DISSERTATION

submitted to the

Combined Faculty of Mathematics, Engineering and Natural Sciences

of the

Ruprecht–Karls University
Heidelberg

for the degree of

Doctor of Natural Sciences

put forward by

M.Sc. Jonas Christian Dann

born in
Wiesbaden-Dotzheim, Hessen

Heidelberg, 2023

FPGA-based Query Acceleration for Non-Relational Databases

Advisor: Professor Dr. Holger Fröning

Date of oral exam: ....

To my family.

Abstract

Database management systems are an integral part of today's everyday life. Trends like smart applications, the internet of things, and business and social networks require applications to deal efficiently with data in various data models close to the underlying domain. Therefore, non-relational database systems provide a wide variety of database models, like graphs and documents. However, current non-relational database systems face performance challenges due to the end of Dennard scaling and therefore performance scaling of CPUs. In the meanwhile, FPGAs have gained traction as accelerators for data management.

Our goal is to tackle the performance challenges of non-relational database systems with FPGA acceleration and, at the same time, address design challenges of FPGA acceleration itself. Therefore, we split this thesis up into two main lines of work: graph processing and flexible data processing.

Because of the lacking benchmark practices for graph processing accelerators, we propose GraphSim. GraphSim is able to reproduce runtimes of these accelerators based on a memory access model of the approach. Through this simulation environment, we extract three performance-critical accelerator properties: asynchronous graph processing, compressed graph data structure, and multi-channel memory. Since these accelerator properties have not been combined in one system, we propose GraphScale. GraphScale is the first scalable, asynchronous graph processing accelerator working on a compressed graph and outperforms all state-of-the-art graph processing accelerators.

Focusing on accelerator flexibility, we propose PipeJSON as the first FPGA-based JSON parser for arbitrary JSON documents. PipeJSON is able to achieve parsing at line-speed, outperforming the fastest, vectorized parsers for CPUs. Lastly, we propose the subgraph query processing accelerator GraphMatch which outperforms state-of-the-art CPU systems for subgraph query processing and is able to flexibly switch queries during runtime in a matter of clock cycles.

Zusammenfassung

Datenbanksysteme sind essenzieller Teil unseres heutigen Alltags. Trends wie Smarte Anwendungen, das Internet der Dinge, und gewerbliche und soziale Netzwerke benötigen Anwendungen, die effizient mit Daten in verschiedenen, domänenspezifischen Datenmodellen arbeiten können. Deswegen stellen nicht-relationale Datenbanksysteme eine breite Vielfalt an Datenmodellen, wie zum Beispiel Graphen und Dokumente, zur Verfügung. Jedoch stehen nicht-relationale Datenbanksysteme wegen des Endes des Dennard Scaling und damit der Performanceskalierung von CPUs vor Performanceherausforderungen. Gleichzeitig haben sich FPGAs als Beschleuniger für Datenmanagement etabliert.

Unser Ziel für diese Doktorarbeit ist es diese Performanceherausforderungen von nicht-relationalen Datenbanksystemen mit Hilfe von FPGAs als Beschleuniger zu lösen und gleichzeitig Designherausforderungen von FPGA-Beschleunigung zu adressieren. Diese Doktorarbeit ist in zwei Hauptarbeitsstränge aufgeteilt: Graphverarbeitung und flexible Datenverarbeitung.

Wegen der mangelhaften Benchmarkmethoden für Graph Processing Beschleuniger präsentieren wir GraphSim. GraphSim kann Laufzeiten von diesen Beschleunigern basierend auf einem Speicherzugriffsmodell des Ansatzes reproduzieren. Mit Hilfe dieser Simulationsumgebung haben wir drei für die Performance kritische Beschleunigereigenschaften identifiziert: Asynchrone Graphverarbeitung, eine komprimierte Graphdatenstruktur, und Multi-channel Speicher. Weil diese Beschleunigereigenschaften noch nie in einem System kombiniert wurden, präsentieren wir GraphScale. GraphScale ist der erste, skalierbare, asynchrone Graph Processing Beschleuniger der auf einer komprimierten Graphdatenstruktur arbeitet. GraphScale übertrifft die Performance aller anderen State of the Art Graph Processing Beschleuniger.

Auf die Flexibilität von Beschleunigern fokussiert, präsentieren wir PipeJSON als den ersten FPGA-basierten JSON Parser für JSON Dokumente mit beliebiger Struktur. PipeJSON kann mit Line-Speed parsen und übertrifft damit die schnellsten, vektorisierten Parser für CPUs. Zuletzt präsentieren wir den Subgraph Query Processing Beschleuniger GraphMatch, welcher alle State of the Art CPU-basierten Systeme übertrifft und den Subgraph Query flexibel innerhalb weniger Taktzyklen ändern kann.

Acknowledgements

This thesis has been an intense journey with many ups and downs that I look back onto with great pride. During this journey, I grew as a person and learned a great deal about myself, academia, and computer science. I am grateful for the many people that supported me during this very rewarding period of my life.

First and foremost, I would like to thank my thesis supervisors Prof. Dr. Holger Fröning and Dr. Daniel Ritter. Back when I was a Bachelor student, Daniel introduced me to academic research in the first place. He continued to be a great mentor during my time as a PhD student. Holger provided me with numerous opportunities to travel and connect with interesting people which massively helped me grow as a researcher. Holger's and Daniel's support and insightful feedback were invaluable to me over the years of my PhD studies.

I would like to thank Dr. Christian Färber for his support of the hardware platforms I used as a PhD student and Dr. Mehdi Moghaddamfar for his valuable advice on FPGA development whenever I hit the limits of my own knowledge on FPGAs. A special thanks also goes out to my students Royden Wagner, Jan Ahlbrecht, and Tobias Götz that I had the pleasure of collaborating with.

I would like to thank my fellow PhD students – past and present – of the Computing Systems Group at Heidelberg University and SAP HANA Research Campus. I enjoyed the time we spent together discussing ideas, sharing knowledge, giving feedback, and encouraging one another. Specifically, I would like to thank Hendrik Borras, Dr. Sven Elbert, Bernhard Klein, and Robert Lasch for their feedback on different drafts of this thesis.

Most importantly, I would like to thank my loving parents Kerstin and Michael Dann for their continued support. They both are great role models to me and have given me the confidence and rigor to pursue this thesis.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Databases are an integral part of today's everyday life and are the foundation of almost every computer-assisted transaction, like online banking, e-government, business processes, and text messaging. These databases are created, updated, and analyzed by database managements systems. Traditionally, database management systems worked on data represented as tables with a fixed schema, also called relational databases [Cod70]. More recently, trends like smart applications, the internet of things, and business and social networks require applications to deal more efficiently with data in various data models close to the underlying domain, as highlighted by recent studies [Bes+19a; DCL18]. For instance, the predominant data exchange format in distributed business applications is JavaScript Object Notation (JSON) [LL19; Dan+22] and working with JSON documents gained popularity in schema-less contexts [DCL18]. Similarly, applications like social network analysis require processing and storage capabilities of graph data, thus increasing the model variety. Traditional relational database systems, however, only hesitantly addressed these requirements of model variety (i. e., expressiveness, flexibility) [Aba12; Bre12; Cat10; Sto10]. Thus, the new class of non-relational database systems emerged to address these requirements of model variety and additionally provide important properties like scalability.

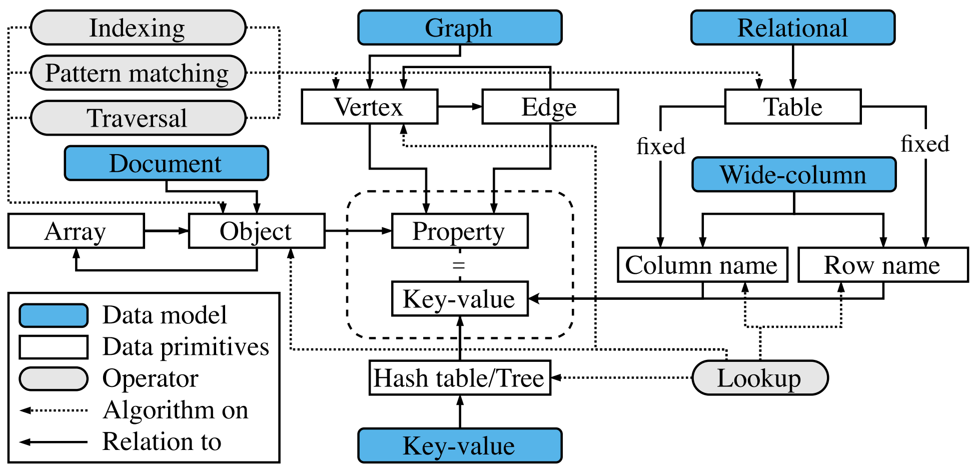


Fig. 1.1 Data models and their relation.

1.1 Non-relational Database Systems

Non-relational database systems provide a wide variety of data models, like graphs and documents, and scalable processing by relaxing traditional relational database system constraints. This allows for flexible and application-specific data modeling and processing of large data sets. In line with recent non-relational database system surveys [Bes+19a; DCL18], for this thesis, we focus on non-relational database systems for the following important data models: graph, document, key-value, and wide-column.

At first sight, non-relational database systems might seem distinct from one another, with shared properties being abstract system design principles like their application specificity and scalability. For instance, complex graph query languages are conceptually different to the straightforward APIs of key-value database systems (e. g., put and get [DeC+07]). Although we acknowledge these differences, we argue that the various data models (blue boxes) are grounded in common primitives in key-value pairs and types of operators (cf. Fig. 1.1). These operators include lookup, traversal (e. g., breadth-first search (BFS) on graphs, path traversal on document hierarchies), pattern matching (e. g., Cypher [Fra+18] on graphs, XPath on documents), and indexing (e. g., PageRank on graphs, full text document indices). These foundational primitives even align with the relational data model. Each data model is characterized by a structural layer of specific primitives (e. g., vertices and edges of a graph) that connect properties or key-value pairs. For instance, the key-value data model directly

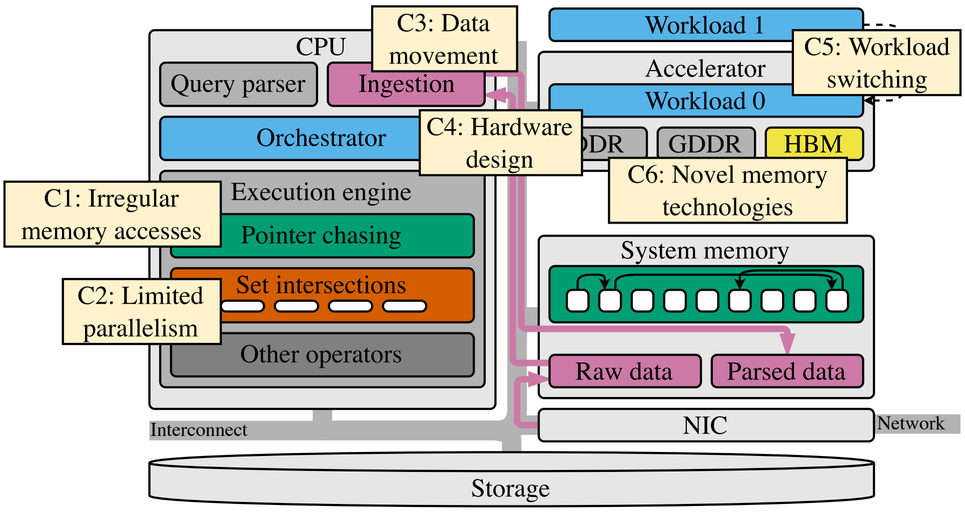


Fig. 1.2 Overview of challenges in FPGA-accelerated non-relational database systems.

stores key-value pairs in a hash table or tree (e. g., [ISA17]). In contrast, the wide-column data model employs both a column and a row name as keys to the key-value pairs (e. g., [LM10]). The relational data model is similar to the wide-column data model but uses tables with a fixed schema of column names where each row has to contain the same columns. Graph database systems use property graphs that store networks of vertices connected by edges, with properties associated with both of those [DCL18]. Documents are built from possibly nested objects and arrays that again refer to properties [DCL18]. Beyond this, all data models facilitate basic lookup functions, while the more expressive data models (graph, document, and relational) support structural traversals, pattern matching, and indexing.

1.2 Challenges and Research Gap

The data models and operations of non-relational database systems are also relevant when considering performance scaling of data processing because the resulting workloads pose unique challenges. Figure 1.2 shows an abstract common database system architecture. Besides the CPU that executes the main components of the database system, the system contains storage to persist data, a network interface controller (NIC) for communication with other nodes of the system and applications (e.g., receiving and answering queries and ingestion of

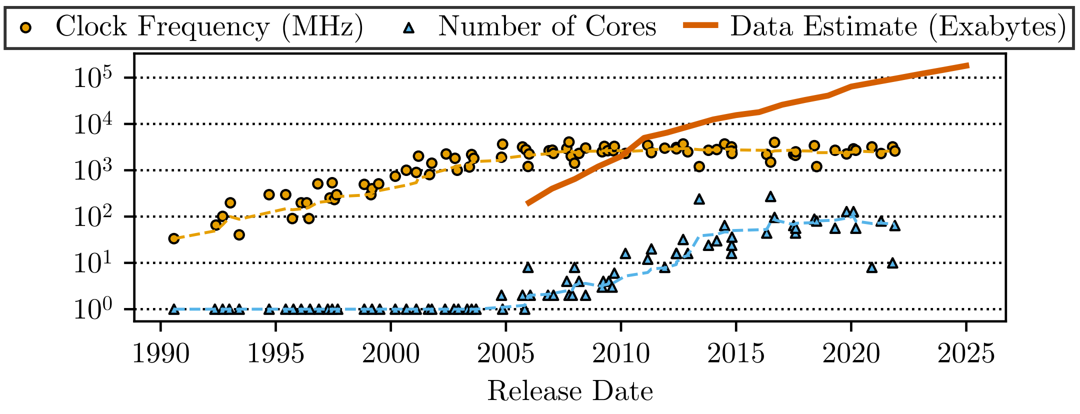


Fig. 1.3 CPU frequency and number of cores scaling compared against estimate of global data over time.

data), system memory for the working data, and, more recently, one or more accelerators. All components communicate with each other over an interconnect (e.g., PCIe). In the following, we will introduce the challenges that non-relational database systems face on traditional CPU-centered architectures (C1–3), highlighting the need for accelerators in modern database systems, and the challenges that integrating an accelerator into the system raises (C4–6). Lastly, we discuss how these challenges are addressed by current research and which research gaps remain and will be tackled with this thesis.

1.2.1 Challenges for Non-relational Databases

The volume of data handled by non-relational database systems starkly increases, for example, into gigabytes of JSON documents (e. g., [LL19; Dan+22]) and big graph datasets (e. g., up to one trillion edges [Chi+15]). Current non-relational database systems, primarily built for general-purpose CPUs (cf. Chapter 2), face challenges with processing these increasing data volumes and scaling out [DRF23b]. Since the end of Dennard scaling [Den+74], meaning transistor density not scaling at approximately constant power consumption anymore, frequency and therewith performance scaling of CPUs is reaching limits (cf. Fig. 1.3). To address this, CPU vendors have turned to optimizations like instruction pipelining, coarse-grained multi-core and hardware thread parallelism, and vectorization. However, even with these optimizations, CPUs performance is constrained as can be seen from the development of the number of cores of current CPUs [TS21]. In this thesis, our focus is on the broader problem of stagnating performance growth in non-relational database systems that rely on traditional CPU-centered architectures. Specifically, we aim to address three challenges that are integral to this overarching issue:

C1 Irregular memory accesses The irregular memory accesses of workloads like graph processing cause significant performance challenges on conventional CPU hardware. On one hand, in the DRAM, these workloads cause effects like frequent row switching and ineffective use of fetched lines [Bes+19a; DRF23b; Lum+07; Ahn+15]. On the other hand, the fixed cache hierarchies of CPUs suffer from frequent cache misses and cache pollution. In total, this results in poor memory access efficiency and underutilization of compute resources for these workloads on CPUs.

C2 Limited parallelism The coarse-grained parallelism provided by modern CPUs works well for task parallelism (cores and hardware threads) as well as single instruction multiple data (SIMD) data parallelism (i. e., vectorization). However, especially SIMD parallelism is very rigid in CPUs and only works for particular workloads. For instance, integer parsing is very hard to parallelize in the rigid structure of CPUs because of the tight data dependencies, e. g., making performance of JSON parsing very unpredictable [Dan+22]. More fine-grained, application-specific, and unstructured parallelism would greatly benefit these kinds of workloads.

C3 Data movement Due to the centralized nature of CPU-centric computer architectures, they require frequent data movement between different components of the system over the interconnect (e. g., PCIe), since all data has to pass through the CPU. This unnecessarily uses a lot of bandwidth and keeps the CPU busy with data movement resulting in limited performance and long latencies [Vog+23]. For example, for JSON ingestion in document stores, data enters the system through the NIC, is written into the system memory only to be read from system memory again into the CPU, parsed, and written back to memory.

Discussion While system architects could previously rely on strong CPU performance scaling to satisfy growing performance requirements, more sophisticated, heterogeneous computer architectures, integrating different hardware accelerators, are explored nowadays. This trades off generality and higher design complexity in favor of higher performance. Due to the massive popularity of machine learning, GPUs are one popular accelerator option that offer massive parallelism. Yet, their performance deteriorates when their internal cores do not execute the same instruction (i. e., warp divergence), e. g., common for graphs with high degree variance [Shi+18]. Thus, for modern non-relational (and relational) database systems, field-programmable gate arrays (FPGAs) are explored for future performance scaling (e. g., [ISA17; Owa+17]).

1.2.2 FPGAs to the Rescue?

As such, FPGAs have gained traction for cloud-based data processing and have found applications in areas like data centers (e. g., Project Catapult [Put+14]), machine learning (e. g., neural network inference [Umu+17]), and also data management. Thus, FPGAs are potentially able to address the challenges faced by non-relational database systems. FPGAs are reconfigurable computing platforms that can implement custom massively parallel workload-specific processor architectures. Their architecture provides unmatched flexibility with no structural restrictions on parallelism. Additionally, all data on an FPGA is bit-addressable, and data types are not restricted to multiples of bytes, at least in on-chip registers. FPGAs give the user full control over memory accesses with custom-usable on-chip memory. Their flexibility also means that FPGAs can be placed anywhere in the system architecture which is important for concepts like near-data processing or processing-on-the-wire which allow for substantial reductions in data movement. When it comes to database systems, FPGAs could potentially even outperform the currently more prevalent GPUs for certain use cases [RL17]. However, FPGAs come with their own unique challenges of which we want to highlight three in the following that we will address with this thesis.

C4 Hardware design In general, the hardware engineering design flow for FPGAs is very different to that of instruction-based processors like CPUs and GPUs. FPGA design is about minimizing data dependencies to maximize achievable clock frequency. Thus, a major challenge with FPGAs is their inaccessible programming with hardware description languages typically used being very foreign to software engineers due to the unprecedented parallelism on FPGAs.

C5 Low latency workload switching On FPGAs, functionality implemented in the custom architecture spatially takes up limited resources on the chip. Different to instruction-based processors (e. g., CPUs and GPUs), for FPGAs, switching workloads may take reconfiguration of the whole chip for sufficiently complex implementations. Thus, one challenge for FPGA designs is coming up with abstractions that strike a good balance between specialization and adaptability. For instance, parameterization can be used for low latency workload switching foregoing costly reconfiguration of the chip.

C6 Novel memory technologies Recently, FPGA platforms with memory subsystems with large bandwidth like high-bandwidth memory (HBM) became available that are very difficult to leverage with CPUs. As the number of memory types used with FPGAs rises, it increasingly becomes a challenge to design systems that utilize the unique properties of the different types of memory.

Discussion FPGAs are not the silver bullet to instantly solve all challenges in non-relational data processing and come with their own set of challenges. Data processing is still relatively new for FPGAs and the tooling is not evolved to the same degree as, e. g., CUDA for GPUs. Additionally, because most database practitioners come from a software engineering background, the challenges C4–6 are particularly prevalent for utilizing FPGAs for databases. However, FPGAs show a lot of potential for acceleration of non-relational database systems [DRF23b] and, thus, we will address these challenges with this thesis.

1.2.3 Research Gap Non-relational Databases on FPGAs

With the combined challenges C1–6 in mind, we surveyed the literature and found the following gaps in existing research that need to be addressed for FPGA-based performance scaling of non-relational database systems. In this thesis, we aim to close the following six research gaps (an extended list of open research gaps can be found at the end of the survey in Sect. 3.4).

Survey research There is only limited academic research in terms of survey studies that cover the topic of non-relational database systems. While non-accelerated non-relational database systems have been extensively studied [Cat10;

DCL18; Gaj12; Ges+17; He15; Jin+11], and two surveys have covered FPGA-accelerated graph processing within the context of high-performance computing (HPC) [Bes+19b; Gui+19], there is a significant gap in survey studies on FPGA-accelerated non-relational database systems.

