] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, “Efficient implementation of sorting on multi-core simd cpu architecture,” *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1313–1324, 2008.
- [182] H. Nakahara, Z. Que, and W. Luk, “High-Throughput Convolutional Neural Network on an FPGA by Customized JPEG Compression,” in *28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2020, pp. 1–9.
- [183] A. Waterman and K. Asanovic, “The RISC-V instruction set manual, volume I: Unprivileged ISA document, version 20191214-draft,” *RISC-V Foundation, Tech. Rep*, 2020.

- [184] J. R. G. Ordaz and D. Koch, “soft-NEON: A study on replacing the NEON engine of an ARM SoC with a reconfigurable fabric,” in *27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE, 2016, pp. 229–230.
- [185] Ordaz, Jose Raul Garcia and Koch, Dirk, “A Soft Dual-Processor System with a Partially Run-Time Reconfigurable Shared 128-Bit SIMD Engine,” in *29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE, 2018, pp. 1–8.
- [186] P. Papaphilippou, *Source code repository for a RISC-V softcore optimised for exploring advanced reconfigurable SIMD instructions*, <http://philippos.info/simodense>, [Online; accessed 25-May-2021], 2021.
- [187] T. UltraSPARC, *Supplement to the ultrasparc architecture 2007*, 2006.
- [188] C. Heinz, Y. Lavan, J. Hofmann, and A. Koch, “A catalog and in-hardware evaluation of open-source drop-in compatible risc-v softcore processors,” in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, IEEE, 2019, pp. 1–8.
- [189] A. Interconnect, “V2. 1 logicore ip product guide,” *PG059, Xilinx, December*, vol. 20, 2017.
- [190] P. Papaphilippou, P. H. J. Kelly, and W. Luk, “Demonstrating custom SIMD instruction development for a RISC-V softcore,” in *2021 31st International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2021.
- [191] J. D. McCalpin, “Stream benchmark,” *Link: www. cs. virginia. edu/stream/ref. html#what*, vol. 22, 1995.
- [192] C. Wolf, *Picorv32-a size-optimized risc-v cpu*, 2019.
- [193] S. Mashimo, A. Fujita, R. Matsuo, S. Akaki, A. Fukuda, T. Koizumi, J. Kadomoto, H. Irie, M. Goshima, K. Inoue, *et al.*, “An Open Source FPGA-Optimized Out-of-Order RISC-V Soft Processor,” in *2019 International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2019, pp. 63–71.

- [194] K. Asanovic, D. A. Patterson, and C. Celio, “The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor,” University of California at Berkeley Berkeley United States, Tech. Rep., 2015.
- [195] E. Matthews and L. Shannon, “Taiga: A new risc-v soft-processor framework enabling high performance cpu architectural features,” in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2017, pp. 1–4.
- [196] Intel (R), *Intel(r) integrated performance primitives: Developer reference, volume 1: Signal processing*. [online; accessed 11-january-2021]. [Online]. Available: <https://software.intel.com/content/www/us/en/develop/documentation/ipp-dev-reference/top/volume-1-signal-and-data-processing.html>.
- [197] E. A. Elsayed and K. Kise, “High-performance and hardware-efficient odd-even based merge sorter,” *IEICE Transactions on Information and Systems*, vol. 103, no. 12, pp. 2504–2517, 2020.
- [198] W. D. Hillis and G. L. Steele Jr, “Data parallel algorithms,” *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.
- [199] www.lowrisc.org, “Ibex RISC-V,” [Online]. Available: <https://github.com/lowRISC/ibex>.
- [200] J. Miura, H. Miyazaki, and K. Kise, “A portable and linux capable risc-v computer system in verilog hdl,” *arXiv preprint arXiv:2002.03576*, 2020.
- [201] E. Matthews, A. Lu, Z. Fang, and L. Shannon, “Rethinking integer divider design for fpga-based soft-processors,” in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2019, pp. 289–297.
- [202] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, *et al.*, “Chipyard: Integrated design, simulation, and implementation framework for custom socs,” *IEEE Micro*, 2020.
- [203] F. Kermarrec, S. Bourdeauducq, H. Badier, and J.-C. Le Lann, “LiteX: an open-source SoC builder and library based on Migen Python DSL,” in *OSDA 2019, colocated with DATE 2019 Design Automation and Test in Europe*, 2019.

- [204] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, “Sonicboom: The 3rd generation berkeley out-of-order machine,” in *Fourth Workshop on Computer Architecture Research with RISC-V*, 2020.
- [205] H.-C. Ng, C. Liu, and H. K.-H. So, *A soft processor overlay with tightly-coupled fpga accelerator*, 2016. arXiv: 1606.06483 [cs.AR].
- [206] P. Yiannacouras, J. G. Steffan, and J. Rose, “Vespa: Portable, scalable, and flexible fpga-based vector processors,” in *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, 2008, pp. 61–70.
- [207] J. Yu, G. Lemieux, and C. Eagleston, “Vector processing as a soft-core cpu accelerator,” in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, 2008, pp. 222–232.
- [208] A. Severance and G. Lemieux, “Venice: A compact vector processor for fpga applications,” in *2012 International Conference on Field-Programmable Technology*, IEEE, 2012, pp. 261–268.
- [209] M. Johns and T. J. Kazmierski, “A minimal risc-v vector processor for embedded systems,” in *2020 Forum for Specification and Design Languages (FDL)*, IEEE, 2020, pp. 1–4.
- [210] C. Chen, X. Xiang, C. Liu, Y. Shang, R. Guo, D. Liu, Y. Lu, Z. Hao, J. Luo, Z. Chen, *et al.*, “Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance risc-v processor with vector extension: Industrial product,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2020, pp. 52–64.
- [211] A. Severance and G. G. Lemieux, “Embedded supercomputing in fpgas with the vector-blox mxp matrix processor,” in *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, IEEE, 2013, pp. 1–10.

Appendices

Appendix A

FLiMS-based sorting function using SIMD intrinsics and optimisation

This appendix introduces the most important parts of the design space exploration needed for fine-tuning the FLiMS-based C++ function implementation (section 3.6). The presented techniques were useful for deciding the final form of the FLiMS merge function, as well as the sort-in-chunks function through bitonic sorting.

A.1 Merge function fine-tuning

First, while developing the FLiMS representation using SIMD-intrinsics, there is the question of how wide FLiMS should be. For example, for $w = 8$, eight integers (32-bit-wide) would be able to be fetched and processed by each SIMD instruction, and would be closer to a physical representation of 8 *MAX* units in a single vector register. However, with the same 256-bit extension set, it is also easy to emulate FLiMS of more inputs, such as with $w = 32$, because of the regularity of the bitonic partial merge structure in FLiMS. For instance, for double width, one more stage is added to the CAS network, and the remaining stages would use approximately double the instructions of a single width.

With respect to the implementation of the selector stage (MAX units), the cB is represented in reverse, and this is also required between the vector registers holding the cB values. For example, for $w = 16$ loading the first half would require fetching the second half of 8 integers, after reversing it.

There is an alternative approach to implement the distributed part (MAX units). The idea is to maintain vectors of seemingly independent pointers corresponding to the series of registers cA and cB in vectors. The pointers will point to consecutive/rotated locations in the input lists and they are updated according to the algorithm requirements (the vector of pointers for the second list is in reverse order). This is achieved by calling mask/gather intrinsics, which selectively update the vectors with the respective memory locations as pointed by the pointers. As the gather instruction is relatively expensive in AVX2 [109], this approach has not been used, even though it resembles the original distributed algorithm of FLiMS more closely.

The proposed approach to represent FLiMS with SIMD intrinsics of section 3.6.1 is scalable. The first layers of the butterfly network are more straightforward, as the workload can be easily split across multiple calls of the same intrinsics. For the last few, only half of the min/max result is used, because it is found to be more costly to align the inputs for fully utilising the min/max units instead. Figure A.1 summarises how the butterfly network of FLiMS is generated for an increased number of inputs, and what a pair of vector registers can implement in each layer.

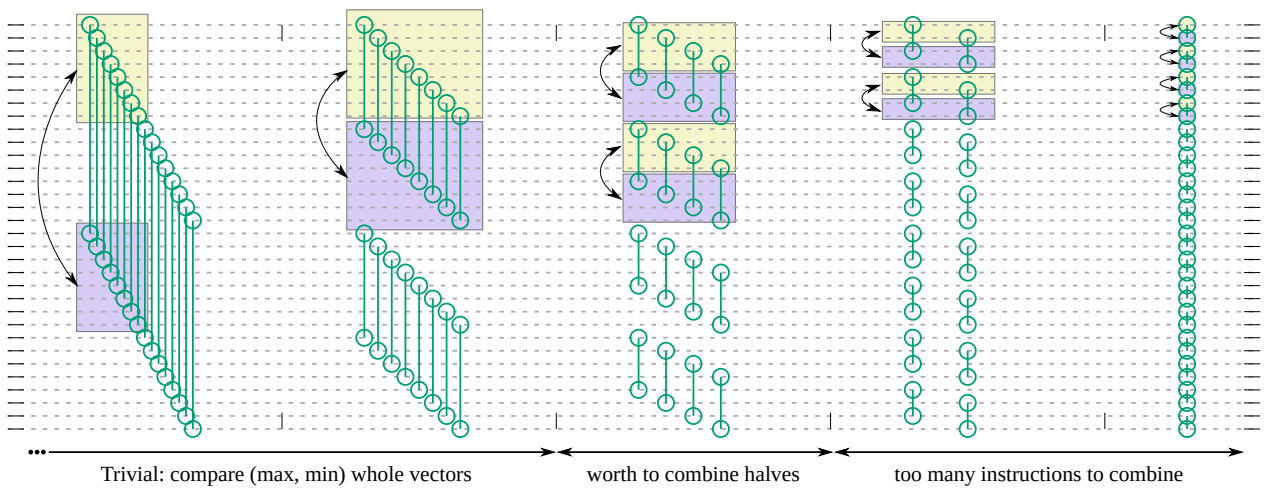


Figure A.1: Vectorising the CAS network of FLiMS using min, max and shuffle AVX2 intrinsics

In principle, emulating a wider FLiMS can result in increased data locality, as more computation

is done in registers. On the other hand, more logic needs to be represented, and the instruction count of the loop increases, with the additional disadvantage of less-obvious data dependencies. Since the number of vector units are limited in a processor, emulating more logic on the merge loop can trigger more cache accesses and worsen the performance.

Figure A.2 shows how the degree of parallelism w of the emulated FLiMS influences the achievable throughput on the AVX2-native processor i7-8809G, running at 4.2 GHz. The same experiment was done using three optimising compilers and the input lists size was 2^{24} integers, with each point representing an average of 25 runs. A Python script was written to produce the desired SIMD-based FLiMS for a given degree of parallelism w . For high values of w , this resulted in tens of thousands of lines of code in C. The main conclusions are that for $w = 16$ or $w = 32$ the throughput is the highest, and there is little variation between different compilers.

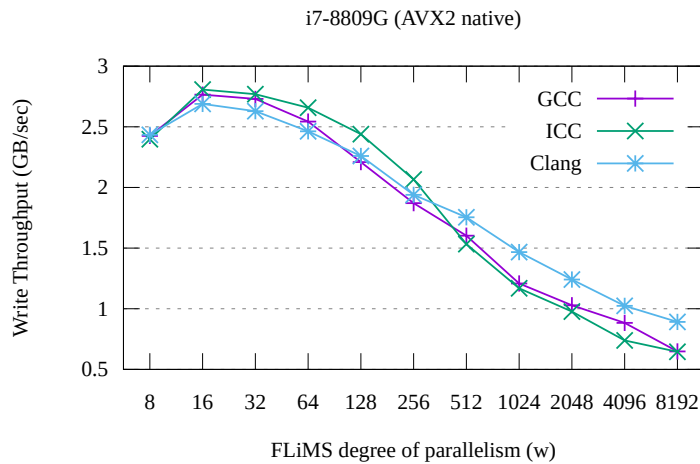


Figure A.2: Finding the optimal w value for AVX2-based FLiMS 2-way merger

A.2 Merge function implementation using AVX-512

Next, the performance when using AVX-512 (using 512-bit vectors) instead of AVX2 is explored individually. The equivalent Python code-generator script is developed to produce the 512-bit SIMD code, hence starting from $w = 16$ for a 32-bit FLiMS. The performance evaluation was done on an Amazon EC2 droplet featuring the Xeon 8124M, running at 3.4 GHz for single thread, though there could be a degradation due to the multi-user environment. Figure A.3

shows the results of this exploration (right), plus the AVX2 code on the same processor (left) to see the benefits of using AVX-512.

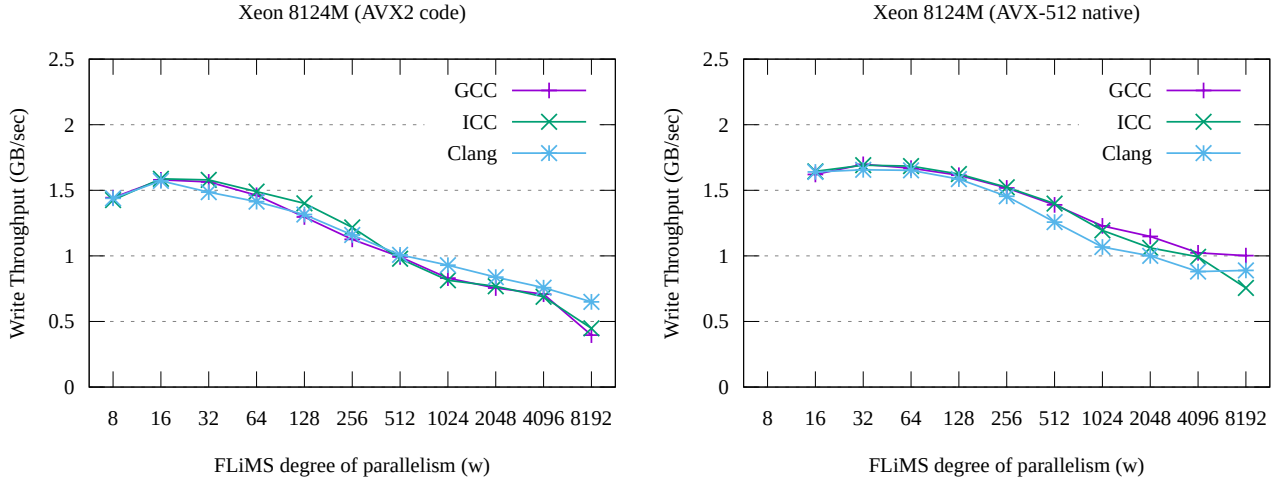


Figure A.3: AVX2 vs AVX-512 FLiMS performance comparison on Xeon 8124M

The main outcome from this experiment is that AVX-512 offers little benefits over AVX2, at least with this processor implementation and FLiMS implementation. It is interesting that the performance of the AVX-512 version of FLiMS peaks at $w = 32$, which is the equivalent result for AVX2's $w = 16$ by using double the vector width. The AVX2 version on the non-server processor i7-8809G from figure A.2 is higher than both the Xeon 8124M experiments, as Xeons are most optimised for multi-threading rather than single-thread performance. Still, the advantages of AVX-512 remain questionable for this application.

A.3 Sorting in chunks using bitonic sort

Finally, for the *sort-in-chunks* function it is appropriate to see what is the optimal sorted chunk size. The methodology to generate its intrinsics representation is very similar to the one described in high-level for the butterfly network inside FLiMS. For smaller chunk sizes, there need to be more merge iterations. For larger chunk sizes, FLiMS benefits from the streaming access pattern, but as with FLiMS, a larger logic representation of the bitonic sorter results in an increased instruction count and working set from more distant data localities. Since this experiment implements the entire AVX2-FLiMS-accelerated merge sort, the baseline is

`std::sort()` for unsorted input. Figure A.4 demonstrates that a sorted chunk size of 512 integers is the optimal chunk size, for an input size of 2^{27} . The slight oscillation is expected and is due to a temporary space optimisation for merge sort, where an odd number of merge iterations requires an additional `memcpy()` at the end, as halving the chunk size increases the number of merge iterations by one.

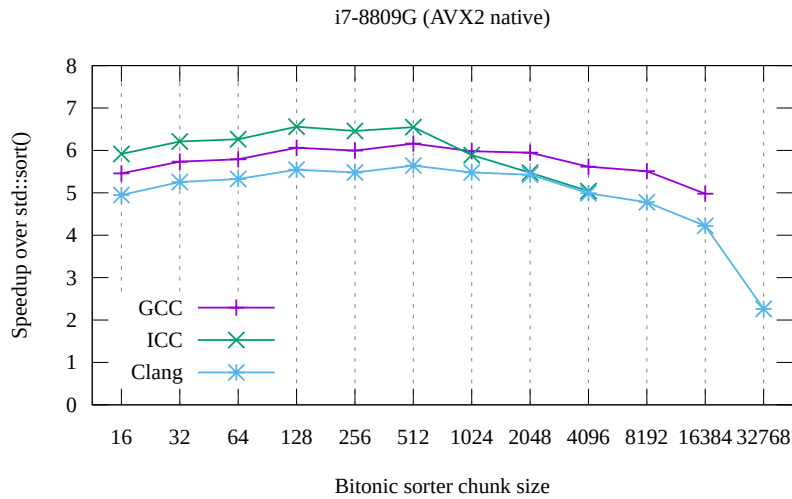


Figure A.4: Finding the optimal chunk size for the sort-in-chunks loop that uses an AVX2-based bitonic sorter

Omitted from the presented exploration are some lower-level optimisations, including reordering the SIMD intrinsics. Such optimisations improved the performance at a lower degree, and seemed more sensitive to the compiler, as well as the micro-architectural features of the processor.