Benchmarks and reproducibility There are no widely agreed-upon benchmark suites and depending on the domain and workload no commonly used data sets. Especially for graph processing accelerators (e.g., [Sha+19; Zho+19; Che+22]), the reproducibility and comparability of existing research systems is lacking. This is due to multiple factors, the most important being almost no open source implementations and many different FPGA platforms making it very hard to directly compare performance measurements or alternatively implementing a set of accelerators for the purpose of comparison (cf. challenge C4).

Bandwidth-efficient accelerators For memory-bound workloads, like graph processing, that are challenged by irregular memory accesses (C1), FPGAs are well suited and a number of systems have been proposed [Yao+18; Sha+19; Zho+19; Che+22]. However, previous research has not explicitly explored how to utilize the available memory bandwidth most efficiently even though this is the bottleneck for these kind of accelerators. A systematic approach to design bandwidth-efficient FPGA-based accelerators is missing.

Scalable accelerators With the progress in memory technology, accelerators may choose from a wide range of different memory technologies. While memory technologies like persistent memory are out-of-scope for this work, modern memory providing high bandwidth data access like HBM and hybrid memory cube (HMC) are relevant in pursuing more performance and providing different memory characteristics to tackle challenges like irregular memory accesses (cf. challenge C1). While there has been general research on utilizing HBM on FPGAs [Wan+20b; Shi+22], in the context of non-relational database systems, it is especially interesting to explore how HBM can be utilized for accelerating graph processing which is notoriously memory-bound (cf. challenge C6).

Flexible data ingestion For workloads like parsing and format conversions that work on data that is already on the move through the network, SmartNIC deployments of FPGAs massively reduce data movement (cf. challenge C3). Currently the most popular data format for document database systems is JSON. While there has been related work on XML parsing [CHL08; DNZ10; Sid13; Hua+14] and parser generators for JSON parsing [Pel+21], there is a gap for a flexible parser that can handle arbitrary JSON documents during runtime. Additionally, even though modern CPU-based parsers (e. g., [LL19]) use vectorization, they are not utilizing the full potential of parallelization that is possible with this workload (cf. challenge C2).

Flexible query processing accelerators Different to HPC that is often targeted by FPGA accelerators, database workloads require flexible query processing with low latency workload switching which is a challenge on FPGAs (cf. challenge C5). There is a gap in suitable hardware abstractions that preserve the required degree of flexibility despite high specialization of the hardware design for optimal performance. There is related work for document and relational query processing (e. g., [TWN12; Mog+23]), however, there are no graph database system solutions for this challenge.

1.3 Research Questions

Based on these research gaps, this thesis answers the following research questions.

RQ1 How can non-relational database systems leverage FPGA acceleration?

- *RQ1.1 How do existing non-relational database systems utilize FPGAs?*
- *RQ1.2 Which solutions and gaps exist in current research on non-relational FPGA acceleration?*
- *RQ1.3 Which reoccurring patterns in the literature guide the design of FPGA-accelerated non-relational database systems?*

Research question RQ1 addresses the gap of lacking survey research for non-relational database regarding FPGA acceleration. The goal is to establish what has already been covered by research and which overarching patterns exist in related work. To this end, sub-question RQ1.1 explores existing commercial database systems to find out how FPGAs are currently utilized in real world systems. Delving into the academic literature, sub-question RQ1.2 seeks to provide an intuition for current research. Lastly, we look for reoccurring patterns in the previously introduced research for the design of new systems and acceleration of existing ones with sub-question RQ1.3. Together, answering these questions will provide a comprehensive picture of commercial use, academic solutions, and overarching patterns for FPGA acceleration of non-relational database systems.

RQ2 How can the FPGA use the available bandwidth for irregular memory accesses most effectively?

- *RQ2.1 Which crucial graph processing accelerator properties contribute to good performance?*
- *RQ2.2 How can the crucial graph processing accelerator properties be combined in one system?*
- *RQ2.3 How can accelerators be scaled to unlock the potential of novel memory technologies like HBM?*

Focusing on graph processing accelerators, sub-question RQ2.1 addresses the gap of lacking benchmarks practices and lacking understanding of existing accelerators. Therefore, we analyze the crucial properties that lead to good performance in existing graph processing accelerators. Sub-question RQ2.2 then seeks to understand how these crucial properties of existing accelerators can be combined as a new accelerator. This addresses irregular memory accesses and the gap of bandwidth-efficient accelerators, since graph processing is largely memory-bound. Lastly, sub-question RQ2.3 will explore the use of HBM for further performance scaling. In total, answering research question RQ2 will provide a comprehensive analysis of the current graph processing accelerator literature and use the insights from that to improve upon existing accelerators.

RQ3 How can flexible data ingestion and query processing be achieved on FPGAs?

- *RQ3.1 How can line-speed data ingestion of arbitrary input data be achieved on FPGAs?*
- *RQ3.2 How can algorithmic software approaches be useful to hardware design?*
- *RQ3.3 How can flexible subgraph query processing benefit from FPGAs?*

Research question RQ3 addresses the gaps of lacking hardware abstractions and flexibility of current accelerators with its three sub-questions. First, we will look at data ingestion of arbitrary JSON documents at line speed (sub-question RQ3.1). Thereafter, sub-question RQ3.2 targets how existing CPU-based software solutions may be brought to FPGA designs. Answering this question, we will focus on an example of set intersections on FPGAs. Lastly, sub-question RQ3.3 addresses query processing on FPGAs with flexible workload switching during runtime which is motivated by the gap on missing flexible accelerators.

1.4 Contributions

The main contributions of this thesis are a survey and several systems (GraphSim, GraphScale, PipeJSON, and GraphMatch) addressing the research questions RQ1–3. The contributions are based on papers that have been published at conference and journals. For each contribution, the following list shows the research question it addresses, in which chapter the research is presented in this thesis, and the papers that count towards it. The majority of the content of each chapter is based on the respective papers.

1. Survey including a *systematic literature review* of related work and a practitioners guide based on *reoccurring design patterns* for FPGA-accelerated non-relational database systems (cf. RQ1; Chapter 3)

   - Jonas Dann, Daniel Ritter, and Holger Fröning. “Non-relational Databases on FPGAs: Survey, Design Decisions, Challenges”. In: *ACM Comput. Surv.* 55.11 (2023), 225:1–225:37. [DRF23b]

   Figures and tables from this paper are used similarly in Fig. 1.1, Figures 3.1 to 3.7, Tab. 3.1, and Tab. 3.2.

2. Simulation environment *GraphSim* for memory-bound graph processing accelerators to make accelerator performance reproducible and comparable and gain deeper insight into accelerator properties. Crucial accelerator properties are: asynchronous graph processing, a compressed graph data structure, and multi-channel memory (cf. RQ2.1; Chapter 4)

   - Jonas Dann, Daniel Ritter, and Holger Fröning. “Exploring Memory Access Patterns for Graph Processing Accelerators”. In: *BTW*. 2021, pp. 101–122. [DRF21b]

  - Jonas Dann, Daniel Ritter, and Holger Fröning. “Demystifying Memory Access Patterns of FPGA-based Graph Processing Accelerators”. In: *GRADES-NDA*. 2021, 3:1–3:10. [DRF21a]

  Figures and tables from these papers are used similarly in Figures 4.1 to 4.11, Fig. 4.13, Fig. 4.14, Figures 4.16 to 4.22, and Tables 4.1 to 4.3.

3. Scalable general-purpose graph processing framework for FPGAs *GraphScale* combining multi-channel memory, asynchronous graph processing, and a compressed graph data structure resulting in an average speedup of 1.86× over state-of-the-art graph processing accelerators. Additionally, scaling to up to 16 HBM channels resulting in another average speedup of 1.53× over the base system (cf. RQ2.2 and RQ2.3; Chapter 5)

   - Jonas Dann, Daniel Ritter, and Holger Fröning. “GraphScale: Scalable Bandwidth-Efficient Graph Processing on FPGAs”. In: *FPL*. 2022, pp. 24–32. [DRF22]
   - Jonas Dann, Daniel Ritter, and Holger Fröning. “GraphScale: Scalable Processing on FPGAs for HBM and Large Graphs”. In: *ACM Trans. Reconfigurable Technol. Syst.* Just Accepted (2023), pp. 1–24. [DRF23a]

   Figures and tables from these papers are used similarly in Figures 5.1 to 5.17 and Tables 5.1 to 5.4.

4. FPGA-based JSON parsing accelerator *PipeJSON* that is fully pipelined for parsing of arbitrary JSON documents at line speed up to almost 20 GB/s denoting the bandwidth of a whole memory channel, achieving an average speedup of 12.56× over the fastest CPU parser (cf. RQ3.1; Chapter 6)

   - Jonas Dann et al. “PipeJSON: Parsing JSON at Line Speed on FPGAs”. In: *DaMoN*. 2022, 3:1–3:7. [Dan+22]

   Figures and tables from this paper are used similarly in Figures 6.1 to 6.10 and Tables 6.3 to 6.2.

5. Subgraph query processing system *GraphMatch* that is able to flexibly switch queries in a matter of clock cycles achieving an average speedup of

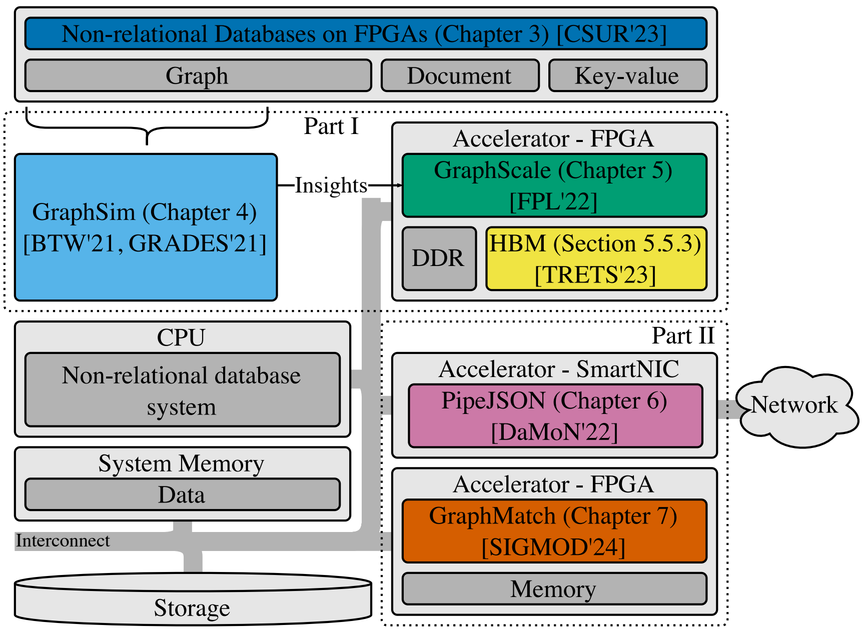

Non-relational Databases on FPGAs (Chapter 3) [CSUR'23]
Graph
Document
Key-value
Part I
Accelerator - FPGA
GraphScale (Chapter 5) [FPL'22]
GraphSim (Chapter 4) [BTW'21, GRADES'21]
Insights
DDR
HBM (Section 5.5.3) [TRETS'23]
CPU
Non-relational database system
Part II
Accelerator - SmartNIC
PipeJSON (Chapter 6) [DaMoN'22]
Network
System Memory
Data
Accelerator - FPGA
GraphMatch (Chapter 7) [SIGMOD'24]
Interconnect
Memory
Storage

Fig. 1.4 Overview of contributions and outline.

2.68× and 5.16× over the two fastest join-based subgraph query processing systems for CPUs (cf. RQ3.2 and RQ3.3; Chapter 7)

- Jonas Dann et al. "GraphMatch: Subgraph Query Processing on FPGAs". In: *SIGMOD*. Submitted (2024), pp. 1–12.

Figures and tables from this paper are used similarly in Fig. 2.3, Figures 7.1 to 7.17, and Tables 7.3 to 7.2.

Figure 1.4 shows the contributions in the context of the abstract non-relational database system architecture from Fig. 1.2. Combined as Part I, GraphSim and GraphScale together contribute to increased understanding of graph processing accelerator properties (reproducibility and comparability) and how to apply them to the real world. Combined as Part II, PipeJSON and GraphMatch show how to attain flexible data ingestion for arbitrary documents and flexible query processing with on-the-fly query switching. Together, the contributions of this thesis address the challenges (C1–6) from Sect. 1.2.

1.5 Thesis Outline

Starting with Chapter 2, we introduce the foundations of non-relational database systems and FPGAs that are shared among the subsequent chapters. Thereafter, the thesis follows the structure layed out in Fig. 1.4. In Chapter 3, we focus on related work across FPGA-based query acceleration for non-relational databases. Based on a taxonomy proposed by us, in a detailed literature analysis, we show existing solutions and later analyze open research gaps. Additionally, we propose a practitioners guide based on common patterns observed in the literature analysis to aid design of FPGA-accelerated non-relational database systems.

Starting Part I, Chapter 4 introduces the simulation environment GraphSim and memory access models for four state-of-the-art graph processing accelerators that are subsequently used to reproduce and compare performance along several dimensions. The insights from this analysis are used in Chapter 5 to design GraphScale. After we introduce the GraphScale concept, we show a detailed evaluation and how GraphScale can be scaled to HBM.

Part II, focused on flexible data processing, introduces the PipeJSON (Chapter 6) system architecture showing how it can parse arbitrary JSON documents followed by a detailed evaluation of parsing performance. Thereafter, we propose GraphMatch (Chapter 7). We show how set intersection, instrumental to subgraph query processing, benefits from FPGAs and is integrated into GraphMatch for flexible query processing, before we finish with a performance evaluation.

Finally, we conclude the thesis in Chapter 8, revisiting the challenges C1–6 (cf. Sect. 1.2.1 and Sect. 1.2.2) and research questions RQ1–3 (cf. Sect. 1.3) and provide an outlook on future work.

CHAPTER 2

Foundations

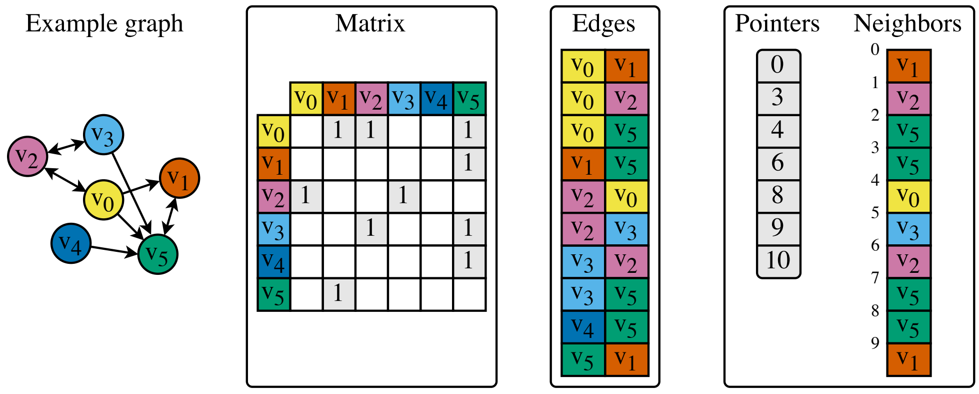
In this chapter, we introduce non-relational database systems and FPGAs.

2.1 Non-relational Database Systems

Non-relational databases are categorized based on their data models (as established in Fig. 1.1). Survey research on non-relational database systems [Cat10; DCL18; Gaj12; Ges+17; He15; Jin+11] shows that the predominant data models for these systems are graph, document, and key-value. While there are other data models like wide-column, spatial, object-oriented, and timeseries, they are not explored in this thesis. In the subsequent sections, we will introduce aspects of graph, document, and key-value database systems in that order, placing a special emphasis on the graph data model.

2.1.1 Graph

Graphs are abstract data structures $G = (V, E)$ comprising a vertex set V and an edge set $E \subseteq V \times V$. Intuitively, they are used to describe a set of entities (vertices) and their relations (edges). Throughout this thesis, we consider graphs to be directed by default, meaning edges have a direction from a source to a destination vertex. In some cases, marked as such, we also consider undirected graphs where we duplicate each directed edge as an inverse edge in the other direction to make the graph undirected.



(a) Adjacency matrix (b) Sorted edge list (c) Compressed spare row

Fig. 2.1 Example graph and graph data structures.

Figure 2.1 shows an example graph consisting of six vertices and ten edges and three distinct possible data structure representations of this graph. The graph's adjacency matrix A is a matrix of dimensions $|V| \times |V|$, where $A_{ij} = 1$ when there is an edge between vertices v_i and v_j (cf. Fig. 2.1(a)). Subsequently, Fig. 2.1(b) shows the same graph represented as a sorted edge list which stores the graph as an array of edges with a source and a destination vertex each. The array is sorted in ascending order by source and then destination vertex. Another possible data structure is a set of adjacency lists where each list contains the neighbors of a given vertex in the graph. As an exemplary implementation of this representation, the compressed sparse row (CSR) format is shown in Fig. 2.1(c). Used for the efficient storage of sparse matrices (in this case the adjacency matrix of the graph), CSR uses two arrays. The values of the pointers array at position i and $i+1$ delimit the range in the neighbors array that contains the neighbors of vertex v_i. For instance, for vertex v_3 the neighbors are the values of the neighbors array between indices 6 and 8, i. e., vertices v_2 and v_5.

For graph database systems, graph partitioning is often utilized to divide the graph into smaller subgraphs for the purpose of distributed processing. During graph partitioning, the graph is split up into multiple subgraphs. Within the context of this thesis, two primary partitioning dimensions are relevant: vertical and horizontal. The dimensions are in reference to the way the adjacency matrix of the graph is split up. Vertical partitioning divides the vertex set into distinct intervals such that each partition contains the *incoming* edges of its corresponding interval. In contrast, while horizontal partitioning also divides up the vertex set into intervals, each partition contains the *outgoing* edges of its corresponding interval. As a third partitioning approach, interval-shard partitioning [ZHC15] employs both vertical and horizontal partitioning at once. These partitioning approaches can be freely combined with the different graph data structures.

In the following, we will discuss the two main graph database workloads that are relevant to this thesis: graph processing and subgraph query processing.

2.1.1.1 Graph Processing

Graph processing traverses a data graph with given execution directives to analyze properties of the overall structure, its vertices, and edges. For this workload, we consider labeled graphs, i. e., each vertex has a value associated with it, its vertex label. Optionally, the edges may have edge weights. In the context of this thesis, we focus on five specific graph problems [Eve11]: breadth-first search (BFS), single-source shortest paths (SSSP), weakly-connected components (WCC), sparse matrix-vector multiplication (SpMV), and PageRank (PR).

For BFS, the vertices of the graph are visited in a breadth-first order. Starting with a root vertex as the frontier, in each iteration, every unvisited neighbor of the current frontier vertices is marked as visited, assigned the current iteration as its vertex label, and added to the frontier of the next iteration. The result per vertex is the number of edges to the root vertex along the shortest path.

For SSSP, the shortest distance to the root vertex is determined for each vertex $v \in V$ of the data graph. The shortest distance equals the minimum sum of edge weights of any path from the root to vertex v. If we consider each edge to have weight 1, the result is identical to that of BFS.

WCC identifies which vertices in the data graph belong to the same weakly-connected component. Two vertices are considered to be in the same weakly-connected component if there is any undirected path connecting them.

SpMV multiplies a vector (equal to the vertex labels of V) with a matrix (equal to E) in iterations over the data graph. PR is a measure to describe the importance of vertices in a graph originally used to rank web pages. It is calculated by recursively applying $p(i,t{+}1) = \frac{1-d}{|V|} + d \cdot \sum_{j \in N_G(i)} \frac{p(j,t)}{d_G(j)}$ for each $i \in V$ with damping factor d, neighbors N_G, degree d_G, and iteration t. The vertex labels are initialized as $p(i,0) = 1/|V|$.