Here some inefficiencies can be observed, including the fact that there are two comparisons for each CAS unit (since there are calls for both a min and max instruction). Appendix C discusses a possible future architecture path to solve the shortcomings of vectorisation using a fixed set of SIMD intrinsics, and demonstrates a solution for a similar use case using an FPGA-based softcore.

A.4 Comparing sorting performance on different micro-architectures

The first sets of results are illustrated in figure A.5. The performance of a FLiMS-based sort function is compared against the C++ Standard Library implementation of sorting `std::sort()`, as well as a highly-optimised [5] SIMD-based radix sort implementation from Intel’s integrated performance primitives (IPP). This is done for two Intel CPUs of different generations, the i7-6700 on the left, and i7-8809G on the right. The y-axis represents the speedup over `std::sort()` as it is a popular baseline and its performance is found to be relatively consistent across different environments.

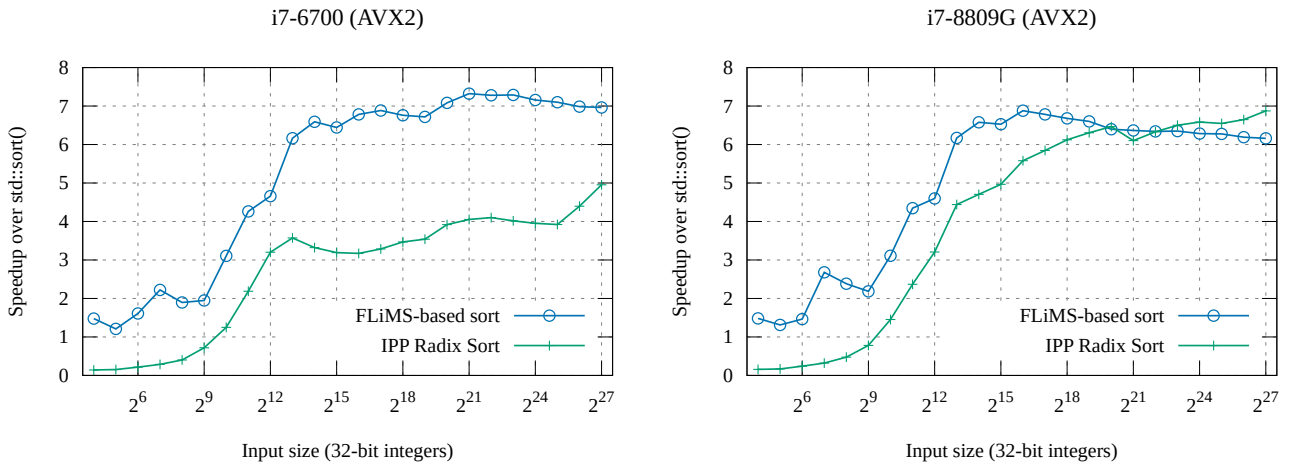


Figure A.5: Comparison of FLiMS-based sorting and alternatives on Intel CPUs of different generations

As an observation, FLiMS-based sort is consistently outperforming the alternatives on the older generation, with around a 2x speedup over radix sort, and plateaus to around 7 times over `std::sort()` for lengthier inputs. Though on the newer generation, radix sort seems to marginally win towards the end of the x-axis range.

Some additional exploration was done to identify the reason for variation in the performance of IPP radix sort, including manipulating the number of memory channels used by the processor. However, the reason is still yet to be found. This could relate to the individual processor implementation, and further investigation would be outside of the scope of the thesis.

The results presented in this figure were based on GCC-compiled binaries, but there was relatively little variation between compilers in this experiment. More generally, the performance of FLiMS seems more consistent across different architectures and compilers, also considering the results of sections 3.6.1 and 3.6.3.

Appendix B

FPGA-based network switch implementations dataset

This appendix presents the dataset results of the network switch design space exploration (section 4.5.2), which was also used in the journal extension of the network switch [44]. This is done for the Xilinx Alveo U280 FPGA, and is summarised in Table B.1.

The first set of columns contains the network switch configuration, as well as the optimised pipeline latency ($latency_{opt}$) after the latency reduction optimisation. The middle set of columns contains the resource utilisation (LookUp Tables (LUT) and Flip-Flop registers (FF)) and timing characteristics (Worst Negative Slack (WNS) and maximal operating frequency (f_{max}) for an initial target clock frequency of 600 MHz, after synthesis and place-and-route by Vivado 2020.1 for U280. The final set of columns includes some additional performance metrics derived indirectly, such as the port-to-port latency in nanoseconds, that is the $latency_{opt}$ plus one cycle, representing the minimum time a packet will stay in a queue, multiplied by the clock period. The throughput is the product of the operating frequency and the packet size. Finally, the last column is the product of the number of ports (P) and the throughput, and represents the aggregate throughput for all ports in the switch.

Not discussed in the rest of the thesis are the 32-port switch configurations, out of which only two were successfully implemented: 256-bit switches with $S=3$ and $S=5$. Those implemen-

Table B.1: Synthesis and implementation dataset, based on reports from Vivado 2020.1

P	packet size	S	$latency_{opt}$	LUT	FF	WNS (ns)	f_{max} (MHz)	p2p lat. (ns)	Thr. (Gb/s)	Agg. Thr. (Gb/s)
2	256	1	4	1877	4803	-0.89	391.44	12.77	100.21	200.42
		2	2	2381	3778	-0.74	416.38	7.21	106.59	213.19
		3	1	1874	2734	-0.95	382.46	5.23	97.91	195.82
	512	1	4	3523	9419	-0.73	417.07	11.99	213.54	427.08
		2	2	4463	7378	-1.29	338.10	8.87	173.11	346.22
		3	1	3448	5294	-1.01	373.46	5.36	191.21	382.42
	256	1	7	8516	18791	-1.28	339.60	23.56	86.94	347.75
		2	4	7881	13662	-0.85	397.98	12.56	101.88	407.54
		3	2	10282	11613	-1.24	343.92	8.72	88.04	352.17
		4	2	8070	11490	-0.57	447.09	6.71	114.46	457.82
		5	1	8236	7378	-0.93	385.41	5.19	98.66	394.66
	512	1	7	15985	37255	-0.84	398.46	20.08	204.01	816.04
		2	4	14815	27038	-0.94	384.07	13.02	196.65	786.58
		3	2	19158	22990	-0.93	384.96	7.79	197.10	788.40
		4	2	14975	22754	-0.97	378.98	7.92	194.04	776.15
		5	1	15092	14546	-0.91	388.25	5.15	198.78	795.13
8	256	1	10	40783	84236	-0.80	405.24	27.14	103.74	829.93
		2	5	45283	60522	-1.13	357.44	16.79	91.50	732.04
		3	4	38223	44037	-0.97	379.27	13.18	97.09	776.74
		4	2	38678	39763	-1.74	293.20	10.23	75.06	600.47
		5	2	38864	39498	-1.88	282.11	10.63	72.22	577.77
		6	2	41234	39404	-2.15	261.73	11.46	67.00	536.03
		7	1	41967	22920	-3.07	211.03	9.48	54.02	432.19
	512	1	10	80856	167505	-1.75	292.34	37.63	149.68	1197.43
		2	5	91110	120289	-1.55	311.17	19.28	159.32	1274.56
		3	4	77778	87547	-1.66	301.05	16.61	154.14	1233.12
		4	2	77068	78933	-2.25	255.45	11.74	130.79	1046.32
		5	2	80418	78575	-2.55	237.44	12.64	121.57	972.54
		6	2	79612	78398	-3.79	183.43	16.36	93.92	751.33
		7	1	79476	45429	-5.31	143.38	13.95	73.41	587.27
	256	1	14	172923	380475	-2.70	228.90	65.53	58.60	937.59
		2	7	170927	229670	-4.81	154.40	51.81	39.53	632.42
		3	4	174724	154347	-4.18	171.04	29.23	43.79	700.57
		4	4	172916	155476	-3.17	206.97	24.16	52.98	847.74
		5	3	182177	150602	-5.84	133.21	30.03	34.10	545.65
		6	2	179797	145789	-5.18	145.99	20.55	37.37	597.99
		7	2	174749	145038	-6.81	117.97	25.43	30.20	483.21
		8	2	176048	144270	-5.79	134.14	22.36	34.34	549.45
		1	14	339176	754511	-4.07	174.23	86.10	89.20	1427.26
		2	7	334309	452424	-3.19	206.11	38.81	105.53	1688.49
		3	4	341205	304413	-5.58	138.03	36.22	70.67	1130.76
		4	4	334730	304384	-4.96	150.97	33.12	77.30	1236.78
		5	3	362053	295889	-10.03	85.53	46.77	43.79	700.67
		6	2	340507	287664	-8.16	101.73	29.49	52.09	833.40
		7	2	351596	287648	-12.31	71.56	41.92	36.64	586.25
		8	2	339829	287632	-8.45	98.87	30.34	50.62	809.91
	512	1	14	339176	754511	-4.07	174.23	86.10	89.20	1427.26
		2	7	334309	452424	-3.19	206.11	38.81	105.53	1688.49
		3	4	341205	304413	-5.58	138.03	36.22	70.67	1130.76
		4	4	334730	304384	-4.96	150.97	33.12	77.30	1236.78
		5	3	362053	295889	-10.03	85.53	46.77	43.79	700.67
		6	2	340507	287664	-8.16	101.73	29.49	52.09	833.40
		7	2	351596	287648	-12.31	71.56	41.92	36.64	586.25
		8	2	339829	287632	-8.45	98.87	30.34	50.62	809.91
		1	20	796403	1718863	-	-	-	-	-
		2	10	921362	1129280	-	-	-	-	-
		3	7	881187	848720	-19.08	48.21	165.95	12.34	394.92
		4	5	827109	577687	-	-	-	-	-
		5	4	795848	569310	-24.2	38.66	129.33	9.90	316.70
		6	3	927131	559998	-	-	-	-	-
		7	3	799999	559967	-	-	-	-	-
		8	3	840207	559936	-	-	-	-	-
		1	20	1582246	3426381	-	-	-	-	-
		2	10	1823635	2251584	-	-	-	-	-
		3	7	1831691	1692494	-	-	-	-	-
		4	5	1684437	1150937	-	-	-	-	-
		5	4	1578126	1134558	-	-	-	-	-
		6	3	1839919	1117059	-	-	-	-	-
		7	3	1589338	1117030	-	-	-	-	-
		8	3	1688442	1116992	-	-	-	-	-

tations used special Vivado directives (such as AlternateRoutability, ExploreSequentialArea, SSI_SpreadLogic_high, SSI_SpreadSSLs or AggressiveExplore) to overcome some routing com-

plications from using multiple Super Logic Regions (SLRs). Although modular, the switch design requires frequent communication between modules. This becomes less of a problem when most logic fits inside a single SLR, such as for $P \leq 16$ on U280.

The rest of the data points used the default Vivado directives without any aggressive optimisation, with the exception of 512-bit switch designs for $P=8/S=2$ and $P=16/S=2$ and $S=6$, that were initially outliers, possibly relating to the performance of the tool heuristics in place-and-route.

With respect to the switch configuration, one observation is that there are different values of S for the same P , that map to the same $latency_{opt}$. This correctly follows the respective formula. However, those designs with the same number of pipeline registers are not equivalent, hence the marginal differences in the resource utilisation and other metrics. This is because their pipeline registers can be in different placements, due to the different compulsory register position. In contrast, when S is sufficiently large, the $latency_{opt}$ becomes equal to 1, in which case the only register remaining is the compulsory register. In the latter case, the designs with a single pipeline registers are equivalent, as the compulsory register position is a function of P . For this reason, the redundant data points have been removed from the table, hence the points for $P=2$ stopping at $S=3$, for example. The design space originally included S values ranging from 1 to 8.

Appendix C

Exploring a possible future path for CPUs to accelerate big data analytics

Modern general purpose CPUs support a variety of single-instruction-multiple data (SIMD) instructions [109] as a way to accelerate applications that exhibit high data-level parallelism. Although there can be notable performance improvement in certain applications [5], the instruction set can become bloated with overspecialised instructions [180], and sometimes it is still difficult to express efficiently a parallel task using a fixed set of SIMD intrinsics [181].

As an alternative acceleration platform, FPGAs can achieve impressive processing performance in specialised tasks [11], [182]. They have been getting attention for use in the data center, with numerous industrial and academic solutions focusing on processing big data [25], [26], [41]. However, the combination of specialisation and their placement in highly heterogeneous systems has considerable development and deployment costs.

In combination with the openness of the RISC-V instruction set, and its support for custom instructions [183], this is a great time to start considering custom SIMD instructions on general purpose CPUs. Small FPGAs can be integrated, to implement custom instructions [180] and are demonstrated to improve the performance over existing extensions significantly [184], [185]. In the literature, the exploration of custom instructions on CPUs, and specifically SIMD, is rather limited, even though small reconfigurable regions working as instructions is a promising

idea and possibly the future of acceleration.

This appendix presents Simodense, an open-source RISC-V softcore that is optimised for exploring custom SIMD instructions. In order to achieve high throughput for streaming applications, the focus was given on the cache hierarchy and communication. The framework allows easy integration of custom vector instructions and evaluation in simulation and hardware. A number of microarchitectural features, such as the vector register width, can be conveniently set by parameters before implementation. Finally, examples of custom instructions are evaluated and utilised for providing insights on introducing small FPGAs as execution units in CPUs. The exploration was done on a Zynq UltraScale+ FPGA, where the FPGA with the softcore shares the same DDR4 memory with the embedded cores, allowing a direct comparison.

With respect to the contributions of this project, an open-source [186] softcore framework is provided for evaluating novel SIMD custom instructions, as well as design choices for maximising streaming performance (sections C.1, C.3.1). A set of non-standard instruction types is presented to enable optional access to a high number of registers for expressing complex tasks with fewer instructions (section C.2.1). A clean Verilog template for custom acceleration as an SIMD ISA extension is proposed for use in such a lightweight softcore (section C.2.2). Last, a demonstration of the approach with novel SIMD instructions is provided (section C.3.3), with promising performance gains for memory-intensive applications.

This appendix is based on the paper “Simodense: a RISC-V softcore optimised for exploring custom SIMD instructions” [45], (and its workshop version [46]).

C.1 Simodense microarchitecture

The proposed softcore supports the RISC-V 32-bit base integer instruction set (RV32I v. 2.1), plus the “M” extension for integer multiplication and division [183]. The novel features of the proposed approach include a series of design choices to: (1) enable high-performance for custom vector instructions and streaming applications; (2) allow efficient implementation on

recent FPGAs by enhancing the block RAM (BRAM) organisation and the behaviour of the inter-chip communication.

C.1.1 Cache hierarchy optimisations

On the first level, there is an instruction cache (IL1) and a data cache (DL1), and on the second level there is a unified last-level cache (LLC). LLC responds to requests from both IL1 and DL1. It communicates in bursts to DRAM through an interconnect such as AXI. It resembles a modified Harvard architecture, as the address space is common between data and instructions. Figure C.1 provides a high-level example for the data communication throughout the cache hierarchy in the evaluation platform (Ultra96).

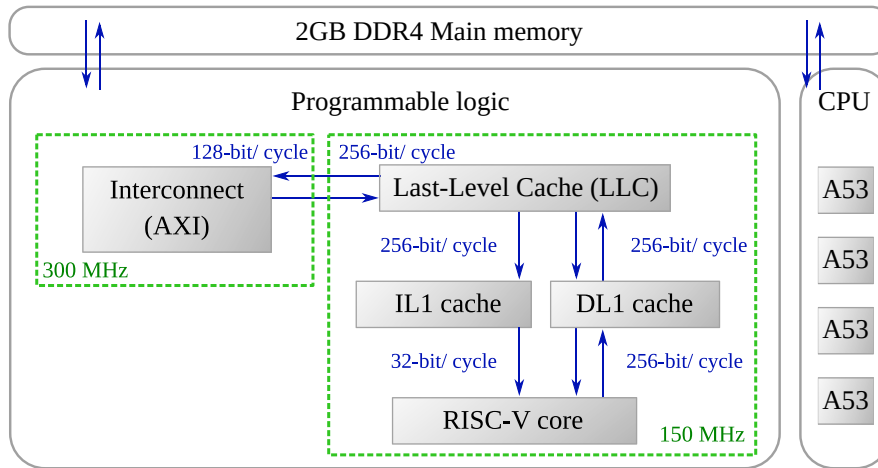


Figure C.1: Data movement in an example configuration

All caches use the writeback policy, with the exception of the IL1 cache, where writing is not needed. This is achieved by storing a dirty bit alongside each stored block, to acknowledge modification. While DL1 and LLC are set-associative caches, the IL1 is direct-mapped for fast lookup of the next instruction, to avoid a stall on instruction hits.

The LLC is implemented in block memory (BRAM), to accommodate the high capacity of the last level. The IL1 is implemented in registers for a reduced latency, in order to be able to provide the successor instruction immediately on the next cycle, if it is a hit. The DL1 is

implemented in BRAM by default, although changing the implementation directive to registers yields similar performance and utilisation results, due to its relatively low size.

At the set-associative caches (DL1 and LLC), each block can be allocated into multiple possible ways, represented by different parallel block RAM sections. The block replacement policy for these caches is not-recently-used (NRU). It uses one bit of meta-information per block [187], but closely resembles the Least-Recently-Used (LRU) eviction policy. In some cases, it is also known to surpass the performance of LRU, and is also equivalent to other approximations of LRU when implemented with a single bit per block, including SRRIP [16]. The choice of a replacement policy can be crucial to the performance of streaming applications, due to the wide data blocks and the reduced cache space on the FPGA. For instance, a random policy would stagnate the bandwidth for memory copying (*memcpy()*), when the source and destination are aligned.