Depending on the underlying graph data structure, these graph problems may be solved based on two fundamentally different approaches: edge- and

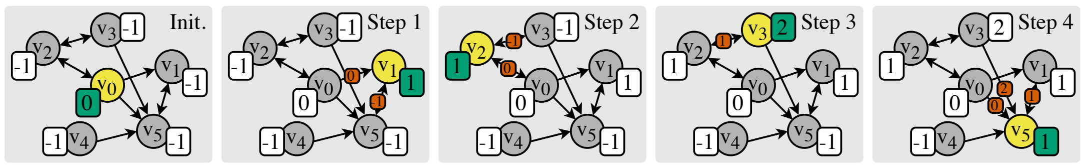


Fig. 2.2 Example for vertex-centric, pull-based, asynchronous BFS.

vertex-centric graph processing. Edge-centric graph processing iterates over the edges as primitives of the data graph on an underlying edge list. Vertex-centric graph processing iterates over the vertices and their neighborhoods as primitives of the data graph on an underlying adjacency lists data structure (e. g., CSR). Furthermore, for vertex-centric graph processing, there are two ways for vertex labels to flow along edges: push- and pull-based data flow. Push-based data flow denotes that vertex label updates from the current vertex are pushed along the forward direction of edges to update neighboring vertices. For pull-based data flow, vertex label updates are pulled along the inverse direction of edges from neighboring vertices to update the current vertex. Lastly, we differentiate between two update propagation schemes: asynchronous and synchronous graph processing. Asynchronous graph processing immediately applies updates to vertex labels as soon as they are produced and synchronous graph processing stores all vertex label updates in a separate data structure in memory and applies them only after the current iteration over the graph is finished.

Figure 2.2 shows an example of vertex-centric, pull-based, asynchronous BFS on the example graph from Fig. 2.1. The execution starts with the initialization of the vertex labels attached to each vertex. All vertices except the root vertex v_0 (vertex label 0) are assigned the vertex label -1. Thereafter, we try to update each vertex with the minimum of the vertex labels of the neighbors connected to the current vertex with incoming edges plus 1. For example, in step 1, the vertex label vertex v_1 is updated with $0+1$ from vertex v_0. Vertex labels of -1 are filtered out. This is done until we converge on a stable result and there are no more possible updates.

2.1.1.2 Subgraph Query Processing

Subgraph query processing seeks to find instances of a given query graph structure in a given data graph (i. e., pattern matching). It either computes subgraph homomorphisms or isomorphisms. Both denote subgraphs of the same shape as

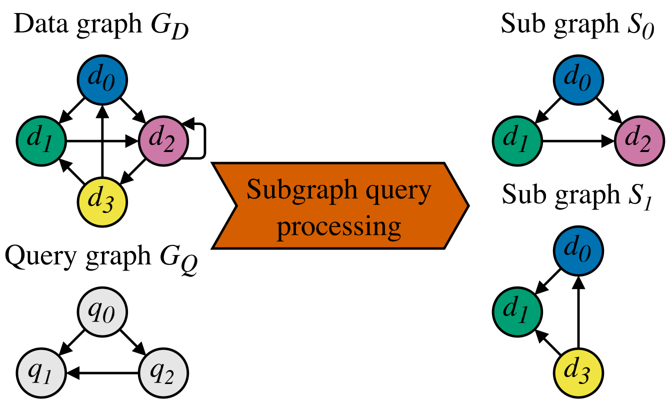


Fig. 2.3 Subgraph query processing example and all its isomorphisms.

the query graph, but homomorphisms allow duplicate vertices within matchings, while isomorphisms do not. Figure 2.3 shows an example of an unlabeled subgraph query processing task for directed graphs, given a data graph $G_D = (V_D, E_D)$ with four vertices and seven edges and an input query graph $G_Q = (V_Q, E_Q)$ with three vertices and three edges:

$$\begin{aligned}
V_D =& \{d_0, d_1, d_2, d_3\} \\
E_D =& \{(d_0, d_1), (d_1, d_2), (d_2, d_3), (d_2, d_2), (d_3, d_0), (d_0, d_2), (d_3, d_1)\} \\
V_Q =& \{q_0, q_1, q_2\} \\
E_Q =& \{(q_0, q_1), (q_0, q_2), (q_2, q_1)\} \ .
\end{aligned}$$

In the example, we identify all subgraph isomorphisms of the triangle G_Q within G_D. Thus, all results of the task are triangle subgraphs of G_D, where edges of the same direction exist in the data graph and no vertex is used multiple times. The result contains two graphs:

$$\begin{aligned}
S_0 = (V_{S_0}, E_{S_0}) :=& (\{d_0, d_2, d_1\}, \{(d_0, d_2), (d_0, d_1), (d_1, d_2)\}) \\
S_1 = (V_{S_1}, E_{S_1}) :=& (\{d_3, d_1, d_0\}, \{(d_3, d_1), (d_3, d_0), (d_0, d_1)\}) \ .
\end{aligned}$$

The homomorphisms of the example in Fig. 2.3 would include the subgraph isomorphisms S_0 and S_1 and four subgraphs with multiple occurrences of vertex d_2. There are two main approaches to processing subgraph queries [SL20; Lee+12]: exploration- and join-based.

Exploration-Based Subgraph Query Processing For exploration-based algorithms (also known as backtracking), the general idea is to explore the entire graph and create candidate sets of data vertices for each query vertex based on a set of constraints lowering the problem complexity. Thereafter, those sets are taken and all valid isomorphisms are enumerated.

In the literature, three different enumeration variations are known [Sun+20a]: direct-, index-, and preprocessing-enumeration. Their underlying ideas are the same, but they avoid or handle the start of the algorithm differently (e. g., enumerate subgraphs directly or create an index structure on the data graph beforehand). The general backtracking algorithm works similar for all variations. A candidate set along a query vertex ordering (QVO) is computed containing all data vertices that might be a valid entry for the given position. Additional information, like edge connections between candidates, is also collected in separate data structures. Afterwards, the algorithm uses the collected information and data structures to recursively enumerate all subgraph isomorphisms along the QVO. Additionally, each iteration computes a local candidate set by taking the connections between previous and future vertices into account. The recursion ends after the whole query graph is processed. The algorithm can also be adapted to find homomorphisms instead by allowing duplicate vertices during the enumeration step.

Join-Based Subgraph Query Processing Worst-case optimal joins (WCOJ) limit their runtime complexity to the worst-case output size of the algorithm [Ngo+18]. Join-based subgraph query processing is based on the WCOJ algorithm Generic Join [NRR13]. Generic Join describes an iterative approach to construct all homomorphisms (or isomorphisms with appropriate constraints) by joining one vertex at a time to a temporary subgraph (partial matching).

A known variation of Generic Join that is used in a lot of subgraph query processing systems is Leapfrog Triejoin [Vel12]. Leapfrog Triejoin starts with two vertices connected with an edge and extends those step by step. To find valid join candidates, it intersects the corresponding neighbourhoods of all previously matched vertices that share an edge with the new vertex in the query graph. All elements of the result set are joined to the current subgraph. This process creates multiple new partial matchings for the next iteration with the trie structure of the algorithm. After each of the subgraphs represents a valid matching for the full query graph or no partial matching can be extended anymore, the algorithm terminates. The intersections of Leapfrog Triejoin are computed by its Leapfrog algorithm. This algorithm searches for potential result values in ordered sets. It leaps from one value to the next within sets and jumps from set to set to exclude values that can not be part of the intersection result.

2.1.2 Document

Document database systems store formatted text documents in a hierarchical binary representation. Currently, the two most popular document formats for document database systems are Extensible Markup Language (XML) [Bra+00] and JavaScript Object Notation (JSON) [Bra+14]. Every document in such a database adheres to such a fixed format. Upon ingestion into the database system, they are parsed from a string representation that is slow to process to an internal binary representation to facilitate faster processing.

Listing 2.1 JSON example.

```json
{
  "name": {
    "first": "Alan",
    "last": "Turing"
  },
  "isAlive": false,
  "born": 1912,
  "almaMater": ["University of Cambridge",
    "Princeton University"]
}
```

Most leading document database systems utilize JSON, which is a text-based, language-independent data interchange format. Listing 2.1 shows an example of a JSON document. The JSON format can be recursively formalized as follows (omitting formal definitions of String and Number [Li+17b]):

$$\begin{aligned}
\textsc{Text}&\text{—}\textsc{Object}|\textsc{Array}\ldots\textsc{Object}|\textsc{Array}\\
\textsc{Object}&\text{—}\{\textsc{String}:\textsc{Value},\ldots,\textsc{String}:\textsc{Value}\}\\
\textsc{Array}&\text{—}[\textsc{Value},\ldots,\textsc{Value}]\\
\textsc{Value}&\text{—}\textsc{Object}|\textsc{Array}|\textsc{String}|\textsc{Number}|\textsc{true}|\textsc{false}|\textsc{null}
\end{aligned}$$

A JSON TEXT is a sequence of one or more JSON objects or arrays. Each OBJECT is enclosed in curly braces ("{" and "}"), and contains a sequence of zero or more key-value pairs, also called fields. Each key-value pair consists of a string key, followed by a single colon (":"), and a corresponding value. An ARRAY is an ordered collection of zero or more values separated by commas and surrounded by square brackets ("[" and "]"). Lastly, a VALUE can be an object, array, string in quotes ("""), number, true, false, or null. Thus, these structures can be used to represent nested objects and arrays.

Parsing means transforming a document represented as a raw string into a binary representation for an underlying application like a database system. The binary representation aids the accessing application in traversing the document and quickly accessing values. In particular, parsing includes, among other things, building an easily traversable token representation of the nested objects and arrays and transforming the text-based numbers into integers and floats.

2.1.3 Key-value

Key-value database systems operate on key-value pairs. The key is used for quick lookup of the value and the attached value may be arbitrary data. They may be used as an underlying persistence layer of another database or standalone depending on the use case. For this thesis, two data structures important to key-value database systems are of particular interest, namely hash tables and log-structured merge-trees (LSM-trees).

Hash tables are dictionary data structures that map keys to values [MS08]. Figure 2.4 shows an example hash table with a hash function that maps arbitrary string keys to five buckets. In this simplified example, each bucket is a linked list of key-value pairs. Thus, a hash table is able to provide quick look-ups and inserts for a very large sets of possible keys (arbitrary strings) where only a

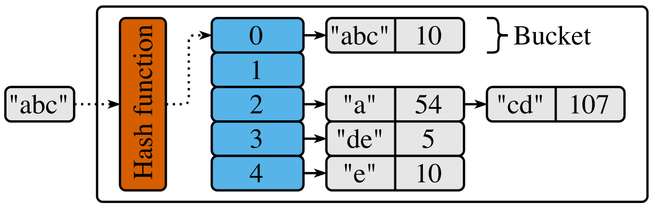


Fig. 2.4 Hash table.

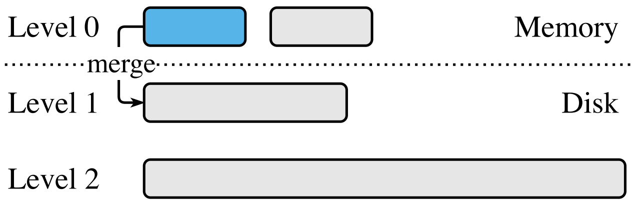


Fig. 2.5 LSM tree.

comparably small number of keys is actually in use with a small memory footprint. One of the main challenges for well performing hash tables are choosing a good hash function that equally distributes keys over buckets and thus produces as little hash conflicts as possible. A hash conflict happens when there is already a key in the bucket where a new one is inserted. In our example, we append the conflicting entry to the respective linked list making the look-ups for this bucket increasingly slower as the linked list grows.

LSM-trees [ONe+96] are used by key-value database systems to avoid write amplification and slow random write accesses to disk. The data structure is split up into multiple levels with the first one residing in memory and all subsequent levels residing on disk (cf. Fig. 2.5). Initially, new entries are inserted into blocks in memory that may represent different kinds of tree data structures. After a block of a fixed size is full, the block is merged into the first level on disk making all writes sequential. Depending on the implementation, disk levels may be split up into multiple blocks and blocks in disk levels may be merged into increasingly larger blocks in subsequent levels when hitting a certain threshold size.

2.2 Field-programmable Gate Arrays

A field-programmable gate array (FPGA) is a flexible processor architecture that can be programmed to mimic custom architecture designs, essentially imitating a specific set of logic gates and their connections (circuit design). This is similar to

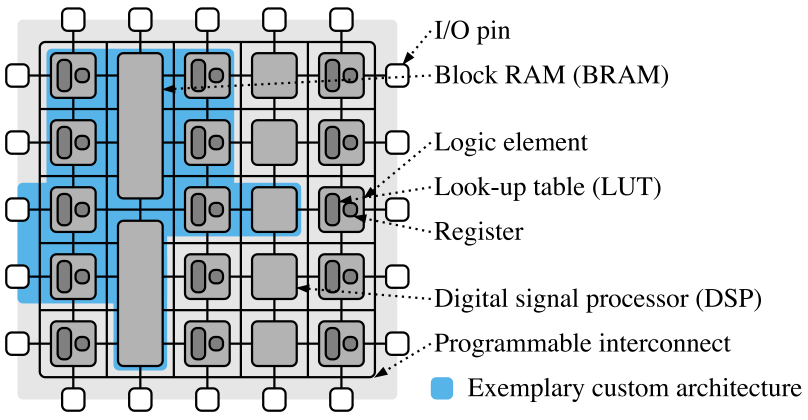

I/O pin
Block RAM (BRAM)
Logic element
Look-up table (LUT)
Register
Digital signal processor (DSP)
Programmable interconnect
🟦 Exemplary custom architecture

Fig. 2.6 FPGA on-chip resources.

application-specific integrated circuits (ASICs) that are custom-made to represent a circuit design. However, FPGAs are reprogrammable, allowing developers to modify the design at configuration time when needed. This flexibility has a trade-off: FPGAs are less power- and area-efficient and achieve lower clock frequencies than designs implemented as a custom ASIC. However, economic reasons as well as the need to adapt to changes in application behavior often prevent the use of ASICs. Thus, FPGAs emerged as accelerators for data processing (e. g., [TW13]) and are becoming increasingly important in the cloud [Bob+22]. They provide unique opportunities to implement functionality, like custom single instruction multiple data (SIMD) units or processing and data structure hybrids, like systolic arrays. While CPUs and GPUs are limited by the number of instructions performed per second, FPGAs are limited by the resources on the chip and the parallelism achieved by the circuit designer. However, whereas CPU and GPU programs are only practically restricted in complexity by program memory, the limited FPGA resources require careful planning of resource utilization. Additionally, features like synchronization among processing elements or memory prefetching have to be implemented by the designer and are not supported by default. In the following, we discuss the resources found on FPGA chips, the design flow for FPGAs, and how FPGAs are integrated into the computer architecture with FPGA boards.

2.2.1 Resources

Figure 2.6 shows an abstraction of an FPGA chip. On the chip, a custom architecture is mapped to a grid of resources (e. g., logic elements or block RAM

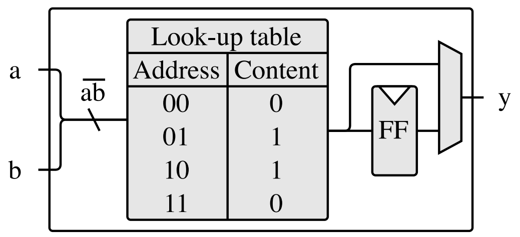


Fig. 2.7 Simplified FPGA logic element with look-up table and register (adapted from [Mog23]).

(BRAM)) connected with a programmable interconnection network. Each custom architecture uses a certain amount of resources upon configuration such that multiple custom architectures can be deployed on the same chip.

The logic itself is implemented with logic elements (cf. Fig. 2.7) (Intel calls them adaptive logic modules (ALMs) [Int22]) containing a look-up table (LUT) with multiple inbound (two in this simplified example) ports and one outbound bit port, a flip-flop (FF), which we will call register, for optional storage of the outbound bit, and a multiplexer. Each individual LUT is configurable to map any combination of input bits to either 0 or 1 as the output bit and is programmed upon configuration time. The LUTs are based on SRAM and are themselves purely combinational and stateless. Only the connection with the register makes them sequential and statefull. In modern FPGAs, logic elements additionally feature a carry chain connecting an additional output bit to neighboring LUTs for the implementation of adders.

The FPGA further contains on-chip BRAM in the form of SRAM memory components for high bandwidth and low latency (1 clock cycle) storage of data. BRAM is organized in blocks with a read and a write port that are simultaneously usable and, at least for Intel Stratix FPGAs, 512 entries deep of 40bit-wide values. On modern FPGAs, all BRAM blocks combined are about as large as the caches on a CPU ($\sim$ 30MB) but finely configurable to the mapped architecture. Additionally, the FPGA contains hardened digital signal processors (DSPs) that allow fast arithmetics on fixed- and floating-point numbers which would otherwise be very resource intensive if implemented in soft reconfigurable logic. Data enters and leaves the FPGA chip through I/O pins that are layed out on the edges of the chip.

All of these resources are connected with a programmable interconnection network running as bundles of wires horizontally and vertically between them (cf.

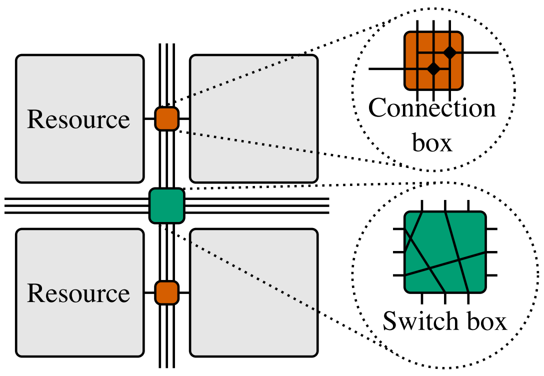


Fig. 2.8 Simplified FPGA interconnect (adapted from [Mog23]).

Fig. 2.8). The resources are connected to the interconnect via connection boxes that are also programmed at configuration time. In this simplified example, the connection box connects the left resource to the middle wire and the right resource to the right wire. Additionally, on each crossing of the wire bundles, there is a programmable switch box implemented as a crossbar which is able to connect arbitrary wires running through it. The switch boxes too are programmed at configuration time to make connections between farther apart resources. Because the number of wires is limited, wires themselves are resources and limit the number of connections locally as well as globally.

2.2.2 Design Flow

The FPGA design flow consists of eight steps which we discuss in the following: architecture design, verification, synthesis, mapping, place and route, timing analysis, bitstream generation, and FPGA programming.

Custom architectures for FPGAs have traditionally been *designed* with hardware description languages (HDLs) like VHDL (used in this thesis) and Verilog which describe architectures at the register transfer level (RTL), meaning an abstract description of registers and how data is transformed and transferred between them. More recently, higher level synthesis (HLS) languages, like Intel OneAPI, have been introduced. For OneAPI, architectures can be described in a C++ variant for data-parallel programming (DPCPP) making the design flow more accessible for software developers not familiar with RTL design at the cost of resource utilization overhead. However, the higher level representation of HLS languages is transpiled to RTL before entering the usual design flow. Interleaved with the architecture design, its functionality is tested with verification. Usually *verification* is done with testbenches for each component of the system (e. g., the VHDL designs in this thesis were verified with SystemVerilog testbenches).

Synthesis transforms the RTL description into a gate-based intermediate representation and *mapping* maps this representation to the technology-specific resources. Next, during the *place and route* step, the design is mapped to the actual resources by first assigning each resource required by the architecture design to a spot in the resource matrix available on the FPGA and then routing the connections between the resources through the programmable interconnection network. Since this is an NP-complete problem, place and route is done with a heuristic. After place and route, the number of each resource used in relation to the available resources on the FPGA determine the resource utilization which limits the architecture design complexity. The result of place and route is then run through *timing analysis*. Timing analysis looks at each path between two registers and determines the path with the longest signal propagation time of the whole design. The propagation time is influenced by two factors: delay due to resources (e.g., LUTs) the signal has to travel through and delay due to the signal traveling through the interconnect. This longest path (also called the critical path) is then used to determine the maximum clock frequency the design can be safely run at. Finally, a *bitstream is generated* that encodes the configuration of all used resources, like LUT configurations, and *programmed* to the FPGA. Modern FPGAs also support partial dynamic reconfiguration where only parts of the FPGA can be reprogrammed with a bitstream.

During the design process, low resource utilization is always a goal because this means fitting more functionality on the FPGA at once. However, oftentimes, clock frequency is the most critical factor because it directly influences the performance of the custom architecture. To improve clock frequency, the designer may split up the critical path into shorter paths by introducing registers repeatedly (this is only possible up to a certain point). This is called pipelining which trades off resource utilization for higher clock frequency.

2.2.3 Boards

Similar to the CPU being placed on a mainboard to integrate it into the computer architecture, FPGA chips are soldered onto a board connecting components like network ports, DRAM chips, baseboard management controllers (BMC), and

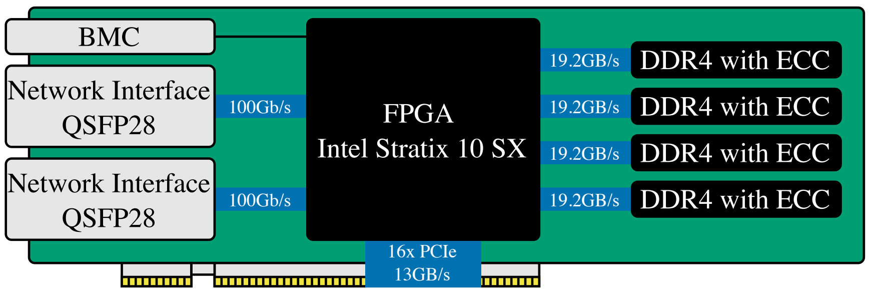


Fig. 2.9 Simplified FPGA board representing an Intel D5005 board (adapted from [Int19]).