A series of design choices are presented for optimising the performance and applicability for streaming SIMD applications and FPGA implementation.

Level-1 block size

One optimisation is to set the block size of the DL1 to be equal to the **vector register width**, such as 256 bits. This is because a wider block size would require an additional read on each write, from the cache of the higher level, so that the entire block becomes valid. When the data are from vector registers and are properly aligned, there is no need to wait for fetching that block on a write miss, because the whole block will contain new information.

The IL1 uses the same block size for easier arbitration between DL1 blocks, at the cache of the higher-level (LLC). Additionally, since IL1 is direct-mapped, using a wider-block than 32-bits is also beneficial to performance, as it can also be seen as a natural way of prefetching.

LLC block size

An important feature for increased streaming performance is to use **very wide blocks** for the last-level cache, such as 8192-bit wide. This is much higher than in today’s CPUs with 512-bit (64-byte) block sizes. The idea is that on write-back to/ fetch from main memory, a larger block achieves a higher speed because of longer bursts through the interconnect. Longer bursts are shown to have a significant impact on the overall throughput, such as in heterogeneous systems with AXI [77], and this is especially helpful in streaming application performance.

Associating entire LLC blocks with bursts was a convenient and practical organisation choice, because of the interconnect protocol limitations, such as for not allowing crossing of the 4KB address boundary in AXI [172].

LLC strobe functionality

A naive implementation of the LLC in BRAM, would be to read the (very wide) blocks in their entirety in a single cycle, as in the DL1 case. However, BRAM is organised in chunks of certain width and length, such as 36-bit wide. With a LLC of just a single wide-enough block, the BRAM capacity of the FPGA can be exceeded, or stagnate timing performance. For this reason, the proposed LLC stores blocks in consecutive BRAM **locations of narrower size**. There is an internal notion of sets that corresponds to the address of the block memory, where each requested data can reside. The tag array only stores the tags of entire blocks. This can be generalised for other memory technologies as well, including ultra RAM.

There is no overhead in access latency by using sub-blocks, as it still takes a single cycle to read an I/DL1-sized block from LLC. Another advantage of this technique is that, on fetch, the requested I/DL1 block can be provided before a read burst from DRAM finishes, since the LLC blocks are stored progressively. When the requested sub-block is closer to the start of the block/burst, there is a resemblance to the “critical-word first” technique. Figure C.2 visualises the indexing and strobe action of the LLC cache, where M is the number of ways, N is the number of sub-blocks in BRAM and B is the number of sub-blocks.

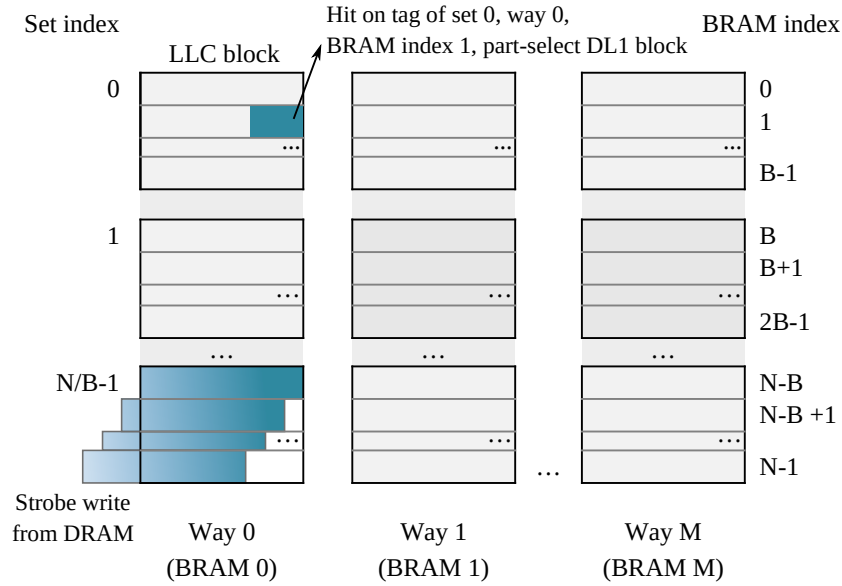


Figure C.2: Last-level cache block RAM organisation

Flushing writes

One consideration is that it is desirable to be able to observe all memory modifications Simodense makes, at least when used as a co-processor. Thus, it also features a recursive *flush-writes* command. Upon request, the DL1 goes through all stored blocks one-by-one and writes back all dirty blocks to the upper level. The changes are propagated recursively to DRAM starting from DL1. Then LLC repeats the equivalent process until all dirty bits are set to 0. This essentially “cleans” all blocks upon request, one-by-one without erasing the caches, and all writes propagate to DRAM.

Doubling the frequency of the interconnect

In contrast to the timing characteristics of Simodense, as well with other well-known softcores [188], the operating frequency of the interconnect on FPGAs can be relatively much higher [189]. Given that the port data widths are rather narrow (e.g. 128 bits/ cycle on Ultra96), this directly impacts the throughput for streaming applications. This optional optimisation involves setting double rate for the interconnect, to **emulate double data width** by fetching or writing twice per cycle. Figure C.1 illustrates this change on the data movement diagram in high-level.

It can be considered a platform-specific optimisation, since the complexity of having the interconnect operating at double the clock speed could vary according to the FPGA model. Moreover, the timing characteristics of future FPGA generations could shift to favour the soft-core implementation, for example. When applicable, it can saturate [77] the bandwidth more easily.

C.1.2 Main core

The core has a single pipeline stage, even though more advanced instructions such as pipelined vector instructions have their own pipeline. Almost all instructions in RV32I consume 1 cycle and the result is available on the immediately next cycle. In practice, this has a similar effect to operand forwarding in pipelined processors, as consecutive dependent instructions are executed sequentially without stalls.

The load and store instructions are handled by the cache system independently. On a data cache hit there is a latency of 3 cycles until the dependent command gets executed. The 3 cycles can be seen as a small pipeline with one cycle for memory access, one for fetching the data and one for updating the registers. This effectively yields a latency of 2 cycles for cache hits, when the next instruction is data-dependent on the load, as execution is in-order.

Having a **single pipeline stage**, so that most instructions complete in a single cycle, is useful for simplifying the dependency checks. The output of simple instructions such as add, addi, etc. is not tracked for dependencies. Of course, there are alternative approaches, but the current implementation mapped well in the evaluation platform and facilitated the SIMD functionality rather efficiently.

In order to monitor the data dependencies in multi-cycle instructions such as the “M” extension, SIMD tasks and loads, there is a ready bit per register that influences the stall decisions. The “M” extension is implemented behaviourally in Verilog, but consumes a few cycles more, to enable Vivado to use DSPs efficiently and ease timing closure.

The implementations of the SIMD instructions follow a template (discussed in section C.2),

that allows a variable pipeline length, abstracted through a *ready* signal for when the result is available. Apart from the 32 base 32-bit registers (as per RV32I), there are up to 8 VLEN-wide registers, such as 256-bit-wide for the SIMD instructions.

C.2 Custom SIMD instructions

In order to support adding and using SIMD instructions on Simodense, this appendix presents (A) new instruction types that refer to the vector registers and (B) HDL (Verilog) code templates for implementing the instructions in hardware.

C.2.1 Vector instruction types

In addition to the RV32IM standards two additional non-standard instruction types are proposed for supporting the custom SIMD instructions. Originally there are 4 main instruction types: R, I, S/B, U/J in RV32I, that define the format of the immediate field and the operands. The proposed instruction types I' and S' are variations of the I and S types respectively.

The general idea is that the space used for the immediate field is repurposed for the vector register names. **Up to 6 registers** (vector and non-vector) can be accessed by the same instruction, reducing the instruction count for complex or stateful (through registers) instructions. The use of the immediate field for the vector registers was also convenient for minimal interference and modification to the RISC-V GNU toolchain in GCC's binutils for supporting inline assembly of custom instructions. There is currently no official tool support for vector registers due to the draft status of the vector extension. Thus, the opcodes dedicated to custom applications are adopted for all custom vector instructions, and can be used through inline assembly.

Type I' provides access to the register operands of the I-type, that is one source and one destination 32-bit register. It also provides access to two source (*vrs1* and *vrs2*) and two destination (*vrđ1* and *vrđ2*) vector registers. Type S' exchanges the space used by *vrs2* and *vrđ2*

for access to an additional 32-bit source register *rs2*. The proposed variations are summarised in Figure C.3.

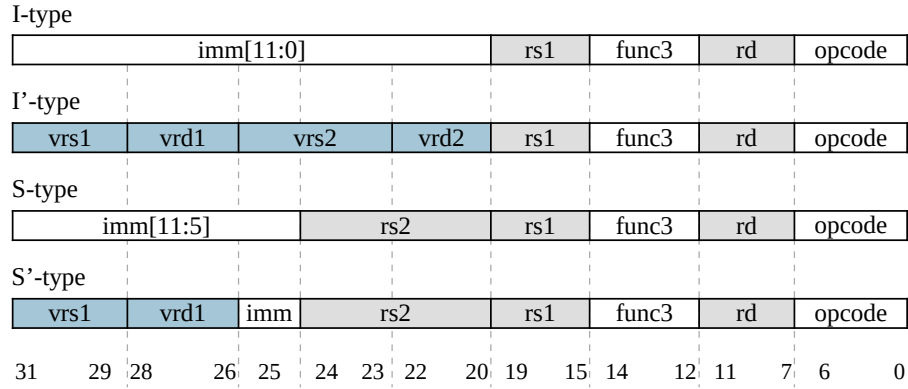


Figure C.3: Two variations of the I and S instruction types

As shown in Figure C.3, the fields for each vector register are three bits wide, which sets the maximum number of vector registers to 8. Vector register 0 is conveniently assigned to a **constant value of 0**, similarly to the 32-bit base registers. It is useful for the proposed many-register instruction types, because not all register operands may need to be accessed at once, allowing different combinations of operands using the same type. In software, this can be achieved with aliasing the unused operands with register 0, and was **not** supported in “V”, as vector 0 originally represents a register.

The official (draft) RISC-V “V” vector extension has not been followed in this work, as it seems to target high-end/ hardened processors. For example, it requires a fixed number of 32 vector registers and features hundreds of instructions [6], also reducing the number of registers an instruction can access. This would also be contrary to the idea of having small reconfigurable regions as instructions, rather than supporting hundreds of intrinsics. An interesting feature in the “V” specification is the ability to chain vector instructions, hence the need for a high number of registers. Since the proposed solution enables custom instruction **pipelines of arbitrary length**, a lower number of registers was considered satisfactory.

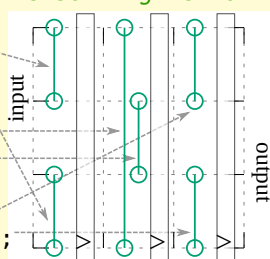
C.2.2 Instruction templates

The custom instruction templates are placeholder modules inside Simodense’s codebase, for adding user code for specialised SIMD instruction implementations. Algorithm 11 is written in Verilog and shows part of the template for I'-type instructions, plus an example user code, marked in yellow. On each cycle, the instruction module also accepts the destination register names (*rd*, *vr1* and *vr2*) to provide them later, when the data result is ready, after the specified pipeline length (*c1_cycles*). In this way, with a pipelined implementation the module can process **multiple calls** one after another. Blocking instructions are also supported with minor modification.

```

1  `define XLEN 32 // 32-bit base registers
2  `define VLEN 128 // 128-bit vector registers
3  `define c1_cycles 3 // pipeline length of c1 custom instruction
4
5  module c1 [...]
6      input clk, reset, in_valid; // valid bit for input
7
8      // Destination register names (1 base and 2 vector)
9      input [4:0] rd; input [2:0] vr1, vr2;
10
11     input [`XLEN-1:0] in_data; // 32-bit input
12     input [`VLEN-1:0] in_vdata1, in_vdata2; // 128-bit input
13
14     // (Delayed) output valid bit and output register names
15     output out_v;
16     output [4:0] out_rd; output [2:0] out_vr1, out_vr2;
17
18     output [`XLEN-1:0] out_data; // 32-bit output
19     output [`VLEN-1:0] out_vdata1, out_vdata2; // 128-bit output
20
21     // Shift register logic to delay rd, vr1, vr2 and in_valid
22     // by c1_cycles, to out_rd, out_vr1, out_vr2 and out_v resp.
23     [...]
24     /// User code ///
25     wire [31:0] net [15:0]; // 32-bit wires inside the sorting network
26     // On pos. edge: a, b -> min(a,b), max(a,b)
27     CAS cas0(clk, net[0], net[1], net[4], net[5]);
28     CAS cas1(clk, net[2], net[3], net[6], net[7]);
29
30     CAS cas2(clk, net[4], net[7], net[8], net[11]);
31     CAS cas3(clk, net[5], net[6], net[9], net[10]);
32
33     CAS cas4(clk, net[8], net[9], net[12], net[13]);
34     CAS cas5(clk, net[10], net[11], net[14], net[15]);
35
36     // Assigning input and output to wires in the sorting network
37     // (only using 1 input and 1 output reg. for this instruction)
38     for (i=0; i<4; i=i+1) assign net[i]=in_vdata1[32*(i+1)-1:-:32];
39     assign out_vdata1={net[12], net[13], net[14], net[15]};
40
41     endmodule

```



Algorithm 11: Verilog template for I'-type, with user-provided code highlighted in yellow

The example instruction implementation in Algorithm 11 is a bitonic sorter of 4 inputs. Such sorting networks are parallel and pipelinable algorithms for sorting a list of N values. (See section 2.3.1 for more information on sorting networks). For a vector register width of 128-bit, this bitonic sorter sorts four 32-bit values in 3 cycles. A demo abstract [190] elaborates on this example.

The template of the S'-type instructions is similar to Algorithm 11, but with an interface that reflects the correct operands (2 base and 1 vector registers for input and 1 base and 1 vector for output). One S' type instruction for loading and storing VLEN-sized vectors (*c0_lw* and *c0_sv* respectively) is provided by default.

C.3 Evaluation

The exploration is divided in three parts according to the outcomes of each set of experiments: (C.3.1) justifies important design choices related to streaming performance, (C.3.2) shows that the performance is still acceptable when no SIMD instruction is used and (C.3.3) explores the behaviour and efficiency of example novel custom SIMD instructions.

The evaluation platform is Ultra96, which features the Xilinx UltraScale+ ZU3EG device. The FPGA on the device shares the same 2GB DDR4 main memory with the 4 ARM cores. The ARM cores run Linux, but the kernel address space is manually configured to end at the 1GB mark, so that the other 1GB is dedicated to the FPGA, that includes Simodense. This softcore is controlled through the Linux environment for quick deployment and comparison with alternative solutions. All source code is available [186].

C.3.1 Design space exploration

The target application is memory copying (*memcpy()*), as its performance is (indirectly) detrimental to big data processing and related evaluations have a long history in HPC applications [191]. *memcpy()* here is manually implemented with the custom instructions for load vector

and store vector, instead of a library implementation using base registers. The data length is 256 MiB, in order to surpass the cache sizes.

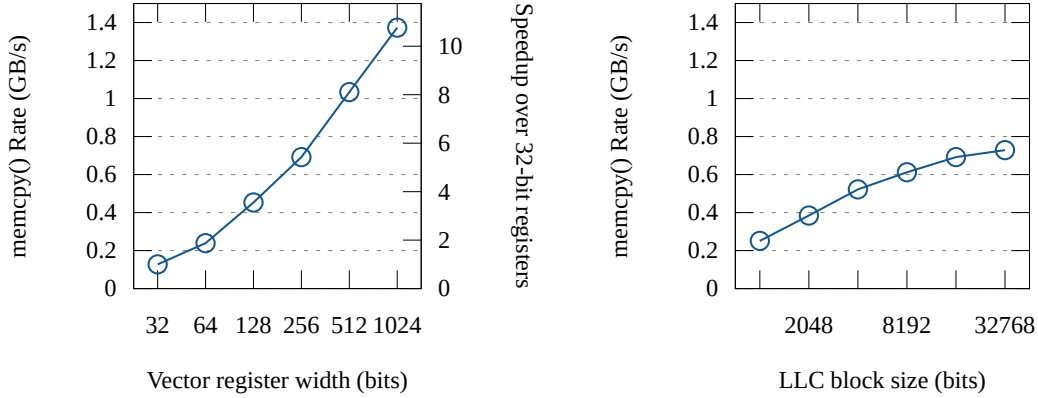


Figure C.4: Memcpy() read and write (bidirectional) throughput for different vector register widths (left) and last-level cache block sizes (right)

Figure C.4 (left) illustrates the impact of the vector register size on *memcpy()*. The 1024-bit Simodense achieved a *memcpy()* rate of 1.37 GB/s. Though, 256-bit (VLEN) registers are preferred, with a rate of 0.69 GB/s, as 512-bit and beyond seemed more challenging to route efficiently when incorporating more complex custom instructions. These designs used a 16384-bit-wide LLC block.

Another experiment (Figure C.4 right) measures the impact of the block size in last-level cache (LLC). Wider LLC blocks seem to be a considerable contributor to memory performance, as they relate to the burst size. The improvement starts to plateau after longer bursts at around 8192 bits. All implementations reached timing closure for a frequency 150 MHz, except the 1024-bit configuration that was clocked at 125 MHz. Table C.1 summarises the selected baseline configuration for the remainder of the evaluation.