PCIe connectors to the FPGA (cf. Fig. 2.9). Concerning DRAM, FPGAs support DDR3 and DDR4 as well as modern stacked memory, like high-bandwidth memory (HBM) and Hybrid Memory Cube (HMC), enabling high bandwidth on-board data processing. As an accelerator, boards containing an FPGA are most often placed in existing hardware systems together with a CPU – the host system. Traditionally, accelerators are connected with the CPU over PCIe but recently there are also cache-coherent interfaces (e. g., UPI, CXL, OpenCAPI). This allows deeper integration of the FPGA into the CPU's data management and sets FPGAs apart from other accelerators like GPUs that are lacking such capabilities. Such integration potentially frees the CPU of managing all data movement to perform other tasks. Lastly, many boards feature one or more network interfaces (e. g., Quad Small Form-factor Pluggable (QSFP) network interface) to directly connect the FPGA to the outside world. This enables deployment of FPGAs as smart network interface controllers (SmartNICs) that integrate compute capabilities into the NIC with desirable latency and throughput properties (more placement options will be discussed in Sect. 3.3.2.1).

The components on the board are directly accessible from the FPGA through the I/O pins. However, for example, for DDR4 memory, the user needs a memory controller between the I/O pins and the FPGA design. Thus, there are wrappers available for most boards that provide easy access to the board components. In this thesis, we use Intel Open Programmable Acceleration Engine (OPAE) for VHDL designs which provides standardized interfaces (called Avalon interfaces) for the memory channels and PCIe bridge and the respective OneAPI board support package (BSP) for OneAPI designs. BSPs are another wrapper on top of OPAE providing higher abstractions for accessing board components.

CHAPTER 3

Non-relational Databases on FPGAs: State-of-the-art

While FPGA-accelerated non-relational database systems may meet the requirements of emerging applications (cf. Sect. 1.2.1), there is only limited academic work in terms of survey research. Figure 3.1 depicts recent surveys related to FPGA-accelerated non-relational database systems. We consider surveys on *relational* and *non-relational* database systems in the context of three kinds of acceleration: *no accelerator*, *FPGA*-, and *GPU*-accelerated. The contribution to non-relational data processing is further specified by the non-relational database system classes (graph, document, key-value, and wide-column).

Leaving the vast amount of work on non-accelerated relational database systems out of scope, there remain several related surveys on FPGA and GPU

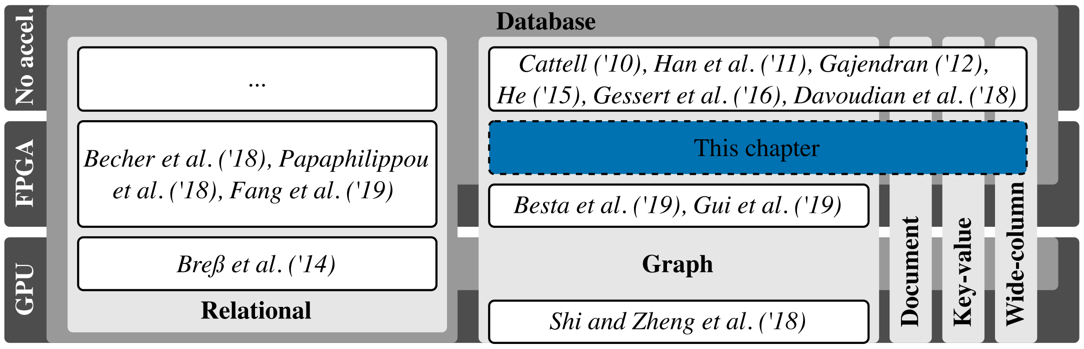

No accel.
FPGA
GPU
Database
...
Becher et al. ('18), Papaphilippou et al. ('18), Fang et al. ('19)
Breß et al. ('14)
Relational
Cattell ('10), Han et al. ('11), Gajendran ('12), He ('15), Gessert et al. ('16), Davoudian et al. ('18)
This chapter
Besta et al. ('19), Gui et al. ('19)
Graph
Shi and Zheng et al. ('18)
Document
Key-value
Wide-column

Fig. 3.1 Related surveys and this chapter.

acceleration of such systems. In [Bec+18], Becher et al. pose the challenge of using the on-the-fly reconfigurability of FPGAs in modern relational databases. This results in open questions on exploitation of heterogeneous hardware, query partitioning, and dynamic hardware reconfiguration. Papaphilippou et al. [PL18] categorize the literature into frameworks (e. g., Centaur [Owa+17]) and specialized accelerators for common operators in a modern relational database, and also highlight upcoming cache-coherent connectors for FPGAs (e. g., OpenCAPI). Most recently, Fang et al. [Fan+20] state main memory access, programmability, and GPUs as the three biggest factors holding back FPGAs in in-memory relational database systems in the past. Regarding GPU-accelerated relational database systems, [Bre+14] raises challenges like the I/O bottleneck and query planning but does not offer convincing solutions.

Non-accelerated non-relational database system are well-covered by surveys [Cat10; DCL18; Gaj12; Ges+17; He15; Jin+11] – which are discussed in Sect. 3.1. Additionally, there are two surveys covering FPGA acceleration of graph processing as a high-performance computing (HPC) workload [Bes+19b; Gui+19]. While this shows the feasibility of graph processing on FPGAs and the respective literature may be relevant for acceleration of graph database operators, these surveys focus only on functional aspects and HPC applications. For FPGA-accelerated non-relational database systems, this literature has to be reinterpreted from the database perspective. GPU acceleration (e. g., [Shi+18] for graph) is considered out of scope of this chapter.

In summary, the related surveys support the relevance and timeliness of the topic but there is no survey on FPGA-accelerated non-relational database systems. The objective of this chapter is to fill this gap in the intersection of FPGA acceleration and non-relational database systems and answer research question RQ1 *"How can non-relational database systems leverage FPGA acceleration?"*. The differences and commonalities of FPGA acceleration of non-relational database system classes result in overarching patterns that will be instrumental to current and new systems and guide the remainder of this thesis.

In the following, we propose a taxonomy of *system aspects* (abstract requirements for FPGA-accelerated non-relational database systems) resulting from a review of existing non-relational database systems and challenges in accelerator design for FPGAs (Sect. 3.1). We provide a table classifying references by system aspects (Tab. 3.2) and short *solution summaries* resulting from a comprehensive literature search (Sect. 3.2). We extract *patterns* from literature with regard to tasks, placements, non-trivial accelerator design decisions, and accelerator justification as a practitioners guide (Sect. 3.3). From the gaps in the literature, we derive *open research gaps* in the field of FPGA-accelerated non-relational database system (Sect. 3.4).

Parts of this chapter have previously been published in the ACM Computing Surveys [DRF23b] journal.

3.1 Taxonomy

In this section, we collect important non-relational database and FPGA system aspects of all non-relational database system classes in a taxonomy (Fig. 3.2) to guide the subsequent literature analysis. For the non-relational database system aspects, we discuss the non-relational database system classes in the context of well-known, commercial systems to provide an answer to research question RQ1.1 *"How do existing non-relational database systems utilize FPGAs?"*.

3.1.1 Non-relational Database System Aspects

Non-relational database systems are defined according to their data models (as established in Sect. 1.1). We found six surveys on non-accelerated non-relational database systems (cf. Fig. 3.1) [Cat10; DCL18; Gaj12; Ges+17; He15; Jin+11]. According to these surveys, the predominant non-relational database system classes are graph, document, key-value, and wide-column. Other non-relational database system classes – that we do not further consider in this work – are e. g., spatial, object-oriented, and timeseries database systems.

To review well-known, commercial non-relational database system, we selected the systems of each class that were at least mentioned three times in the non-relational database system surveys and combined similar systems (e. g., Riak KV is similar to Amazon DynamoDB). Additionally, we added OrientDB and RocksDB (marked as "expert" in Tab. 3.1) despite them only being named in one survey each because of their unique characteristics. OrientDB is the only non-relational database system that successfully combines graph, document, and object-oriented database concepts and RocksDB is based on the relevant LSM-tree data structure. Table 3.1 shows the selection of non-relational database

Table 3.1 Commercial non-relational database systems found in surveys.

Class	System	Survey references	FPGA
Graph	Neo4j [inc21]	[Cat10; DCL18; Ges+17; He15]	🖒
	OrientDB [SE22]	[DCL18] (expert)	🖓
Document	CouchDB [Fou22]	[Cat10; DCL18; Gaj12; Ges+17; He15; Jin+11]	🖓
	MongoDB [inc22a]	[Cat10; DCL18; Gaj12; Ges+17; He15; Jin+11]	🖓
Key-value	Redis [inc22b]	[Cat10; DCL18; Ges+17; He15; Jin+11]	👍
	DynamoDB [DeC+07]	[Cat10; DCL18; Gaj12; Ges+17; He15]	🖓
	BerkeleyDB [OBS99]	[Cat10; DCL18; Gaj12; He15]	🖓
	RocksDB [Bor+22]	[Ges+17] (expert)	🖓
Wide-column	Cassandra [LM10]	[Cat10; DCL18; Gaj12; Ges+17; He15; Jin+11]	🖒
	HBase [Tea22]	[Cat10; DCL18; Gaj12; Ges+17; He15; Jin+11]	🖓

👍: productive FPGA usage, 🖒: explored FPGAs but no productive usage, 🖓: no FPGA acceleration mentioned

systems by class. The review will help to establish non-relational database system aspects by highlighting the systems' design choices and challenges and discuss similarities between non-relational database system classes. We considered public documentation and publications, if available. Subsequently, the concepts of non-relational database system classes are introduced for the systems in Tab. 3.1.

3.1.1.1 Graph

We look at the graph database systems Neo4j and OrientDB. Neither of those systems deploy FPGA acceleration in production. However, Neo4j experimented with the Flash accelerator CAPI SNAP [Wil17], where an FPGA is inserted into the datapath between CPU and Flash storage to accelerate data accesses.

Neo4j [inc21] stores data as a property graph (cf. Fig. 1.1). It supports Cypher [Fra+18] as a graph query language with a cost-based query optimizer, traversal patterns like BFS, and algorithms to solve common graph problems like shortest paths and centrality (e. g., PR). For solving graph problems, Neo4j transforms the property graph into an in-memory projection that is optimized for traversal and supports different index data structures. To provide horizontal scalability, Neo4j can run as a cluster of core nodes that replicate all changes between themselves. For additional read scaling, read replicas can be added to the cluster by registering at a core node. Changes to graphs follow causal consistency where replication to a majority of core nodes has to finish to confirm a transaction. Optionally, ACID (atomicity, consistency, isolation, and durability) transaction guarantees can be enforced. For multi-tenant usage (multiple users working on the same system), Neo4j provides role-based access control (RBAC) and intra-cluster encryption, and encrypted backups provide further security.

Another well-known graph database is OrientDB [SE22] that allows for queries and traversals on a property graph with native support for documents. OrientDB provides SQL-like language support with a graph extension and operators like BFS. For scalability, data can be replicated over a cluster of nodes and availability is guaranteed by multi-master replication (similar to Neo4j) and auto discovery of nodes. When multiple users are working on the database, an optimistic multi-version concurrency control (MVCC) protocol is used and ACID transaction guarantees can be applied.

3.1.1.2 Document

Apache CouchDB and MongoDB are the most-referenced document database systems. We were not able to find FPGA acceleration options for these two document database systems.

Apache CouchDB [Fou22] is a JSON document database system operating as a cluster of master nodes with bidirectional, asynchronous replication. The API allows create, read, update, and delete (CRUD) operators on documents and more advanced view models for filtering and aggregation. Queries can be accelerated with B-tree index data structures. Writes to the database are isolated with MVCC and ACID transaction guarantees can be enforced.

MongoDB [inc22a] is another JSON document database system providing similar query possibilities to Apache CouchDB but adds capabilities for fulltext search and spatial queries. Availability is provided by a master-slave cluster setup, where nodes are placed into the replication set of a master node that handles all writes. If the master node fails, a new one is elected among the replica nodes. Data can also be sharded over multiple replication sets for scalability. Writes are atomic and transactions can span multiple documents.

3.1.1.3 Key-value

We subsequently discuss Redis, Amazon DynamoDB, Oracle BerkeleyDB, and RocksDB. For Redis, there is a recent accelerator extension by Algo-Logic [Loc20] with FPGA accelerators directly attached to the network for increased throughput, lower latency, and less energy consumption.

Redis [inc22b] may be used as an in-memory or persistent key-value database system. Key-value pairs are inserted into a hash table, where the key hash is used as the index in the table for fast lookup of values, but complex queries are not supported. Similar to the graph and document database systems, Redis uses a master-slave setup with sharding for scalability. Additionally, it supports request routing on a proxy node and optional waiting on replication for consistency.

Amazon DynamoDB [DeC+07] features a distribution scheme without a master. The key hashes denote a circular space and each node is randomly placed in this space. Each node is then responsible for the keys preceding it in counter-clockwise order in this circular space and answers all requests to that partition. Data is replicated in clockwise order to a fixed number of nodes and upon failure the node after a failed one is responsible for the failed nodes partition of the circular space. For load balancing, Amazon DynamoDB places many more virtual nodes on the circular space than there are nodes in the cluster. Multiple virtual nodes are then handled by each physical node.

Oracle BerkeleyDB [OBS99] is a local key-value database system without network capability that, in contrast to Redis and Amazon DynamoDB, uses an underlying tree data structure instead of a hash table, specifically a B+ tree. A B+ tree is a k-ary tree with multiple children per node. Additionally, the leaf nodes are linked which benefits range operations.

RocksDB [Bor+22] is a tree-based key-value database system using an LSM-tree data structure. The LSM-tree features multiple levels of pages where key-value pairs are inserted in the first level. If the first level page reaches a certain size, it is merged with second level pages with overlapping key ranges which is then recursively applied to resulting full pages. RocksDB may be partitioned over column families for scale-out to multiple nodes with a guaranteed consistent view over column families. It also provides transactions between tenants with a pessimistic and optimistic mode.

3.1.1.4 Wide-column

Wide-column database systems store data as pairs of keys and values, too. However, the keys have two predefined parts: a row name and a column name. With this predefined structure, wide-column database systems present their data to the user as tables but unlike relational databases these are unstructured, not materialized, and every row may have arbitrary columns. The most-referenced wide-column systems are Apache Cassandra and Apache HBase. For Apache Cassandra, FPGAs may be used to accelerate the data accesses where FPGAs may be used data proxies [Pot19].

Apache Cassandra [LM10] uses a multi-dimensional map, indexed with a key (everything with the same row name constitutes a row) and columns that are grouped into column families. It supports insert, get, and delete operations and additionally has a Cassandra query language (CQL), comparable to SQL. Similar to Amazon DynamoDB, the key hashes are used as a circular space to partition the data to the nodes in the cluster but load balancing is done by moving nodes in the circular space when imbalance is detected.

Lastly, Apache HBase [Tea22] is a wide-column database system based on Apache HDFS. The supported operator set as well as binary representation is similar to Apache Cassandra but Apache HBase distributes updates in a cluster of nodes in a master-slave fashion.

3.1.2 FPGA System Aspects

Subsequently, based on the foundations of FPGAs (Sect. 2.2), we discuss the system aspects of FPGA accelerator design we identified as most relevant to this chapter. These are based on differences of designing FPGA accelerators compared to designing CPU applications and also found to be challenging design decisions in related surveys [Bec+18; Bes+19b; Fan+20], namely *(i)* software vs. hardware design as *design paradigm*, *(ii)* tight *CPU-FPGA collaboration* between existing CPU-based systems and FPGA accelerators, *(iii)* custom *memory access* controllers, and *(iv)* comparability through a *performance model.*

Design paradigm In principle, all LUTs (>1,000,000 on modern FPGAs) can operate in parallel. This opens up a vast design space (for RTL design) compared to the well-formed design space of instruction-based processors (CPUs and GPUs). While there have been efforts to simplify development and increase programmability with higher-level languages (e. g., Intel OneAPI), they do not seem to satisfy performance requirements for complex applications yet [Yan+17]. One key question of non-relational database system acceleration is thus how to use constraints inherent to the non-relational database system class to reduce the accelerator design space without performance degradation.

CPU-FPGA collaboration Adding an FPGA to a data processing system is justified with sufficient improvement in overall performance and energy and cost savings. However, most systems still require a CPU, introducing data movement overhead between the two processors. Thus, effective accelerator integration requires not only high workload utilization of the FPGA but also little data movement overhead. CPU-FPGA collaboration has to solve problems of task orchestration and data management.

Memory access In contrast to CPUs, FPGAs do not access their on-board memory through a deep cache hierarchy that assumes temporal and spatial locality in memory accesses. Thus, FPGAs can implement unique caching strategies and placement of critical data on-chip.

Performance model The circuit-based programs of FPGAs have very different performance implications than instruction-based programs of CPUs. For big designs, it is not easy to comprehend the amount of parallelism and make performance predictions. Thus, custom performance models using the properties of the non-relational database system class are instrumental in understanding design decisions and comparing different approaches (e. g., [DRF21a]).

3.1.3 Discussion

Although the architectures of the systems reviewed exhibit different feature sets, they cover a shared set of system aspects. For example, Neo4j provides an execution engine for Cypher while Apache Cassandra provides CQL. Scalability in Amazon DynamoDB and Apache Cassandra is achieved by partitioning data into a circular hash space while e. g., MongoDB and Redis distribute work in a master-slave fashion. While the query languages and scalability schemes differ in their concrete implementation, queries and scalability are two integral system aspects of any non-relational database system.

Figure 3.2 shows a generalized shared architecture as a component view of the reviewed non-relational database systems capturing the system aspects we found in the system review. The architecture is based on a set of nodes in a networked environment, forming a cluster. Multiple different users or tenants send requests (dotted and dashed arrows) to the system which distributes work with the request router and load balancer in the cluster manager (scalability

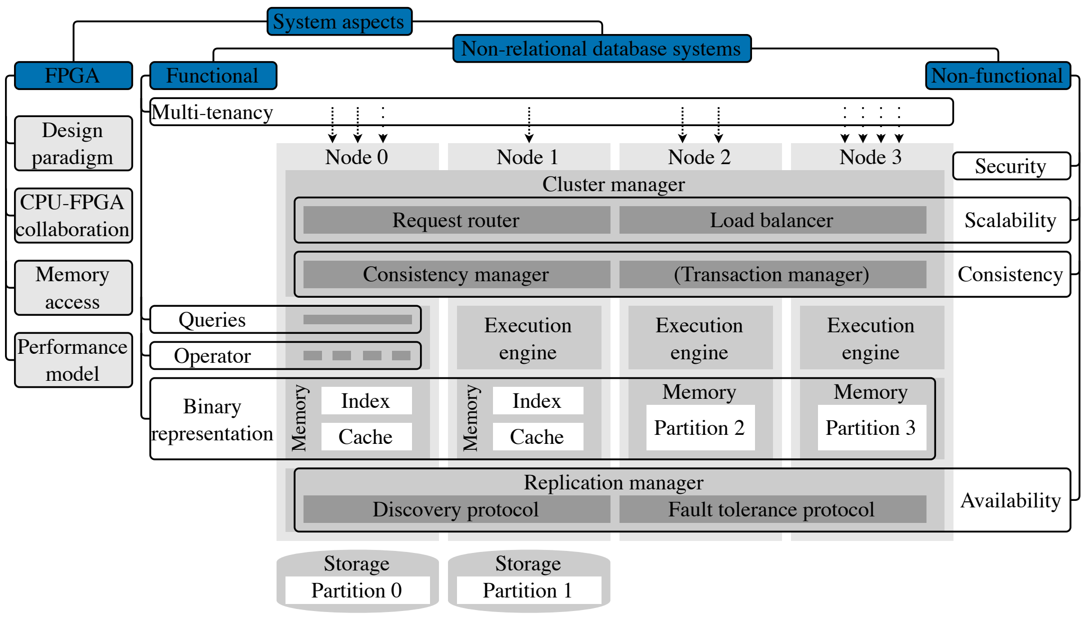


Fig. 3.2 Taxonomy of system aspects along a common non-relational database system architecture (different line styles for requests from different tenants).

means scale out here). Changes to the underlying data are kept consistent by the consistency manager which performs concurrency control and atomic writing. Optionally, the transaction manager provides transactions (e. g., ACID) touching multiple data elements. The execution engine finally processes queries – made up of operators – on the data stored in the system. The underlying data in memory is partitioned over the nodes in the cluster and query performance may be improved with indices and caches. The data may be stored in persistent storage if desired. Lastly, the replication manager aids discovery of new nodes in the system and fault handling. While the components in general might be similar to scale out relational database systems, non-relational database systems emphasize different components because of different application requirements.

As an overlay in Fig. 3.2, we show the FPGA system aspects and functional and non-functional non-relational database system aspects. A complete system would have to satisfy most of the combined system aspects even though FPGAs might not be able to provide improvements on all aspects. This aspect taxonomy will be used in the literature analysis (Sect. 3.2) for classification. In summary, there are some first experiments with FPGAs in non-relational database systems [Wil17; Loc20; Pot19] that show the potential and feasibility but the majority of systems have not yet considered FPGAs.

3.2 Literature Review

In this section, we conduct a literature analysis in order to answer research question RQ1.2 *"Which solutions and gaps exist in current research on non-relational FPGA acceleration?"*. The literature analysis is based on the guidelines described in [Kit04]. The primary selection of references is conducted in the domain of each individual non-relational database system class and with a focus on research papers (no patents and citations) with a connection to FPGAs, reconfigurable hardware, and acceleration. This resulted in 443 hits before the following selection criteria were applied: *(i)* focus on data processing (excluding e. g., robotics, image processing, and graph-based FPGA design), *(ii)* availability of the document, *(iii)* written in English, *(iv)* peer-reviewed (excluding Master and PhD theses). Overall, this resulted in 96 selected papers relevant to this literature analysis. Notably, we did not find dedicated literature for FPGA acceleration of wide-column database systems which, however, can be seen as a special variant of key-value database systems (cf. [DCL18]).

Subsequently, we summarize the solutions identified in the literature search. We organize the summaries by non-relational database system class. Additionally, we structure the papers by their strongest contribution(s) to the system aspects (cf. Fig. 3.2). Table 3.2 shows the papers and subclassification for each non-relational database system class and system aspect pairing. We only present non-relational database system class and system aspect pairings that are covered by literature.

3.2.1 Graph

The most accelerator solutions were provided for graph in the context of HPC (i. e., not specifically tuned towards non-relational database system). In the following, we will discuss the system aspects for graph database systems.

3.2.1.1 Operator

In the literature, we identified solutions for seven graph operators which we discuss in the subsequent paragraphs in the following order: shortest path, breadth-first search, maximum matchings, page rank, centrality, sparse matrix-vector multiplication, and subgraph query processing.

Table 3.2 Contributions by non-relational database system class and system aspects.

	Graph	Document	Key-value
NRDS – functional			
Operator	**SP** [Bet+12b; Bet+12a; Bon+06b; JSL11; Lei+16; Mil+07; THK14; THK15; ZCP15a]; **BFS** [Fin+19; LRG15; UMJ15; Wan+10]; **MM** [Bes+19c]; **PR** [ZCP15b; Mei+20]; **Cent.** [GSP16]; **SpMV** [Fow+14]; **SM** [Jin+21]	**XML parser** [DNZ10; Hua+14; Sid13]; **Filter** [Cha+12a]; **XPath** [EI10; Mit+09; Mou+10]; **Twig** [Mou+11]; **JSON parser** [Pel+21; Dan+22]	**Insert** [Lia+16]; **Hash** [Liu+19]; **LSM compaction** [Sun+20b; Zha+20c; Hua+19]
Binary representation	**HP** [ZCP16]; **VP** [Che+19]; **IS** [Dai+16]; **CSR** [Att+14]; **HWM** [Hue00; MHH02]; **Dynamic** [Wan+21]; **Other** [SSP06; TS18; Wan+15; WZH16; Xu+18]		**Hash table** [Blo+13; Ist+13; TZP15; Yan+20b; Zha+20b]; **CAM** [LM15]; **B+ tree** [Yan+21]
Queries	**Instructable processor** [Kap15]; **Linear algebra** [Hu+21]; **SQP** [Dan+24]	**Skeleton automata** [TWN12]; **Other** [Lun+04]	n/a
Multi-tenancy			**Token bucket** [IAS18]
NRDS – non-functional			
Scalability	**Part.** [Att+15; BFA96; Dai+17; Zha+20a]		**Repl.** [ISA17]; **Part.** [Qiu+20; Xu+16]
Availability			**Repl.** [ISA17]
Consistency	**Locking** [MZC17]		**MVCC** [Ren+19]
Security			
FPGA			
Design paradigm	**VC** [ES16; Nur+14; Ozd+16; WNH13; Yao+18; DRF22]; **EC** [Zho+19; Zho+18; Che+21b]; **Hybrid** [Yan+20a]; **BSP** [Ayu+18; DeL+06]; **Other** [BFA96; DMP99; OO16; Jin+17]		n/a
CPU-FPGA collaboration	**Socket** [Bon+06a; UMJ15; WHN19; ZP17; AOA20; OAA21]; **Near-data** [Lee+17]	**Socket** [VAM09]; **Near-data** [TWN13]; **PCIe** [Van+13]	**Near-data** [LAC14; Xie+19]; **DMA** [Li+17a; Qiu+18]
Memory access	**Req. merging** [Kho+18; AI21]; **Caching** [Sha+19; Sha+20; ZKL17]; **Data placem.** [ZL18]; **HBM** [DRF23a]; **Other** [Bet+11; Ni+14; Yan+19; Wan+20a]		**Bloom filter** [CC14]; **Flash** [Blo+15; Xu+16]; **HBM** [YKP20]
Performance model	**Parallelism** [Bon+06b; Sha+19]; **Lower bound** [Dai+16; Dai+17; EHS18; ZKL17] **Memory access** [DRF21b; DRF21a]		**Lower bound** [Qiu+20]

NRDS: non-relational database system, BFS: Breadth-first search, BSP: Bulk-synchronous parallel, CAM: Content addressable memory, Cent.: Centrality, CSR: Compressed sparse row, Data placem.: Data placement, DMA: Direct memory access, EC: Edge-centric, HBM: High-bandwidth memory, HP: Horizontal partitioning, HWM: Hardware mapping, IS: Interval-shard, MM: Maximum matchings, MVCC: Multi-version concurrency control, Part.: Partitioning, PR: PageRank, Repl.: Replication, Req. merging.: Request merging, SP: Shortest path, SM: Subgraph matching, SpMV: Sparse matrix-vector multiplication, SQP: Subgraph query processing, VC: Vertex-centric, VP: Vertical partitioning

Shortest path. Bondhugula et al. present a tiled Floyd-Warshall implementation solving the all-pairs shortest-paths problem using a pipeline of B processing elements (PEs) which can each process l elements of the adjacency matrix [Bon+06b]. They propose a performance model with these two parameters where B is constrained by FPGA resources and l is constrained by I/O bandwidth. Jagadeesh et al. use a parallel, synchronous Bellman-Ford implementation where each PE on the FPGA represents one node in the graph (severely limiting graph size) [JSL11]. Additionally, they model the internal computation time of their shortest path computation unit in cycles. A parallel implementation of Bellman-Ford that considers conflicting vertex updates is presented by Zhou et al. [ZCP15a]. The edges are processed in parallel and conflicts are resolved by caching updates on-chip until they are applied to memory. To optimize for sequential reads, the edges are sorted by destination vertex. In [THK14; THK15], Takei et al. combine Dijkstra with a SIMD distance comparison unit. A restriction of the approach is that all nodes have to fit into on-chip memory. Lei et al. propose an eager Dijkstra algorithm based on their own memory overflow extension of a priority queue and three memory channels (for overflow queue, graph, and output data) [Lei+16]. This resolves the graph size restriction of the earlier approach by Takei et al. In [Mil+07], the authors propose a solution for all-pairs shortest-path by defining a partitioning that allows for processing the graph with a bidirectional systolic array with an optimal number $|V|$ of PEs. Betkaoui et al. solve all-pairs shortest-paths with parallel BFS kernels [Bet+12b; Bet+12a]. Multiple parallel PEs issue non-blocking memory requests to take advantage of the multi-channel memory system of their particular FPGA setup.

Breadth-first search. Wang et al. propose a solution for a parallel BFS with message passing on a fully-connected network between interval-partitioned soft cores [Wan+10]. The traversal levels are synchronized with a floating barrier. The number of random memory accesses is reduced by keeping the visited status in on-chip memory. Additionally, their approach allows switching traversal patterns (bottom-up and top-down) per level. TorusBFS [LRG15] proposed by Lei et al. implements torus network-based message-passing between PEs in an interval-partitioned graph. The PEs are similar to those in [Bet+12b], and auxiliary data structures are stored in BRAM. In [UMJ15], BFS iterations are substituted by matrix-vector operations on a Boolean semi-ring (i.e. multiply and add are substituted with logical `and` and `or`). The data is horizontally partitioned but the matrix and vectors are never materialized. Dr. BFS [Fin+19] uses vertical partitioning to fit metadata of large graphs into on-chip memory. The tasks of computation and data access are separated into two different modules having a data access module for every memory bank. The computation modules use a pipelined combination network for sequential burst operations.

Maximum matchings. Besta et al. propose a solution for maximum matchings in a graph which describes the maximum size set of edges that do not share a vertex [Bes+19c]. Their substream-centric approach divides the incoming stream of edges by their weight into multiple substreams that are processed in parallel Afterwards, the results are merged again.

PageRank. Zhou et al. propose an implementation of PR split up into a scatter and gather phase using horizontal graph partitioning with vertex, edge, and update sets for each partition [ZCP15b]. The edge set of each partition is sorted by destination vertex to reduce the number of random writes. In [Mei+20], another PR accelerator is proposed using a two-dimensional graph partitioning approach that scales to HBM.

Centrality. In [GSP16], the authors propose a stochastic matrix function estimator written in OpenCL and apply it to the subgraph centrality problem. Subgraph centrality is another measure for importance of vertices in a graph.

Sparse matrix-vector multiplication. Fowers et al. [Fow+14] propose a compressed interleaved sparse row (CISR) format and a banked vector buffer to tackle SpMV. CISR shuffles the neighbors in such a way that for each read from memory all parallel processing pipelines in the accelerator get a new value for the row they are currently working on. Furthermore, since rows can vary in length, rows are greedily distributed to pipelines.

Subgraph matching. Jin et al. propose a subgraph matching accelerator on a CPU-FPGA platform [Jin+21]. A auxiliary data structure called candidate search tree is constructed on the CPU based on the to-be-matched subgraph and loaded onto the FPGA where the embedding enumeration is accelerated.

3.2.1.2 Binary Representation

We found articles that leverage different partitioning schemes (horizontal, vertical, and interval-shard) to aid data placement and parallelization, an optimization of CSR for BFS, and other binary representations like hardware mapping and a dynamic graph data structure.

Horizontal partitioning. In [ZCP16], a horizontal partitioning scheme is proposed with an improved data layout enabling more sequential write accesses. This is achieved by sorting the edges in each partition by destination vertex. This additionally also allows for on-chip update merging.

Vertical partitioning. Chen et al. provide a solution that features a layout improvement to the vertically partitioned data by storing edges inbound to the current vertex set and not storing an update list (i. e., directly streamed) [Che+19]. The data is shuffled to graph PEs.

Interval-shard. FPGP [Dai+16] is an edge-centric graph processing framework based on the interval-shard graph partitioning scheme. Shards $S_{i,j}$ and their corresponding inbound I_i and outbound I_j vertex intervals are fetched one after another to on-chip memory and processed by multiple PEs.

Compressed sparse row. The CyGraph architecture by Attia et al. proposes new optimizations for utilizing the memory bandwidth in parallel BFS with a custom CSR representation [Att+14]. The visitation status of vertices is encoded in the row index which is replaced by the level after visitation. This produces in-place BFS result data and leads to a lower number of memory requests overall.

Hardware mapping. In [Hue00], the complete graph with editable vertices and edges is represented as logic on the FPGA. An extension by Mencer et al. [MHH02] draws parallels to content-addressable memory (CAM) and extends the idea of hardware mapping by extending it to multi-context graph processing. Both approaches require a synthesis for each new data graph.

Dynamic. Wang et al. introduce a dynamic graph data structure that allows for updates on the graph interleaved with workload processing [Wan+21]. They apply packed memory array (PMA)-based edge array representation to CSR which leaves holes in the edge array for updates.

Other. Skylarov et al. represent the graph as an adjacency matrix and use matrix operations to calculate graph colorings [SSP06]. Wang et al. propose an edge-centric graph streaming model on an FPGA for partitioned graphs to deal with load balancing issues of skewed graphs [Wan+15; WZH16]. The kp partitions are assigned to k PEs in a pre-processing step such that every PE processes approximately the same number of edges. The graph is compressed and streamed which results in a high effective bandwidth. When the application permits graph sampling, the data volume and irregular memory accesses can be tamed with a data structure derived from CSR [TS18]. The novel data structure allows storing multiple graphs and removal of vertices and edges with a second pointer array. Xu et al. propose a service-oriented accelerator that does asynchronous processing of BFS [Xu+18]. This is very different to other approaches in that batches of so called Batch Row Vectors are streamed into the accelerator and worked on. A Batch Row Vector contains the start and destination vertex for each edge in the batch and a property for each source and destination vertex.

3.2.1.3 Queries

Queries in the context of accelerator design are about flexible chaining of operators during runtime rather than resynthesizing the accelerator for each workload. There is an instructable softcore processor design and a system mapping graph workloads to a flexible linear algebra accelerator.

Instructable processor. GraphSoC [Kap15] is a custom graph processor built from a 2D array of softcore processors connected by a packet-switched network.

Linear algebra. GraphLily [Hu+21] adopts the GraphBLAS programming interface to map graph problems to a series of linear algebra operations, namely sparse-matrix dense-vector multiplication and sparse-matrix sparse-vector multiplication. This allows to map to different graph problems during runtime without the need to synthesize a separate bitstream for each workload. Additionally, GraphLily works on HBM, delivering massive memory bandwidth.

Subgraph query processing. GraphMatch [Dan+24] is a fully flexible subgraph query processing accelerator for FPGAs that is able to switch workloads in a matter of cycles. It is implemented as a flexible pipeline of set intersection operators specifically optimized for FPGAs.

3.2.1.4 Scalability

There are no complete solutions for scalability. However, we found four articles on graph partitioning to leverage multi-FPGA setups.

Partitioning. Babb et al. presented an early work on scalability by compiling graph problems to whole arrays of FPGAs [BFA96]. Virtual wires are used in between the FPGAs resulting in a multi-FPGA computing fabric. Attita et al. propose a vertex-centric graph processing framework based on the gather-apply-scatter (GAS) principle [Att+15]. With partitioning and message passing, the workload can be distributed to multiple FPGAs. ForeGraph [Dai+17] is an edge-centric graph processing approach as a multi-FGPA extension to FPGP [Dai+16]. For p FPGAs, the graph is partitioned into p intervals resulting in p^2 shards. Each FPGA stores one interval and its outgoing p shards, additionally partitions its subgraph into sub-intervals and sub-shards, and subsequently does the same processing as FPGP on the sub-shards. Updates are propagated to the corresponding FPGA over the network. Zhang et al. propose a graph partitioning method that tries to find a balance between minimum communication overhead due to cut edges and wait time due to partition imbalance [Zha+20a].

3.2.1.5 Consistency

The conflict management for highly concurrent systems can be problematic. We only found one article proposing a locking scheme to achieve isolation.

Locking. Ma et al. define a multi-threaded graph processing engine using a global, transactional shared memory which allows fine-granular locking with an address signature table data structure [MZC17].

3.2.1.6 Design Paradigm

We found multiple graph accelerator design paradigms: vertex-, edge-centric, hybrid, and bulk-synchronous parallel (BSP), among others. Vertex- and edge-centric describe orthogonal approaches of traversing graphs.

Vertex-centric. GraphGen [Nur+14] compiles graph algorithms from a custom domain specific language to RTL. The algorithms are built from user-defined instructions. RTL implementations of these instructions have to be provided together with an update function and a description on how those instructions combine. Weisz et al. provide programmability by using GraphGen and CoRAM in combination [WNH13]. CoRAM allows to port the accelerator architecture to both Intel and Xilinx FPGAs. GraVF [ES16] shows how to compile Migen definitions to hardware. Ozda et al. provide a solution based on a configurable architecture template for vertex-centric graph algorithms [Ozd+16]. In [Yao+18], conflicting updates on one vertex are resolved by accumulating updates in one cycle, parallelizing conflicting vertex updates, and removing sequential application of atomic protection. GraphScale [DRF22] scales the asynchronous graph processing approach on a compressed graph previously only applied to single memory channel FPGAs to higher memory bandwidths. Compared to other approaches, this leads to fewer iterations over the graph and less data movement for more efficient bandwidth utilization.

Edge-centric. HitGraph [Zho+19; Zho+18] (based on [ZCP15b; ZCP16]) compiles edge-centric algorithms to RTL. The processing logic is split up into multiple PEs processing the graph with alternating scatter and gather phases. The vertices are partitioned horizontally and buffered in BRAM and partitions are skipped if they do not contain active vertices. ThunderGP is another edge-centric graph processing system for multi-channel DDR memory [Che+21b]. In contrast to HitGraph, it partitions the graph vertically but also does synchronous graph processing with alternating scatter and gather phases.

Hybrid. Chengbo proposes a hybrid approach where there is a vertex-centric and an edge-centric module on the FPGA [Yan+20a]. A dispatcher switches the modules depending on the workload. The graph is vertically partitioned.

Bulk-synchronous parallel. DeLorimier et al. propose an accelerator, mapping the graph to sparse matrix operations executed in a BSP fashion [DeL+06]. The accelerator is split up into compute leaves with BRAMs attached. Data between the compute leaves is communicated over a network on chip. In [Ayu+18], a design methodology based on the BSP model is proposed. Common architectural features are represented as templates which are specified with user-defined functions for GAS. All data flow is handled by the template.

Other. In [BFA96], problems on directed graphs are reformulated as closed semiring problems and compiled onto multiple FPGAs. For a graph instance the edges are mapped to summations and vertices are mapped to minimum operators. Dandali et al. address long synthesis times with skeleton compilation of precompiled blocks that are adapted to the problem instance [DMP99]. One PE is used for each vertex, restricting graph size to the available FPGA resources. GraphOps [OO16] is a general graph dataflow library that provides graph-specific building blocks for the generation of FPGA designs. It includes a locality-optimized property array. A dataflow-based accelerator is proposed in [Jin+17] based on the observation that instruction-level parallelism in graph processing workloads is low and branch mispredictions pose a challenge for traditional instruction-based processor architectures.

3.2.1.7 CPU-FPGA Collaboration

We found the following graph solutions for CPU-FPGA heterogeneous platforms with different task assignments.

Socket. Bondhugula et al. propose a shortest-path hardware kernel communicating with the CPU via a shared memory region [Bon+06a]. Another part of the solution is a graph data layout for the CPU to reduce cache misses. Umuroglu et al. propose an approach that distributes BFS iterations between CPU and FPGA: the iterations with few active vertices are performed on the CPU while the other iterations are performed on the FPGA [UMJ15]. Similarly, Zhou et al. address the respective drawbacks of vertex- and edge-centric graph processing by switching between the paradigms during execution [ZP17]. The graph is horizontally partitioned (cf. [ZCP16]) and the paradigm is chosen individually for each partition in each iteration based on its active vertex ratio. Partitions with few active vertices are processed by the CPU in a vertex-centric paradigm while partitions with many active vertices are processed by the FPGA in an edge-centric paradigm. Wang et al. propose a general graph processing approach with a novel worklist (priority queue) based graph computation and software scheduler (reorders vertices to be processed) [WHN19]. The FPGA inserts work items (vertices) into a pre-scheduled queue and the CPU reorders them into a scheduled queue. In [OAA21], the authors propose a heterogeneous CPU-FPGA graph processing system where the scatter phase of a synchronous scatter-gather graph processing approach is implemented on the FPGA. The graph data is held in main memory and the FPGA is connected with a cache-coherent interconnect. Additionally, the work utilizes the work stealing approach presented in [AOA20].

Near-data. ExtraV [Lee+17] proposes graph virtualization. The CPU accesses the data through a cache coherent FPGA attached to an SSD storing the graph. Transparently to the CPU, the FPGA applies compression and multi-versioning to writes and decompression and filtering to reads.

3.2.1.8 Memory Access

For memory access, we found solutions for request merging, custom caching, custom data placement, and HBM, among others.

Request merging. A CAM-based approach for the BFS memory access problem is proposed in [Kho+18]. The solution is an architecture-aware software graph clustering algorithm that reduces bandwidth requirements for random requests to visited flags. The clustering is applied as an offline pre-processing step. A memory unit merges multiple requests (cache for storing recently checked flags implemented with CAM). Asiatici and Ienne [AI21] propose utilizing a miss-optimized memory system instead of scratch pads or caches to coalesce memory accesses by cache line as much as possible before making requests. The proposed memory system allows for tens of thousands of non-blocking misses to maximize data reuse when data is actually fetched from memory.