Table C.1: Selected configuration of Simodense

IL1		DL1			LLC				VLEN (bits)	f_{max} (MHz)
sets	block (bits)	sets	ways	block (bits)	sets	ways	block (bits)	sub-blocks		
64	256 (=2KiB)	32	4 (=4KiB)	256	32	4	16384 (=256KiB)	32	256	150

C.3.2 Performance as a RV32IM core

In order to show that there is no significant bottleneck when compared with other non-SIMD cores, some other works with a similar specification are overviewed. Table C.2 presents common benchmark metrics alongside previously reported numbers using FPGAs. Note that this is not for direct comparison, as each work used a different FPGA architecture, core/cache configuration and compilation environment.

Table C.2: Indicative comparison ignoring SIMD

	DMIPS/MHz	Coremark/MHz	f_{max}	FPGA architecture
RVCoreP/radix-4[178]	1.25	1.69	169	Xilinx Artix-7
RVCoreP/DSP[178]	1.4	2.33	169	Xilinx Artix-7
PicoRV32[192]	0.52	<i>N/A</i>	<i>N/A</i>	(simulation)
RSD/hdiv[193]	2.04	<i>N/A</i>	95	Zynq
BOOM/hdiv [193], [194]	1.06	<i>N/A</i>	76	Zynq
Taiga[188], [195]	>1	2.53	~200	Xilinx Virtex-7
<i>This work</i>	<i>1.47</i>	<i>2.26</i>	<i>150</i>	<i>Zynq UltraScale+</i>

Additionally, the performance of the proposed approach is measured for memory-intensive situations, without the use of SIMD. STREAM [191] is an established benchmark suite measuring memory performance, especially in HPC. Figure C.5 shows the obtained throughput in MB/s for each of the 4 kernels.

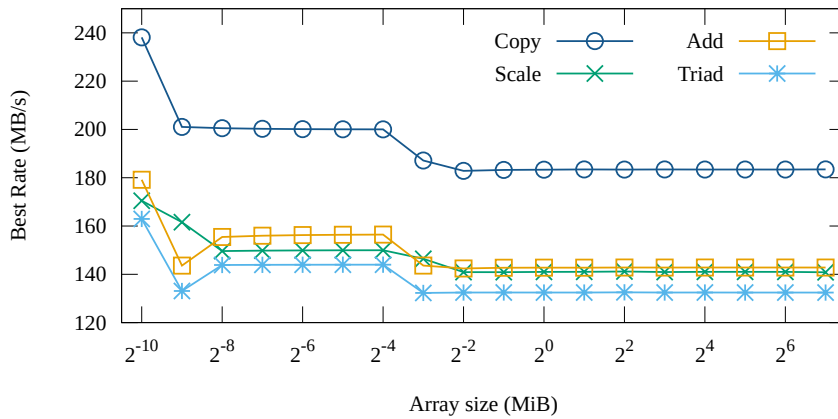


Figure C.5: Adapted STREAM benchmark results, no SIMD

On the same FPGA, a PicoRV32 [192] is placed as a drop-in replacement that supports AXI (Lite). Although it was not designed for performance, it achieves high operating frequencies

(300 MHz on the test platform), partly mitigating for its low IPC [188]. It does not have a cache, although this does not directly impact memory bandwidth, as the data reuse is practically zero. (The steps in Figure C.5 are from reusing data from the initialisation). The results of PicoRV32 were 4.8, 3.6, 4.4 and 4 MB/s for Copy, Scale, Add and Triad consistently across the array size range. This makes the approach **38x** faster for Copy at 183.4 MB/s, or **144x** faster if the 256-bit *memcpy()* performance is considered. This also highlights the importance of optimising communication for streaming applications.

C.3.3 Custom SIMD instruction use cases

Sorting (32-bit integers)

Sorting is a widely applicable big data application. Existing SIMD intrinsic solutions are based on algorithms such as sorting networks [61], radix sort [196], mergesort [181], quicksort [5] and combinations.

The presented approach follows merge sort, with the help of sorting networks for introducing parallelism. Sorting networks were adapted for both software [5], [181] and hardware [32], [69], [70], [197] solutions for sorting arbitrarily long input.

In order to accelerate sorting in the softcore for arbitrary-sized input, a sorting network is used first to sort the entire list first in small chunks, as in [5]. Then, a traditional recursive merge sort approach is performed, but instead of merging each two sublists by comparing one element by one, it uses a parallel merge block. The merge block (the last $\log_2(N)$ layers of odd-even mergesort) is to merge two already-sorted lists together, as demonstrated in a numerical example in figure C.6. In the implementation, one more stage is added to the start of the pipeline to enable merging arbitrarily long lists progressively, and the algorithm is inline with the intrinsics merge algorithm [181].

For brevity, this section only elaborates on the sort-in-chunks loop. Figure C.7 illustrates the instruction start and end times for this loop during a simulated run. From this figure the

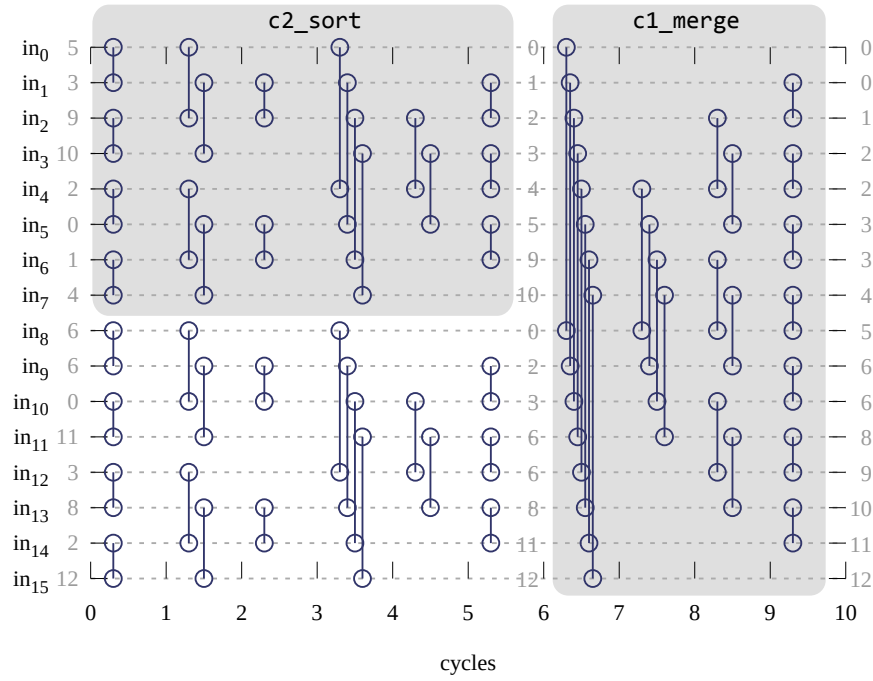


Figure C.6: Two new custom instructions based on odd-even mergesort

pipelining effect can be observed, as two instances of *c2_sort* take place simultaneously, to sort two octuples. The second sort is shifted by two cycles, as it still waits for its operand *v2* from the second load. Then, the merge instruction (*c1_merge*) merges the registers *v1* and *v2* and stores the upper and lower half back to *v1* and *v2* respectively, for sorted chunks of size 16. Another observation is the presence of instructions that take a single-cycle, such as *bne* and *addi*.

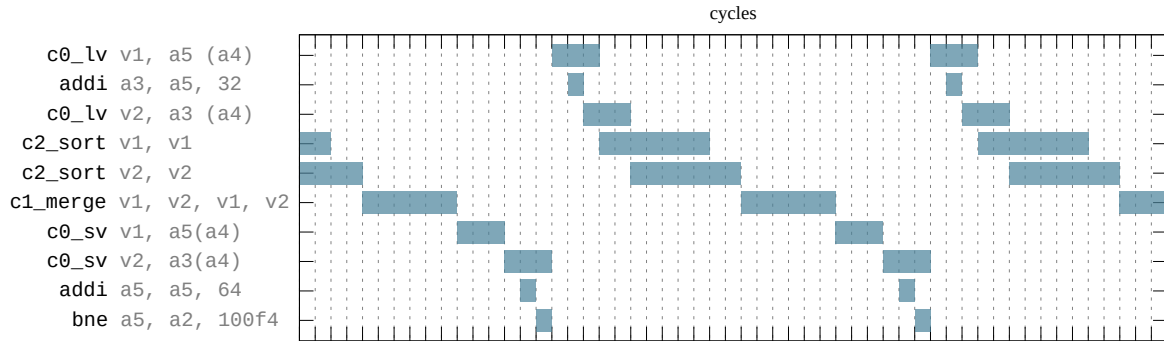


Figure C.7: Instruction execution times from the loop for sorting in chunks

The performance of the resulting mergesort function is compared against non-vectorised code on the softcore, running at 150 MHz, as well as on the ARM A53 core, running at 1.2 GHz. The baseline is *qsort()* from C's standard library. The obtained speedup is **12.1x** and 1.8 times over the *qsort()* on the softcore and ARM respectively, for 64 MiB random input. Comparison

with more optimised code such as multi-threaded NEON-based for ARM, as well as other SIMD algorithms [5] would also be possible.