Caching. Zhang et al. propose a map-reduce based BFS approach on hybrid memory cube (HMC) memory [ZKL17]. With a performance model, they identify the bottleneck as scanning the bitmap of the current frontier. Thus, a second caching layer for the bitmap is introduced where one bit in the caching layer represents the aggregate of many bits in the RAM bitmap. FabGraph [Sha+19; Sha+20] is an extension of ForeGraph [Dai+17] by two-level vertex caching (level L1 attached to pipelines and shared L2; L1 only communicates with L2). The vertices of the current graph partition are stored in L2 and replaced in Hilbert order such that vertices can be used for multiple graph portions.

Data placement. Zhang et al. address the problem of redundant memory accesses caused by high-degree vertices for graph traversals [ZL18]. They correlate vertex degree with data access frequency and propose degree-aware data placement and degree-aware adjacency list compression (Exp-Golomb variable length coding) combined with hybrid traversal approach on HMC memory.

HBM. An extension of GraphScale [DRF23a] scales the system to up to 16 channels of HBM memory. Scaling is limited by achievable clock frequency and resource utilization of the hardware platform.

Other. Betkaoui et al. address stalling pipelines caused by memory access latency through a memory crossbar to share off-chip memory [Bet+11]. The solution issues many parallel memory requests and decouples memory access and execution units. Ni et al. accelerate BFS by applying horizontal partitioning allowing to distribute the graph and its associated metadata over multiple memory channels [Ni+14]. In this way, multiple PEs can traverse the graph in parallel utilizing a high memory bandwidth. Upon level synchronization, active vertices are exchanged between the PEs. A memory access improvement for vertex-centric graph processing is given by Yan et al. [Yan+19]. Random memory accesses are sequenced and graph pruning is applied to prevent ongoing traversal from the leaves. The solutions further includes an online pre-processing step during the apply phase when bandwidth is under-utilized. Wang et al. propose a scheduler for conflict-free scheduling of edges to parallel processing pipelines [Wan+20a]. It is assumed that per edge one source vertex has to be read from a BRAM scratch pad which may cause stalls when multiple pipelines try to read a value from the same scratch pad bank. The authors present a preprocessing step that splits up the graph adjacency matrix into tiles and schedules them to processing pipelines without conflicts as far as possible.

3.2.1.9 Performance Model

Performance models are about understanding effects of design decisions on a conceptual level to aid decision making. We found the following solutions on modeling parallelism, lower bounds, and memory access patterns.

Parallelism. Bondhugula et al. define a performance model for parallelism in two orthogonal parameters B number of PEs, and l denoting the number of operators in each PE (process l elements at once) [Bon+06b]. Then constraints for different FPGA resources are modelled for the two parameters. Shao et al. model the runtime of their design as the sum of the time of the vertex transmission and the time of the edge streaming [Sha+19]. Both are modeled dependent on the internal parallelism parameters of their design. The model is used to determine the size of L1 and L2 caches in the design.

Lower bound. FPGP [Dai+16] finds the optimal number of PEs by modeling the runtime as the maximum of the time spent on interval loading, edge loading, and edge processing since those can be overlapped. ForeGraph [Dai+17] extends this model by also including time to load intervals from other boards in their multi-FPGA setup and compares theoretical performance against other systems. Zhang et al. specify a performance model that is a slight variation of the network model [ZKL17]. The memory system is represented by the packet size, packet overhead, bandwidths, and internal latency of memory. A lower bound performance model for vertex-centric graph processing on multiple FPGA systems is proposed by Engelhardt et al. [EHS18]. The solution includes an architecture generator for multiple FPGAs with an application kernel and fitting dataset.

Memory access. To address some of the shortcomings (e. g., mostly closed source and opaque system parametrization) of current benchmarking practices in graph processing, Dann et al. propose a simulation environment [DRF21b] based on modelling memory access patterns of graph processing accelerators to approximately estimate their runtime on a unified benchmark. The work in [DRF21a] expands on this with more simulated systems and deeper analysis of important performance factors and design decisions.

3.2.2 Document

The literature on FPGA-accelerated document database systems focuses on XML. While JSON is the predominant document format in commercial document database systems (cf. Sect. 3.1.1.2), we did find few solutions in the literature. In the following, we will discuss solutions we found to the system aspects for the document non-relational database system class.

3.2.2.1 Operator

In the literature, we found operators for XML parsing, document filtering, XPath evaluation, Twig, and JSON parsing, which we subsequently discuss.

XML parser. In [DNZ10], Dai et al. propose a solution that leverages recurring idioms in XML processing (one-to-one string match, one-to-many string membership test, one-to-many string search), and a speculative pipeline structure skewed to provide high throughput in the common case and only stall the pipeline for rare edge cases. The FPGA is placed close to the network on a SmartNIC. An alternative approach by Sidhu implements tree automata as a pair of a lexical and a tree automaton where the states of the lexical automaton form the transitions of the tree automaton [Sid13]. Huang et al. propose a sliding window XML parsing accelerator [Hua+14]. They assume that the XML is valid and based on that can process multiple non-delimiter characters in one cycle. Delimiter characters are still processed one by one.

Filter. Chalamalasetti et al. [Cha+12a] provide an implementation of document filtering with document scoring against topic profiles. The work mainly improves the bloom filter design of [VAM09]. The new design leverages multiple banks and reduces contention on these.

XPath. Mitra et al. propose a solution for XPath-based filtering of XML documents by mapping the XPath queries to regular expressions [Mit+09]. These expressions are clustered by common profile prefixes and mapped to FPGA state machines (one per XPath). A global stack is used for the inherent parent-child relationships. In [EI10], publish-subscribe systems are extended to become an XML broker using an XPath processor. The article also provides a hardware-based XML parser. Moussalli et al. [Mou+10] address the challenge of recursive XML filtering. For that, each XPath is mapped into a stack whose width matches the XPath depths in bits and the height corresponds to the depth of the document. Open tags are handled as push events and close tags as pop events.

Twig. The same authors propose an FPGA-based solution for twig matching on XML documents based on [Mou+10] combined with a dynamic programming approach [Mou+11]. However, the maximum tag length supported is 2.

JSON parser. Peltenburg et al. [Pel+21] propose a JSON parser generator for accelerators that take a document in a predetermined structure and return it in the Apache Arrow format. However, to support multiple document formats this requires generating and synthesizing multiple parsers which is not feasible in the database context. Therefore, PipeJSON [Dan+22] implements a full JSON parser able to parse arbitrary documents on FPGAs. It effectively utilizes the pipeline parallelism achievable with FPGAs to parse at line speed.

3.2.2.2 Queries

For document stores, two different solutions for microsecond reprogrammable state machines as integral parts of string matching were proposed. This does not allow query processing in itself but flexible chaining of operators which can be used for query processing with only a query parser and optimizer missing.

Skeleton automata. Teubner et al. propose the idea of skeleton automata [TWN12] as a fixed finite state automaton structure with parameterized transitions, allowing dynamic workload changes in microseconds, instead of long synthesis and (partial) reprogramming of the FPGA.

Other. ZuXA [Lun+04] implements a programmable state machine with a hash index data structure on a rule table and a clustering scheme for very large automata that is applied as an XML acceleration engine.

3.2.2.3 CPU-FPGA Collaboration

For document database systems, we found three collaboration schemes for data movement in hybrid CPU-FPGA systems: socket, near-data, and PCIe.

Socket. Vanderbauwhede et al. address the challenge of power consumption of information filtering on streams of documents with a multi-FPGA setup [VAM09]. A CPU places the document stream into main memory from where it is directly fetched by the FPGAs. An on-chip BRAM Bloom filter is used to quickly discard irrelevant documents before the documents are matched against profiles stored in a hash table data structure in on-board memory.

Near-data. The XLynx system [TWN13] by Teubner et al. provides a solution for hybrid CPU-FPGA XQuery processing with dynamic XML projection. The FPGA implements the same XML projector as in [TWN12] placed into the data path between the document server and XQuery engine such that data is filtered before being queried, effectively reducing load on the XQuery engine.

PCIe. In [Van+13], previous work on document filtering [Cha+12a; VAM09] is advanced by embedding it into a CPU-FPGA system. The CPU handles parsing the network document stream passing it to the FPGA using separator words between documents. Additionally, the words are dictionary encoded.

3.2.3 Key-value

The most recent non-relational database system research on FPGAs concerns key-value, arguably the conceptually simplest class. We subsequently present the system aspect solutions for key-value database systems.

3.2.3.1 Operator

Solutions are found for insert, hash, and LSM compaction operators.

Insert. Liang et al. address the unpredictable insert performance of Cuckoo hashing. Cuckoo hashing avoids hash collisions by computing multiple hashes per key and reinserts one key-value pair when all hash positions are already occupied [Lia+16]. The to-be-reinserted pair possibly triggers another pair to be reinserted into the hash table stalling naive pipelines. The proposed solution splits up the pipeline into an insert and reinsert pipeline.

Hash. Fast key-based value access requires reliable hash-functions like Murmurhash2 [App16] that are missing on FPGAs. Liu et al. contribute an implementation of Murmurhash2 for FPGAs with different kernels for different key sizes that are applied through dynamic reconfiguration [Liu+19].

LSM compaction. For tree-based key-value stores, LSM compaction may be a big performance factor. Sun et al. propose an FPGA-based compaction engine for the merging of pages during inserts [Sun+20b]. The design is integrated with LevelDB. Similarly, Zhang et al. [Zha+20c] propose FPGA-acceleration of LSM compaction. They describe in more detail how FPGA-accelerated LSM compactions are done in X-Engine [Hua+19], a fully fledged key-value store. Additionally, they also build a theoretical model for analysis of performance.

3.2.3.2 Binary Representation

The predominant binary representations are based on hash tables and B+ trees but there is also a CAM implementation.

Hash table. Istvan et al. leverage the FPGA's scalability and energy efficiency for a hash table implementation sustaining 10Gbps by implementing a sophisticated pipeline with concurrency control and separate key-hash and value store [Blo+13; Ist+13]. Collision handling is done with buckets by chaining fixed length pre-allocated memory regions for a tradeoff between probability of collisions and memory bandwidth. Tong et al. present a hash table with one operation per clock cycle throughput [TZP15]. They propose two hash table access schemes. The first one provides multiple slots for each hash value to reduce collisions where each operation scans all slots. For bandwidth limited deployments, instead of scanning all slots, a second hash function decides the slot to work on similar to Cuckoo hashing. The tradeoff is again between probability of collisions and memory bandwidth. FASTHash [Yan+20b] provides even higher throughput by processing p queries in each cycle. This is achieved by using p parallel, data-independent PEs with eventual consistency of updates. The hash table is split up into p partitions each owned by one of the PEs. Each partition is replicated to all PEs but only the PE that owns it inserts new values into it. Based on this work, Zhang et al. propose a hash-table design on XOR-based memory in [Zha+20b]. This XOR-based memory requires the data to be stored in on-chip BRAM at a benefit of guaranteeing a configurable parallel number of requests irregardless of the particular access pattern.

Content-addressable memory (CAM). Lockwood et al. propose a CAM-based, network-attached solution [LM15]. They define their own message format for optimized hardware parsing of requests. The key-value data is either stored in

BRAM or DRAM while only BRAM guarantees low latency access and the value addresses are looked up with an emulated CAM.

B+ tree. Yang et al. [Yan+21] propose a heterogeneous key-value store on a CPU-FPGA platform based on a B+ tree data structure with hash table leaves. The FPGA ingests requests over the network and implements a B+ tree used to dispatch requests to hash tables in CPU memory.

3.2.3.3 Multi-tenancy

Multi-tenancy is about performance and data isolation between multiple users working concurrently on a system.

Token bucket. Istvan et al. achieve this by defining traffic shapers with token buckets (inspired by similar concepts in networking) and introducing tenant-specific registers for temporary query data [IAS18].

3.2.3.4 Scalability

For scalability of key-value stores, there are solutions on data replication for increased read throughput and data partitioning to aid work distribution.

Replication. The Caribou system [ISA17] addresses scalability and fault-tolerance with data replication over a cluster of FGPA nodes. The FPGAs are deployed as a distributed storage layer in the storage nodes and allow for operator push-down (full scan and value predication) for near-data processing. For scalability, key-value pairs are replicated between the nodes such that read requests can be served by any node in the storage layer.

Partitioning. FULL-KV [Qiu+20] is a network-attached accelerator for CPU-FPGA hybrid systems, extending [Qiu+18] to two nodes. The key-value store is partitioned to the nodes and requests are routed by a proxy. BlueCache [Xu+16] acts as a caching layer of distributed network-attached FPGAs. For operation with BlueCache, all application servers are equipped with a PCIe-attached FPGA that is connected to the other FPGAs via network and Flash for data storage. Each CPU collects key-value store requests and passes them to its accelerator card in batches. After parsing, requests are routed to the FPGAs containing the data and answered there.

3.2.3.5 Availability

Availability is about fault tolerance meaning responsiveness even when nodes fail. In the literature, this was achieved with replication.

Replication. Besides its contribution to scalability, Caribou [ISA17] provides availability through replication. Writes are replicated from a master node to all other nodes in the storage layer. Data is still available when one node fails.

3.2.3.6 Consistency

For the consistency system aspect, there is one solution on isolation with MVCC, thus covering only one facet of consistency.

Multi-version concurrency control. Ren et al. define and implement MVCC support on top of [Qiu+18] by storing a version-value B-tree for every key [Ren+19]. They also define atomic operations like compare and swap, compare and get, and predecessor version.

3.2.3.7 CPU-FPGA Collaboration

We found near-data solutions with the FPGA between CPU and data and solutions based on direct memory access to expand value storage.

Near-data. In [LAC14], the FPGA is used as an in-line accelerator for Memcached acceleration processing 96% of the requests. The CPU is only used as a fallback. Based on [Qiu+18], Xie et al. design an FPGA distributed memory proxy with four pipelined data paths and a pipelined consistent hashing processor [Xie+19]. The orchestrator for the key-value store reaches up to 100Gbps.

Direct memory access. Li et al. propose a network attached key-value store based on SmartNICs that accesses the main memory over PCIe DMA [Li+17a]. The problems of the CPU cache hierarchy and FPGA low storage capacity are addressed by Qiu et al. [Qiu+18]. Similar to [Li+17a], the hash table is in on-board memory and the data in main memory (accessed with PCIe DMA). The solution features novel memory allocation and fragmentation schemes.

3.2.3.8 Memory Access

For key-value stores, we found solutions for Bloom filters, Flash, and HBM.

Bloom filter. Cho et al. propose a key-value store with cuckoo hashing and decoupled hash table and values [CC14]. A Bloom filter is used to control hash table read access and thus reduce the amount of memory requests.

Flash. Flash storage was proposed as a viable storage medium for values by Blott et al. [Blo+15]. Values are much larger in size than keys and are only very selectively accessed after their address has already been found over the hash table. Thus, storing values in Flash storage allows much larger amounts of data to be stored and persisted at the same time. Blott et al. scale out a Memcached server to 40 Terabytes of data this way [Blo+15]. Similarly, BlueCache stores the hash table data structure as an index cache in RAM and stores the corresponding values on the FPGA-attached Flash storage [Xu+16].

HBM. A parallel hash table design is proposed in [YKP20] that supports scaling to HBM. Since the many memory channels of HBM may lead to complex memory interfaces with a monolithic design, each channel is assigned its own processing engine (PE) allowing for decoupled, parallel processing of requests.

3.2.3.9 Performance Model

We found one solution modeling the theoretical maximum performance and minimum latency of key-value FPGA accelerators.

Lower bound. Qiu et al. examine the theoretical performance of their system based on the network performance as the lower bound [Qiu+20]. They split up their analysis by put and get operator.

3.2.4 Synthesis and Discussion of System Aspects

The system review in Sect. 3.1 resulted in a taxonomy of relevant FPGA and non-relational database system aspects (cf. Fig. 3.2) that guided the literature analysis that addresses research question RQ1.2 *"Which solutions and gaps exist in current research on non-relational FPGA acceleration?"*.

We found a large body of work on general graph operators (e. g., BFS) and binary representation in the HPC context. While HPC has different motivations and constraints to database processing, the work denotes a starting point for the graph database research. Notably, when it comes to database-specific system aspects like queries (e. g., GraphSoC [Kap15]), scalability (most notably ForeGraph for multi-FPGA graph traversal [Dai+17]), and consistency (e. g., simple locking

[MZC17]) only few solutions are provided. For the FPGA system aspects, there is a large body of work on design paradigms (e. g., HitGraph [ZCP16; Zho+19; Zho+18]), CPU-FPGA collaboration (e. g., most notably ExtraV [Lee+17]), memory access, and performance models. No solutions were found for availability, multi-tenancy, and security.

In the document class, there are only few solutions for functional system aspects. Most notable is the XLynx system [TWN12; TWN13]. Non-functional system aspects are not covered. FPGA system aspect solutions were only found for CPU-FPGA collaboration [TWN13; VAM09; Van+13].

For key-value stores, important functional topics like operators and binary representation are covered (e. g., hash tables [Ist+13; TZP15; Yan+20b]). Queries and sophisticated design paradigms are not applicable to key-value since there are only CRUD operators. The non-functional system aspects scalability, availability, and multi-tenancy are mainly studied in the Caribou system [IAS18; ISA17]. Further solutions are provided for consistency (i. e., isolation with MVCC [Ren+19]) as well as CPU-FPGA collaboration, memory access, and one performance model. We found no solutions for security.

In summary, we only found fully-featured, FPGA-accelerated non-relational database system for key-value databases, while for the other non-relational database system classes, most solutions are focused on purely functional system aspects in the context of HPC. For non-relational database systems, these solutions would have to be adapted to databases to be usable. There were only few non-functional solutions provided. However, abstract from databases, there is a body of work relevant to especially the non-relational aspects of non-relational database systems that we could not include in this chapter due to brevity. For example, there are papers on virtualization of FPGAs [KRA20; Vai+18] (e. g., for dynamic operator switching or multi-tenancy) and secure engine design [ABK14; Huf+08; CG03] (addressing the non-functional aspect security). Additionally, solutions could possibly be adapted between domains or from CPU and GPU systems to augment the gaps in Tab. 3.2. However, this has not been studied in detail yet.

3.3 FPGA-accelerated Non-relational Database System Patterns

This section provides several perspectives on non-relational database system class commonalities, addressing research question RQ1.3 *"Which reoccurring patterns in the literature guide the design of FPGA-accelerated non-relational database systems?"*. We revisit the current state in research (cf. Sect. 3.2) and discuss overarching patterns which we found for system architecture of non-relational database systems. We arrive at six key insights while looking at how to design an FPGA-accelerated non-relational database system starting from the need of accelerating a system, over its design and implementation, and finally the evaluation of the accelerator's impact:

1. For key-value database systems, FPGAs are most useful to accelerate the communication layer while graph and document database systems benefit the most from fused operator and memory access acceleration.
2. There are four fundamental patterns of FPGA placement governing data movement in the system: extension card, near-data, SmartNIC, and socket.
3. The accelerator task in combination with the workload characteristics of the non-relational database system class can be used to decide the FPGA placement.
4. Workload switching is a difficult problem with strategies on a spectrum between full reconfiguration to flexible query processing.
5. In the considered literature, there are six memory access optimization patterns universal across non-relational database system classes.
6. Portable, relevant benchmark suites (for single classes or overarching) that cover all necessary artifacts are missing for robust justification of accelerator usage decisions.

3.3.1 Accelerator Task Categories

When dealing with a concrete system, either an accelerator is added to an existing system or becomes relevant to the design of a new one. However, in both cases, one has to decide whether an FPGA is suitable. As a first step, we analyze the problems typically solved by FPGAs in the different non-relational database system classes and task categories that are well suited to FPGAs. Notice, however, that in this section, we cannot represent all possible motivations and tasks that are encountered by practitioners.

Graph For graph processing, FPGAs are mainly used to offload operators because of inefficiencies in the cache hierarchy and coarse-grained memory access of CPUs resulting from the inherent irregular memory access patterns of graph workloads (cf. [Att+14; Dai+17; Zho+19]). An FPGA helps to alleviate the problem of irregular memory accesses through custom memory controller design and full control of data placement in on-chip memory (cf. Sect. 3.3.3.2).

Document FPGAs for document processing are mainly used in the literature as bandwidth amplifiers for the CPU. Much of document processing is parsing and filtering with large data movement costs putting heavy load on the CPU even tough a lot of the data is discarded and pollutes the cache hierarchy. Thus, FPGAs can be used as a flexible stream processing accelerator in the data path to the memory (e. g., [Cha+12a]), disk, or network (e. g., [DNZ10]). Sometimes, however, the CPU is completely bypassed and the FPGA is used as a standalone accelerator in the network (e. g., [TWN12]).

Key-value For key-value stores, the literature mainly shows two schemes for motivation of FPGA usage. Either a single operator instrumental to key-value stores is accelerated (e. g., insertion [Lia+16] or hashing [Liu+19]) because the CPU does not meet the latency requirements, or building a full system is motivated by large roundtrip latencies of CPUs from the network through the operating system network stack and back (e. g., [ISA17; Li+17a; Qiu+20]).

Summary The two biggest problems that FPGAs solve for the different non-relational database system classes are: *(1)* data movement from peripherals (e. g., network or disk) to the CPU and *(2)* memory access inefficiencies caused by the fixed CPU cache and memory access architecture. Figure 3.3 shows a simplified version of the system architecture from Fig. 3.2. The tasks that FPGAs might solve in an non-relational database system to address these problems fall into three categories: operator, data access, and communication layer acceleration.

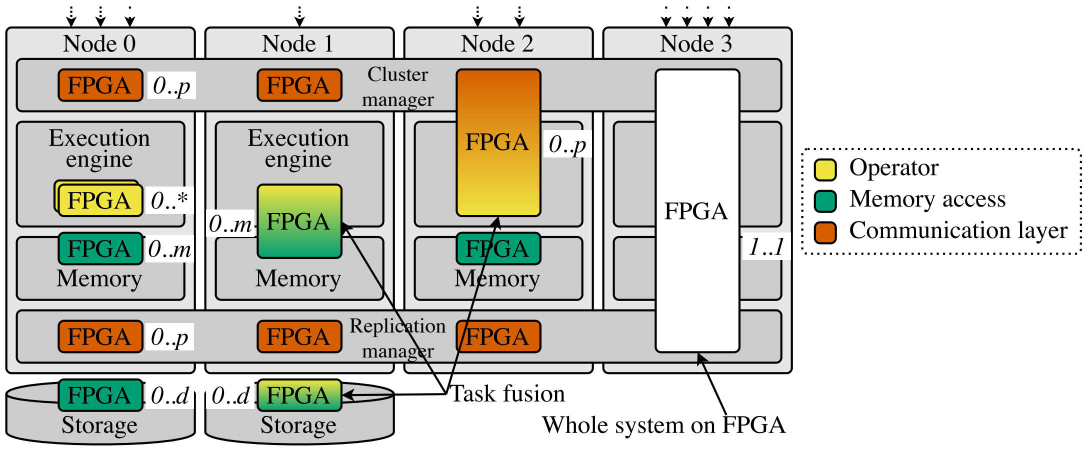


Fig. 3.3 Potential FPGA tasks in non-relational database systems.

Operator acceleration focuses on improving performance for one or multiple operators. A lot of the literature focuses on this task category with implementations of specific operators. Particularly interesting for non-relational database system are accelerator implementations in the context of hybrid CPU-FPGA systems (e. g., [Bon+06a; JSL11; VAM09; Van+13; WHN19]). In data-centric applications, the FPGA can take pressure off the CPU with data access acceleration. One already existing example is graph virtualization, where non application-specific access patterns are accelerated on an FPGA near storage [Lee+17]. Placing FPGAs in the communication layer is another promising option. One example is a proxy layer for key-value stores where the request router (cf. Fig. 3.2) is placed in an FPGA outside the other nodes which routes traffic to the correct CPU nodes [Xie+19].

On Nodes 1 and 2 in Fig. 3.3, we show possible combined acceleration of tasks that we call *task fusion*. Task fusion is possible when the resources on the FPGA fit both tasks. Task fusion should be done if more than one task benefits from FPGA acceleration. The tasks combined on one FPGA may even accelerate the overall system more than if they would be accelerated separately because data movement costs and thus overhead are reduced.

In the relational database literature, we found examples of operator push down where operator acceleration and data access are fused into one (e. g., [Fra+11; WIA14]). This worked especially well for filter operators pushed to the FPGA that reduce the amount of data communicated to the CPU. This kind of accelerator uses its close proximity to memory to reduce the data load on the

CPU. The newly presented Enzian system [Alo+20] also enables this with a cache-coherent attachment of the FPGA and opens up questions of near-data processing where the FPGA is used for on-the-fly conversion of binary representation. One prominent example of fusing operators with the communication layer at data center scale is the Microsoft Catapult project [Put+14]. There, servers push document classification workloads to a communication layer of multiple FPGAs connected to each other. Another example from the key-value literature is [Qiu+18]. They use the FPGA as an entry point from the network and do pre-processing of queries in the FPGA while – due to size – storing the actual data values in the memory of the CPU.

In extreme cases, one to all nodes in the non-relational database system cluster can be mapped to FPGAs (Node 3) (e. g., [ISA17]). This works well for key-value systems (and might work for wide-column systems) since the overall system is simple enough, and also works for providing standalone services on FPGAs (e. g., [TWN13]). In this case, the FPGA has to be network-attached which saves a lot of overhead by not going through the CPU cache hierarchy and operating system network stack.

To aid the decision of how many FPGAs to put into a system, we added cardinalities to each possible FPGA task (Fig. 3.3). In the confines that are set by the system hardware, independent FPGAs could be added for different tasks. One FPGA can be added for each network port, memory subsystem, and disk in the system (p is number of network ports; m is number of memory subsystems; d is number of disks). For operators, there is no such restriction. There can be as many FPGAs as fit into the hardware system. Insight 1 answers which problems FPGAs are well-suited for:

Insight 1. *For key-value database systems, FPGAs are most useful to accelerate the communication layer while graph and document database systems benefit the most from fused operator and memory access acceleration.*

3.3.2 Accelerator Placement Patterns and Decision Tree

In this section, we discuss FPGA placement patterns in the context of a single cluster node. In the literature, we discovered four FPGA placement patterns which we discuss before we show how to chose a placement based on the task (cf. Sect. 3.3.1) and characteristics of the workload.

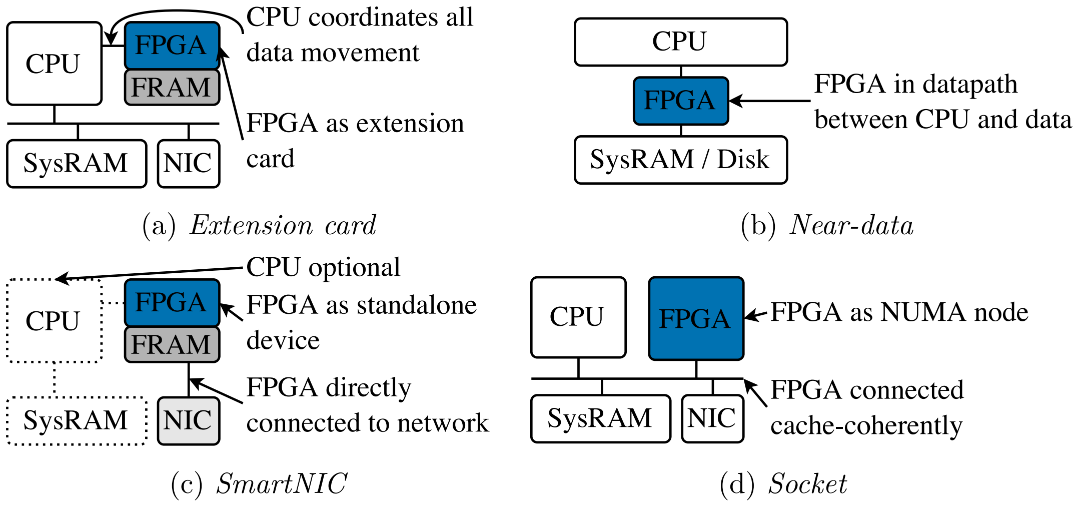


Fig. 3.4 Placement patterns.

3.3.2.1 Placement Patterns

We differentiate between the main memory of the overall system (SysRAM) that is possibly used by a cache-coherently attached FPGA but mainly used by the CPU and RAM directly attached to the FPGA on the board (FRAM).

For the *extension card* accelerator placement (Fig. 3.4(a)), the FPGA is attached only to the CPU (e. g., over PCIe) and an on-board FRAM much smaller than SysRAM. Input data is directly written to FRAM by the CPU and execution is triggered by the CPU. The FPGA works on the input data in FRAM, and the results are transferred by the CPU to the SysRAM on notification of the CPU by the FPGA. This placement only allows master-slave setups, and thus can introduce a lot of overhead for acceleration because the FPGA cannot move data in the system on its own, and CPU cycles are wasted on data movement and orchestration. The FPGA-accelerated in-memory database survey [Fan+20] shows this placement as *IO-attached accelerator*.

The second placement we found in the literature is the *near-data* placement (Fig. 3.4(b)). It is defined by the way the FPGA is inserted into the data path between the CPU and the SysRAM or disk. In this way, the FPGA provides an interface to the CPU to interact with the underlying resource. In a more restricted way, [Fan+20] define this as a *bandwidth amplifier* which decompresses data, however this placement can accelerate more workloads than just decompression, e. g., filtering and binary representation conversion. In [MT09], there is a similar placement where the FPGA is placed in the data path between disk and CPU.

Figure 3.4(c) shows the *SmartNIC* placement where the FPGA is directly attached to the NIC. This placement option may even completely eliminate the CPU in the system if there are no tasks besides what is implemented on the FPGA. The SmartNIC placement optimizes for low latency of the overall system by saving multiple round trips through the operating system kernel on the CPU.

In emerging systems (e. g., [Alo+20]), the FPGA may be placed as a *socket* (Fig. 3.4(d)) with a cache-coherent access to SysRAM. The three previously discussed placements can be represented with the FPGA being a socket with little overhead. However, the socket placement also enables new work distribution strategies where the CPU does not have to coordinate execution on the FPGA and data movement. In [Fan+20], this placement option is called *co-processor*.

3.3.2.2 Placement Decision

FPGA placements discussed in the literature can be reduced to the four fundamental patterns. Based on these, Fig. 3.5 shows a decision tree guiding the practitioner towards choosing an accelerator placement pattern depending on task and workload properties. Additionally, we added CPU- and GPU-based systems as alternative system architectures.

For tasks that incur large data movements from either memory, disk, or network, we have introduced shortcuts (shown as blue arrows) to the near-data and SmartNIC placements, respectively.

For workloads that do not exhibit massive parallelization opportunities, we do not see much potential in applying an accelerator. Thus, this leads to adding more CPUs to the system or alternatively adding more nodes to the cluster. For compute-bound problems with structured parallelism, meaning large numbers of homogeneous threads running in parallel, we would choose GPUs over FPGAs because they are specifically made to handle these workloads [Lin+08]. Similarly, for tasks with heavy reliance on unstructured floating-point operations, we would most of the time advise against using an FPGA as an accelerator because the DSP floating point units on the FPGA will quickly become the bottleneck.

For compute-bound problem instances that are not better suited to GPUs or multi-CPU, an *extension card* accelerator approach is chosen. The data movement is a big source of overhead in this placement model such that it only works for compute-bound problems where data movement costs, on the slow link between CPU and FPGA, are negligible compared to the duration of the

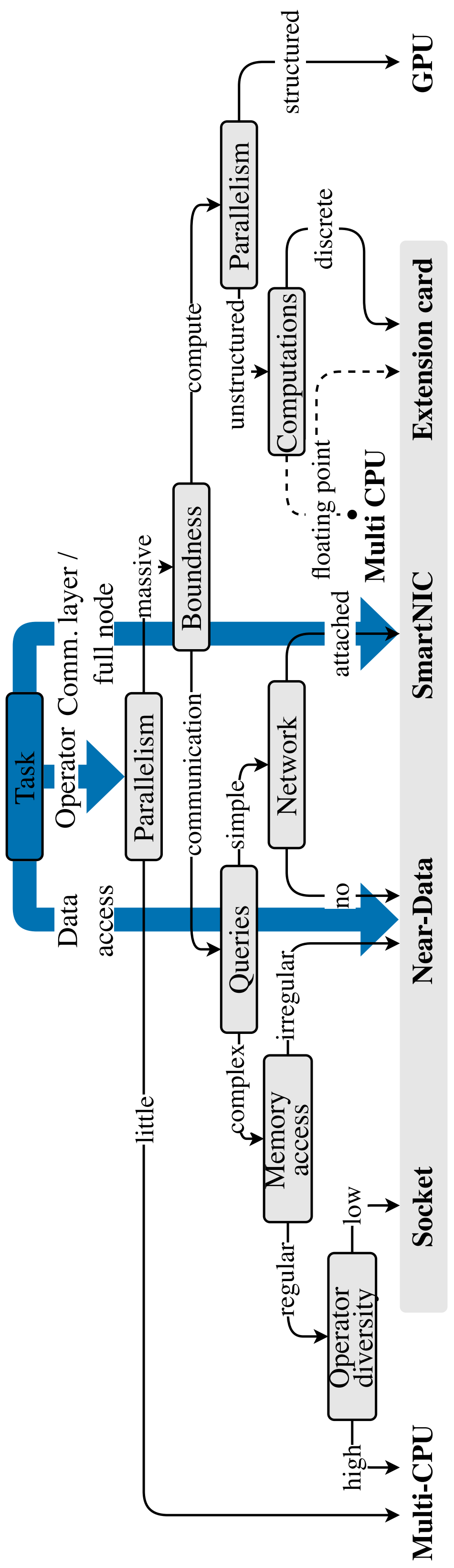


Fig. 3.5 Accelerator placement decision tree.

computation. The extension card accelerator approach is implemented by most of the graph literature (e. g., [Dai+17; Fin+19; Zho+19]). While we think that the extension card pattern could be a viable option for database systems, we focus on more significant improvements through acceleration.

In the category of memory-bound problems, we differentiate between workloads with simple operators (e. g., lookup) and workloads with complex operators (e. g., graph traversal). Simple queries are defined as a combination of few simple operators with few predicate expressions. This category includes key-value and wide-column database systems and can include document and graph database systems in certain scenarios (e. g., data provider for graph neural networks). If the database system is network-attached, we choose a *SmartNIC*. This placement was also found to be efficient in related data processing domains like data-intensive messaging [Rit17; Rit+17]. If the database system is only part of a larger architecture and not network-attached, we chose the *near-data* approach. This placement option is also chosen when there are complex queries with irregular memory accesses (e. g., in graph traversal). We think that the *socket* placement will benefit FPGA-accelerated systems especially in the database context.

As shown in Fig. 3.5, adding an FPGA to the system is not always the best strategy to improving performance of a system. For some workload characteristic combinations, we recommend multi-CPU or GPU setups. Moreover, the traditional approach of placing an accelerator in a system as an extension card is often not the best option for database systems.

3.3.2.3 Summary – Accelerator Placement

In this section, we first showed how FPGAs can be attached to the other hardware components in a system. Especially compared to a CPU, FPGAs can be placed close to the data, whether in memory, disk, or network.

Insight 2. *There are four fundamental patterns of FPGA placement governing data movement in the system: extension card, near-data, SmartNIC, and socket.*

Thereafter, we established a decision tree guiding the practitioner from the tasks (cf. Sect. 3.3.1) and the characteristics of the operators towards choosing a placement pattern. We have validated this decision tree with the systems

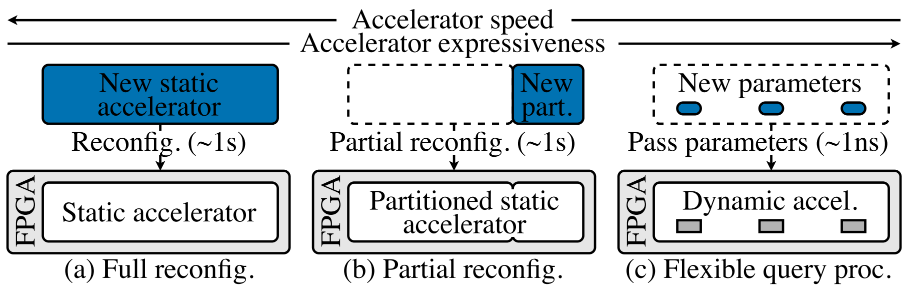


Fig. 3.6 Strategies for workload switching.

found in the system review and literature analysis by comparing our and their placement decision (not shown here).

Insight 3. *The accelerator task in combination with the workload characteristics of the non-relational database system class can be used to decide the FPGA placement.*

3.3.3 Accelerator Design Patterns

After looking at tasks and placements for FPGAs, we next discuss system and operator design. Although implementation details largely depend on specific algorithms and data structures of the non-relational database system classes, we found patterns for two critical accelerator design considerations. In the following, we introduce workload switching strategies and memory access optimization patterns common for all non-relational database system classes on FPGAs.

3.3.3.1 Workload Switching Strategies

In the queries system aspect part of the literature analysis, we found little about accelerators that are able to switch workloads without compiling a new accelerator each time. However, it is a requirement of most non-relational database system accelerators to process multiple different workloads in parallel and in very quick succession on multiple different datasets in memory, since it is not sufficient for an accelerator to improve the performance of only one workload to offset added cost and complexity. This is easy to achieve on instruction-based architectures, like CPUs, since their workloads are easily switched out by calling a different function but difficult to achieve on FPGAs since they cannot switch their architecture without significant overhead. This is shown in Fig. 3.6(a) as *full reconfiguration* where a workload switch takes seconds [PDH11].

To alleviate the overhead of full reconfiguration, the relational database community pursued *partial reconfiguration* shown in Fig. 3.6(b) where only parts of the accelerator architecture are switched out (e. g., [Owa+17; Wer+17; Zie+16]). While this works for coarse-grained functionality switching during runtime, the part that is reconfigured still is unavailable for seconds.

Thus, we advocate for a more elegant and expressive solution in what we call *flexible query processing* shown in Fig. 3.6(c). The flexible query processing strategy is based on a dynamic, parameterized or instructable, class-specific accelerator (e. g., [ISA16; Kap15; Lun+04; TWN12; TWN13]) that allows to process multiple different workloads in parallel and with only cycles switching delay on multiple data sets by just passing new parameters or instructions instead of a new bitstream. The added accelerator expressiveness might come at the loss of some accelerator speed [Teu17] but saves magnitudes in workload switching delay. The difficulty of designing such an accelerator lies in finding abstractions of high generality without introducing too much overhead that slows down performance. The examples we found above focus on a small set of class-specific primitives that are combined into a flexible accelerator.

Although we did not see implementations of it in this chapter's literature, the workload switching strategies can be applied in combination and are not mutually exclusive. For example, depending on the task, it might be beneficial to partially or even fully reconfigure the FPGA as long as it is done at a low enough frequency (e. g., if lasting workload changes are detected).

Insight 4. *Workload switching is a difficult problem with strategies on a spectrum between full reconfiguration to flexible query processing.*

3.3.3.2 Memory Access Optimization Patterns

For non-relational database systems, memory accesses are one of the most instrumental challenges to good performance (e. g., [Che+21a; DRF21b]). We identified six memory access optimization patterns in the literature shown in Fig. 3.7 that are applicable to all non-relational database system classes. Each pattern is implemented in the memory controller (endpoint to memory on the FPGA) of the accelerator design. The memory controllers performance can be

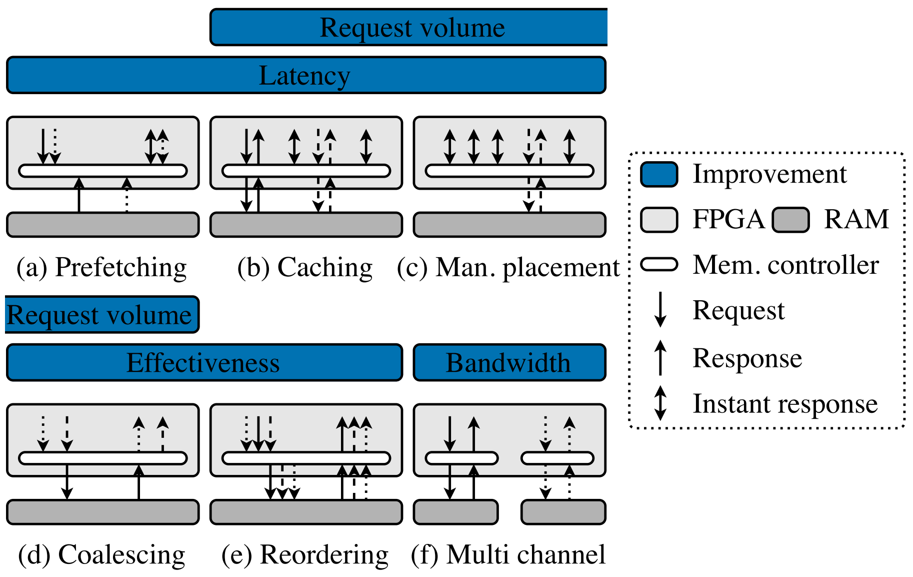


Fig. 3.7 Memory access optimization patterns (different line styles for different request-response pairs).

improved along four axes: by reducing *latency* of accesses, reducing the number of accesses (*request volume*), increasing the amount of effective data per access (*effectiveness*), and increasing raw memory *bandwidth*. In the following, we introduce these six combinable patterns.