One of the most useful insights from such exploration is about the reduction in the number of instructions and cycles required for a task. For instance, the *c2_sort* instruction is able to sort a list of 8 32-bit elements in 6 cycles. In contrast, a sorting network implementation of only 4 32-bit inputs in older Intel processors required 13 SIMD instructions and 26 cycles [181]. This **13x** and **4.3x** reduction of instructions and cycles respectively, while solving a bigger problem, is due to the unavailability of such specialised instructions. Even with the latest AVX-512 intrinsics [109], for each layer of compare-and-swap (CAS) units, a pair of separate instructions *min* and *max* are required, as well as a few calls of *shuffle* that permute the inputs for correct alignment [181]. A similar discussion can be made for prefix sum [38], though parallel prefix sum uses more comparisons than the serial case, hence the less notable speedups.

Note that a FLiMS-based approach (chapter 3) would have achieved a lower resource utilisation for the sorting use case, but in terms of performance there would not be a significant difference, because the amount of pipelining in this implementation is not that advanced, to benefit from the removal of the feedback. FLiMS however results in lower instruction count in AVX2 and AVX-512 implementations (see section 3.6 and appendix B) in x86 processors, because the reduced resource utilisation over different merging algorithms (see section 3.4) translates into a reduced instruction count when using a predefined instruction set. This fact can be considered one more advantage of the reconfigurability in instructions. Still, advanced pipelining of the instructions in the core architecture would be an appropriate future work, to benefit from such lower-level optimisations as FLiMS.

Prefix sum

Another fundamental operator is prefix sum, and has numerous applications in databases, including in radix hash joins and parallel filtering [38]. The prefix sum for a series of values is the cumulative sum up to each value inclusive, (i.e. $out_k = \sum_{i=0}^k in_i$ for $k \in \{0, 1, \dots, N-1\}$), where N is the number of inputs. The serial implementation of prefix sum is trivial and easy for

compiling efficient code. Each element is read one by one and is added to a counter initialised with 0. On every read, the value of the counter is written back as output for the corresponding position.

Data parallelism in prefix sums is mainly exploited either horizontally or vertically. Here, the vertical computation is selected, i.e. to exploit data-level parallelism in a single pass of the data without partitioning or multi-tasking, similarly to the sorting in chunks sub-task. A widely-used algorithm for parallelising prefix sum is from Hillis and Steele [198], and is used in recent SIMD implementations [38], that consumes $\log_2 N$ parallel steps. This algorithm is for a fixed input size, but it can be applied to arbitrarily long input by reading the input in increments and adding the sum of the preceding batch, as with existing SIMD implementations [38].

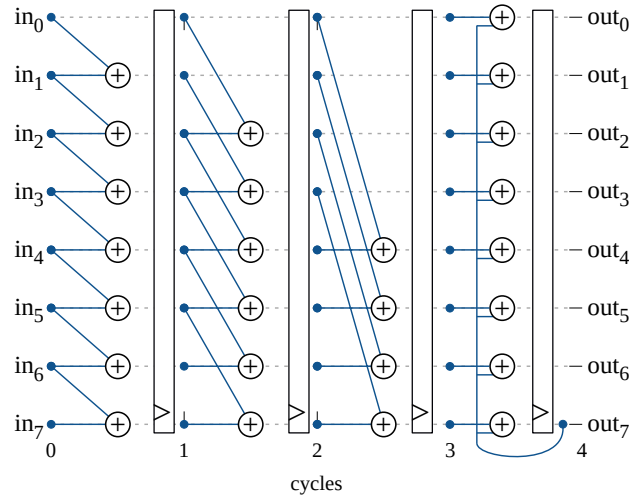


Figure C.8: New SIMD instruction for prefix sum

Figure C.8 presents the proposed custom instruction for the task. The first $\log N$ steps contain a pipelined version of the Hillis and Steele [198] algorithm, plus one additional stage that adds the cumulative sum of the previous batch, that also happens to be the cumulative sum of the entire input up to that batch. In this way, it can calculate the prefix sum of an arbitrarily long input in a pipelined and non-blocking way. This design is also appropriate for stand-alone accelerators. For 64 MiB input, vectorising prefix sum yielded a speedup of **4.1x** over the serial version, though it had 0.4x the speed of an ARM A53 core.

C.4 Comparison to related work

As this appendix concerns a framework with a unique purpose, there are no direct competitors. However, there are alternative works that could be used instead, but with a need for significant alterations to their code-base (if available), to support the desired features and performance for exploring SIMD instructions.

Nguyen Dao et al. presented FlexBex [180], an open-source framework for embedding small FPGAs in a modified Ibex RISC-V core [199]. Although it provides a form of SIMD functionality, it is for embedded solutions and without a cache, and the operation is done on multiple 32-bit registers. This makes it limiting for benefiting memory-intensive applications, which is the target of the thesis. On a similar note, Ordaz et al. [185] developed a closed-source 128-bit wide SIMD engine shared between RISC-V cores [195]. Unlike the proposed softcore, the memory interface was much narrower than the SIMD engine, and both of these works mostly focus on the fabrication aspect rather than instruction development and streaming performance. Moreover, the presented softcore is demonstrated on memory-intensive applications instead, and for wider vector widths.

There is a plethora of open-source softcores implementing the RV32IM or similar instruction set [188], from both industry and academia. Many have special features, such as out-of-order execution [193], [194], running Linux [200] and optimised arithmetic-logic units [178], [201]. Though, fundamental limitations, such as a 32-bit memory interface, are usually hard-coded and prevent efficiently incorporating vector instructions.

There are softcores that may have the potential to be adopted for exploring custom SIMD instructions [202], [203], but one major consideration is their use of higher-level languages and generators, which are known to be a problem for adoption [178]. Novel modifications can be incompatible and require considerable effort altering the framework. One such candidate is the latest variant of Berkeley's BOOM [204] that still does not support SIMD instructions. In this thesis, the framework is purposely written in Verilog with SIMD in mind, as well as for the templates, to enable low-level customisation.

There are also numerous FPGA-based vector processors and overlays, that could potentially perform better but are not suitable for exploring arbitrary SIMD instructions for CPUs. For example, Cheung et al. introduced a simple RISC-V extension for coupling FPGA-based accelerators, but the new instructions are only for passing and retrieving exclusive control to a dedicated accelerator logic [205]. Other works elaborated on a fixed vector instruction set [206]–[210] but not for adding custom instructions, and of a custom pipeline complexity. There are also frameworks that eliminate the base instruction set [211] in favour of only executing a specialised task. Additionally, there are works on accelerator generators that are based on SIMD instructions, but there is a complete absence of instructions in the implementation itself [92].

C.5 Summary

This open-source [186] softcore, in combination with the proposed optimisations and instruction templates can be used to explore novel high-performance SIMD instructions. The provided instruction templates and types provide the ability to develop advanced SIMD instructions with a few lines of code, and minimise the instruction count for increased performance. It is demonstrated that custom SIMD instructions can provide an order of magnitude of speedup over serial implementations for memory-intensive applications. Sometimes, the custom SIMD instructions can approach the performance of the hardened cores, especially if there is more “internal” computation, whose efficiency cannot be surpassed with predefined instructions.

Simodense aims at exploration. It is shown to perform well on real hardware, and does not assume 1-cycle-latency memories for instructions or data, a convention often found in other works [188]. However, sometimes it cannot compete with dedicated FPGA accelerators, as with the sorter of section 5.1). This gap is expected and relates to the presence of instructions, such as by trying to avoid internal states. Still, further pipelining of the caches can help to a degree, especially for simpler applications like prefix sum. Non-memory-intensive tasks can still achieve higher speedups, depending more exclusively on the FPGA’s capabilities.

Holding states can be challenging, especially for higher-end CPUs. Some of the presented instructions here do hold a (minimal) state, including the cumulative sum of the last valid output for prefix sum (which resets on request (by setting $vr d1 = 0$). Though, holding a state is allowed in template 11 and the core is simple enough to not be of concern, such as with no wrong execution paths. With respect to sorting, since it is not a purely streaming task, many accelerators try to “internalise” processing to reduce the number of times the data is read [32], but not on CPUs [181]. See section 6.2.5 for a discussion on the challenges of holding states inside instructions.

The availability of small reconfigurable regions as instructions in future generations of CPUs would be a more efficient use of silicon and processing cycles, as also demonstrated through the studied use cases. This would also simplify designs and potentially solve the main memory bottleneck found in today’s FPGA-based datacenter accelerators.