Prefetching shown in Fig. 3.7(a) is a technique to hide memory access latency that starves processing of input data. Therefore, memory requests are issued before the data is processed to overlap computation and loading of data if the access locations are known beforehand. By the time the data is processed, it already resides on chip to be consumed. One example is partitioning the data and overlapping partition loading with partition processing [Sha+19; ZCP15b]. Another is issuing large amounts of non-blocking memory requests such that there is always data to process [Bet+12b].

Caching or automatic data placement as shown in Fig. 3.7(b) reduces high-bandwidth utilization by storing accessed values in on-chip memory (cache). If the cache is full, there is an automated policy (e. g., least recently used) replacing values in the cache with new ones [JL19]. If a value in the cache is requested, it is instantly served from on-chip memory but this only works well for workloads with strong temporal locality. One example is multi-level caching in [Sha+19]. In [WHN19], Wang et al. combine caching with reordering to increase the spatial locality of memory accesses.

FPGAs provide the practitioner with full control over what data resides in quickly accessible on-chip memory. Thus, not only custom caching techniques can be employed but critical data can be *manually placed* on the FPGA as shown in Fig. 3.7(c). This may be done for frequently accessed data structures critical to the performance of the accelerator (e. g., [Bet+12b; Lia+16; ZKL17; ZL18]). Another example is storing a highly efficient data structure like a bloom filter that allows filtering of memory accesses for presence of values in a dataset in on-chip memory (e. g., [Bec+15; CC14; VAM09]).

Coalescing as shown in Fig. 3.7(d) means merging multiple data requests into one memory access (e. g., [Kho+18]). Since modern DRAM operates on rows of memory that are in the kilobyte range, accessing single data items is wasteful. If the workload exhibits strong spatial and temporal locality, coalescing can reduce the number of memory accesses per single data item and thus decrease the request volume by simultaneously increasing the effectiveness of each memory access. In [Fin+19], this is done by having many more compute units than access units that issue many accesses enabling access units to coalesce some of the accesses. Another example is combining write requests before they are written to memory and thus reducing the overall number of writes [ZCP15a; ZCP16].

Usable memory bandwidth suffers from irregular accesses. *Reordering* of memory requests as shown in Fig. 3.7(e) can improve upon this if the workload exhibits spatial locality. This can be done online (e. g., [Yan+19]) at the cost of increased latency or offline (e. g., [ZCP15b; ZCP16]) if there is a correlation between data and memory access order.

If *multiple memory channels* are available, the memory bandwidth can be increased by distributing memory accesses over those channels as shown in Fig. 3.7(f). This is a meta-pattern that can be combined with any of the aforementioned patterns. One example of using multiple channels is placing different data structures on different channels [Lei+16; Ni+14].

3.3.3.3 Summary – Accelerator Design

While there was no focus on workload switching in the HPC-motivated literature, there has been some work on the topic especially in the document database system class. We see it as a crucial consideration in FPGA-accelerated non-relational database systems. Furthermore, the memory access optimization patterns are especially important to non-relational database systems since memory access acceleration is one of the big motivations to use FPGAs. Thus, we conclude with the following insight:

Insight 5. *In the considered literature, there are six memory access optimization patterns universal across non-relational database system classes.*

3.3.4 Justifying Accelerator Usage

In an FPGA-accelerated system, the FPGA's improvement on performance must not be evaluated in isolation on the accelerated part of the workload but in the context of the whole system. An FPGA introduces a new component into the system that entails costs having to be justified by the performance improvement. Costs occur in the form of cost of ownership (which might benefit from FPGA energy efficiency), cost of programming (long design cycles and lacking debugging capabilities) and operating a whole new hardware architecture, and data movement in the system. While the decision if the performance improvement outweighs the cost lies in the judgement of the practitioner, in this section, we will elaborate on measuring performance improvements with benchmarking and performance models for FPGA-accelerated non-relational database systems.

3.3.4.1 Benchmarks

As a guideline to good benchmarking, we follow the four criteria for class-specific benchmarks from [Gra93]. Benchmarks should be easy to understand (*simple*) and scale from small to powerful systems in the present and future (*scalable*). However, the most critical criteria (because they are the most difficult to achieve) to FPGA-accelerated non-relational database systems is that benchmarks are *portable* and *relevant* which we discuss in the following.

Benchmarking only works well when either comparing different systems running the same implementation or different implementation running on the same underlying system. Either the implementation or the system as variables have to be fixed for comparability. This especially poses a big problem for benchmarking on FPGAs since different FPGAs have very different specifications, and implementations are often tuned to one specific FPGA. We did not find any solutions for this problem in literature.

We kept track of the data sets and workloads used in the literature (not shown here) and considering the overall number of papers we found by non-relational database system class, no workload or data set is widely established especially when compared to e. g., TPC in the database literature. For key-value database systems, in the literature, benchmarking workloads are mostly just a random sequence of a subset of the three basic API functions (i. e., get, put, and delete). However, a well-formed benchmark establishes not only data sets and workloads but all of the following artifacts: *(1)* workloads (e. g., for graph: BFS, shortest path, weakly connected components) *(2)* data sets (e. g., for graph: twitter, rmat, live-journal) *(3)* class-specific performance measures (e. g., for graph: traversed edges per second (TEPS)) *(4)* benchmark reference implementation or implementation details. Thus, current performance measurements not only lack relevance but also some of these artifacts to be regarded as complete benchmarks.

A solution could be provided by existing comprehensive benchmarks that are not yet widely used in literature [Ren+17]. The YCSB program suite [Coo+10] offers capabilities for benchmarking key-value and document database systems and only recently emerged for evaluation in some key-value papers. For graph database systems, there are the LDBC Graphalytics Benchmark, LDBC Social Network Benchmark, LSQB Benchmark, and GAP Benchmark Suite. The LDBC Graphalytics Benchmark and the GAP Benchmark Suite cover kernels common in graph processing (e. g., BFS or PageRank) while the LDBC Social Network Benchmark and LSQB Benchmark cover general querying workloads and subgraph query processing. Recently, a cross non-relational database system class benchmark was proposed [Zha+18]. These benchmarks, initially designed for CPU-based systems, could also be used to benchmark new designs on FPGAs with modifications for FPGA-specific problems to make them portable.

3.3.4.2 Performance Models

As established in Sect. 3.1.2, FPGAs exhibit unstructured parallelism exacerbating comprehension of performance and algorithm complexity on an abstract performance model level. In the literature, we found different ways (cf. Tab. 3.2) to model performance of the proposed solutions breaking down to modeling pipeline and data parallelism in the face of constrained resources (logic resources and memory bandwidth). This means, the system architecture is broken down into components with known performance (i. e., pipeline steps or replicated PEs) where performance is measured in throughput of data which scales linearly with pipeline steps and data parallelism. Sometimes this is embedded into a roofline model where the performance is first capped by the amount of parallelism and later by the available memory bandwidth.

3.3.4.3 Summary – Justification

Justification in the form of workloads and datasets performing well on FPGAs is provided by the literature but the performance measurement landscape is scattered and sometimes specifically tuned to the contribution of the article. The biggest challenge, however, is the lacking portability of current benchmarks because measurements are performed on vastly different FPGA setups without accounting for e. g., different memory bandwidths. One possible solution could be performance modelling like presented in Chapter 4 which provides a comparison of multiple state-of-the-art graph processing accelerators based on a simulation environment to approximate graph processing accelerator runtimes with drastically reduced implementation effort. Still, it is sometimes unclear if performance improvements stem from better design or just better hardware. Thus, we conclude with the following insight:

Insight 6. *Portable, relevant benchmark suites (for single classes or overarching) that cover all necessary artifacts are missing for robust justification of accelerator usage decisions.*

3.3.5 Discussion – Insights

Over the course of this section, we gained six insights into building an FPGA-accelerated non-relational database system. From the motivations of the different non-relational database system classes in Sect. 3.3.1 and resulting tasks, we saw that two of the biggest challenges of current CPU-based systems are data movement and memory access. With the patterns we found for placement and memory access optimization, we guide the practitioner towards addressing these challenges with FPGAs regardless of non-relational database system class augmented with common operator switching strategies. However, performance largely depends on specific algorithms and data structures designed on a use case per use case basis. In this regard, we were not able to uncover even more inter-class structure at this abstraction level. Nonetheless, the insights we gained help practitioners to apply FPGAs to existing or newly designed non-relational database systems, regardless of data model, to take pressure off the CPU or eliminate it from the system architecture completely.

3.4 Open Research Gaps

The literature analysis in Sect. 3.2 did not only summarize many interesting solutions but showed several research gaps completely lacking solutions. In this section, we summarize and discuss the important open research gaps we think should be pursued in the near future.

Regarding the differences between the non-relational database system classes, the research gaps vary. We found that key-value database systems are exceptional in that there already exist complete non-relational database system (research and commercial) and discussion on non-functional non-relational database system aspects (e. g., [Ist20]). However, key-value database systems are arguably the simplest non-relational database system class, and their system design leaves many questions unanswered that come up for other classes. Wide-column database systems are not represented in the literature at all, but solutions from the key-value class should be applicable (cf. [DCL18]). The literature on document and graph database systems as well as existing accelerator prototypes indicate feasibility. However, the existing literature is rather HPC-specific and cannot directly be applied to non-relational database system. Thus, there are in general many gaps to be addressed towards a complete non-relational database system.

Besides these rather broad considerations, we identified several open research gaps in the course of this chapter that we discuss subsequently.

Non-functional system aspects As a broad trend in the literature, the coverage of non-functional non-relational database system aspects are an open research gap. While consistency protocols may transferred from the non-accelerated non-relational database system literature, it is broadly unclear how to provide production-grade scalability, availability, multi-tenancy, and security with an FPGA-accelerated non-relational database system. FPGAs as a relatively new processor architecture for data processing are not integrated as deeply into current systems and do, in contrast to CPUs, not have widely used operating systems providing basic functionality. Possibly some solutions can be transferred from FPGA-accelerated relational database systems.

Flexible query processing accelerators The static implementations presented in the most of the current literature need to become more flexible and easier to program to be useful in the database context. There are elegant solutions (e. g., skeleton automata [TWN12]) to be found for instructable accelerators that can process more than one rigid workload though. This follows the proposal of domain-specific architectures in [HP19] and will also largely improve the projected ratio of the workload processed by the accelerator leading to better overall performance of the system.

Benchmarks and reproducibility As discussed in Sect. 3.3.4, there are no commonly used benchmarks in the non-relational database system classes on FPGAs yet. Standardized benchmarks will be instrumental in gaining more credibility in performance claims, comparability, and justification of FPGAs as non-relational database system accelerators. Moreover, better performance measures will help uncover performance impediments in other domains like DRAM (bank parallelism utilization analyzed in [Gho+19b]).

Collaborative memory usage Data movement overhead dominates accelerated systems and narrows their potential for performance improvements. The emergence of cache-coherent attachments of FPGAs to the system main memory might alleviate this. FPGA-directed data movement and orchestration could take pressure off the CPU and also make more fine-grained acceleration possible. However, we did not find any literature on the collaborative usage of the system main memory and smart movement of data.

Bandwidth-efficient accelerators Especially for document database systems, we found deployments of the FPGA in the datapath between CPU and memory or disk to reduce the volume of data being moved to the CPU. Since graph workloads are highly memory-bound, we see big potential for a tighter integration of FPGA and memory to hide inefficiencies of irregular memory accesses.

Accelerator integration A relatively new trend in non-relational database systems are cross data model systems, i. e., systems that allow storing and accessing data in multiple data models simultaneously [LH19]. One example is OrientDB that supports polymorphic queries over graph and document data in one unified system. FPGAs could, e. g., be used near-data to transform and change the binary representation on-the-fly as data is loaded to the CPU.

Heterogeneous computing As GPUs and FPGAs get more popular as accelerators and ever more present in the data center, there will be more performance gains to be had in heterogeneous systems using multiple accelerator types at the same time. There are first works on those systems in other research areas (e. g., [HPM12]), but none in the non-relational database system literature. This kind of acceleration might be especially beneficial to non-relational database systems supporting multiple data models, where different workloads are particularly well-suited to different processor architectures.

3.5 Conclusion

FPGAs are an instrumental tool in achieving performance gains in data-centric systems in the near future. In this chapter, we open up the field of FPGA-accelerated non-relational database systems by studying and answering the research question RQ1 *"How can non-relational database systems leverage FPGA acceleration?"* with its three sub-questions RQ1.1 - RQ1.3 formulated in Sect. 1.3.

To start with, for research question RQ1.1 *"How do existing non-relational database systems utilize FPGAs?"*, we conducted a system review of commercial non-relational database systems. We found three experimental extensions to existing systems using FPGAs as accelerators [Wil17; Loc20; Pot19], showing the non-relational database system acceleration feasibility but no mainstream adoptions of FPGAs in non-relational database systems, yet. Hence, we confirm the potential of FPGAs as accelerators for non-relational database system, but also conclude that this potential is not yet realized in commercial systems. To give an answer to research question RQ1.2 *"Which solutions and gaps exist in current research on non-relational FPGA acceleration?"*, we derive a system aspect taxonomy that guides an extensive literature analysis that categorizes the research and provides an overview of existing solutions. Taking the results of the literature analysis as a knowledge base, we derived common patterns, answering research question RQ1.3 *"Which reoccurring patterns in the literature guide the* *design of FPGA-accelerated non-relational database systems?"*. Therefore, we propose easy-to-apply patterns for FPGA task definition, FPGA placement, accelerator design considerations, and benchmarking.

In summary, we provide a comprehensive introduction of FPGA acceleration and CPU offload potential for non-relational database systems and present it in a form suitable to everybody interested in the field. However, we especially regard this chapter as a guide for system architects in their decision making and a reference for researchers to guide and conduct new research.

Part I

Graph Processing on FPGAs

Chapter 4

GraphSim: Demystifying Memory Access Patterns of Graph Processing on FPGAs

For graph processing, traditional CPU-centered hardware faces performance challenges caused by irregular memory accesses and little computational intensity inherent to the workload (cf. Sect. 1.2.1) [Bes+19a; DRF23b; Lum+07]. Example 1 illustrates the effect of irregular memory accesses for breadth-first search (BFS) with an edge-centric approach. When not reading sequentially from DRAM, bandwidth degrades quickly [Dre07], due to significant latency introduced by DRAM row switching and partially discarded fetched cache lines.

Example 1. Let each cache line consist of two values, the current BFS iteration be 1 with root v_0, and e_2 be the current edge to be processed. Figure 4.1 shows an example graph with a simplified representation in DRAM memory. The graphs edge array is stored in rows r_0–r_4 and the current value array is stored in r_5 and r_6. We begin by reading edge e_2 which incurs activating r_1 in the memory and reading a full cache line. Then, we activate r_5 and read the first cache line containing v_0 and v_1, but only use v_0. Finally, we activate r_6 to read v_5 and write the new value 1 to the same location, while wasting bandwidth of one value on each request (i. e., reading and not writing the value of v_4 respectively).

FPGA-based graph processing accelerators (cf. Sect. 3.2.1) emerged as one possible solution. FPGAs enable unique memory access pattern and control flow optimizations with their custom-usable on-chip memory and logic resources

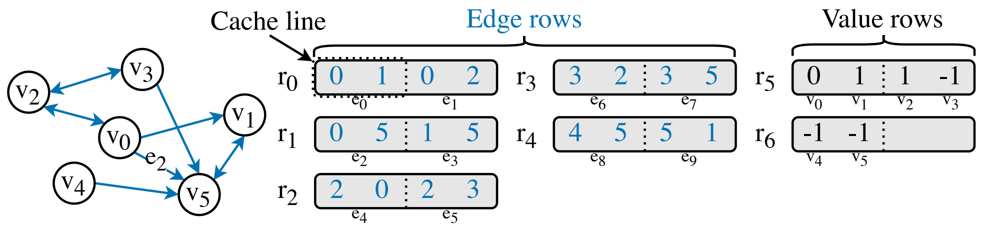


Fig. 4.1 Illustration of irregular memory accesses for BFS.

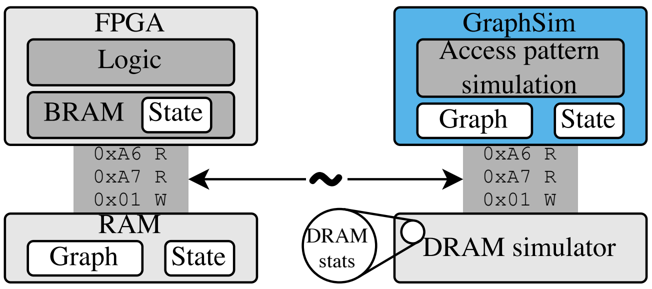


Fig. 4.2 Memory access pattern simulation approach.

that are not constrained to a predefined architecture. Additionally, modern memory technologies like high-bandwidth memory (HBM), available for FPGAs, help alleviate the pressure on the memory subsystem. While FPGA-based graph processing accelerators show good results for irregular memory access pattern acceleration (e. g., [Yao+18; Zho+19]), programming FPGAs is time-consuming and difficult compared to CPUs and GPUs where the software stack is much better developed [Aba+19; BRS13]. Additionally, there are deficiencies in benchmarking of these accelerators due to a multitude of configurations regarding available FPGAs, memory architectures, workloads, input data, and the lack of accepted benchmark standards (cf. Sect. 3.3.4.1). This makes it difficult to assess the implications of different design decisions and optimizations, and – most importantly – to compare the proposed accelerators [DRF23b; Ahn+15]. This leads us to the two main challenges in the field: (i) time-consuming and difficult development of graph processing accelerators, (ii) reproduction and comparison of accelerators is hard due to differences in hardware platforms and benchmark setups and thus hinders understanding of accelerator design decisions.

To address challenges (i) and (ii) and answer research question RQ2.1 *"Which crucial graph processing accelerator properties contribute to good performance?"*, we propose a simulation environment for graph processing accelerators GraphSim based on the idea in Fig. 4.2. GraphSim is a methodology and tool to quickly

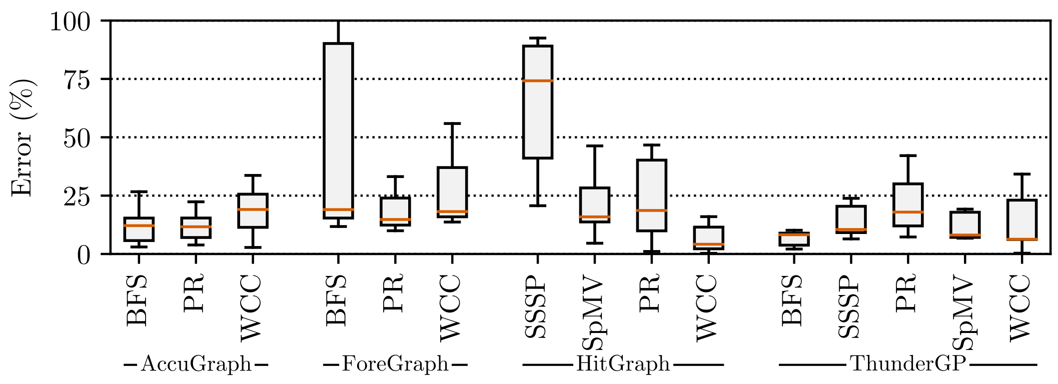


Fig. 4.3 Average simulation error by accelerator and workload.

reproduce and compare different accelerator approaches in a synthetic, fixed environment. On a real FPGA, the on-chip logic implements data flow based on on-chip (in BRAM) and off-chip state and graph data in the off-chip DRAM. However, we observe that the access to DRAM is the dominating factor in graph processing performance. Thus, we only implement an approximation of the off-chip memory access pattern in GraphSim, based on the graph and state independently of the concrete (difficult to implement) data flow on the FPGA, and feed that into a DRAM simulator. While the performance reported by GraphSim may not exactly match the real performance measurements, it provides a good approximation at largely reduced implementation cost and we see a high potential to better understand graph processing accelerators. Our simulation approach significantly reduces the time to test new graph processing accelerator ideas and also enables design support and deeper inspection with DRAM statistics as well as easy parameter variation.

In Sect. 3.2.1, we found many graph processing accelerator approaches that we may simulate with GraphSim. Based on criteria like reported performance numbers on commodity hardware and sufficient conceptual details, we chose four state-of-the-art systems – namely AccuGraph [Yao+18], ForeGraph [Dai+17], HitGraph [Zho+19], and ThunderGP [Che+21b] – representing the different approaches to graph processing on FPGAs. While AccuGraph and HitGraph are orthogonal approaches representing the currently most relevant paradigms, edge- and vertex-centric graph processing (both with horizontal partitioning), ForeGraph is one of the few systems with interval-shard partitioning and compressed edge list, and ThunderGP uses vertical partitioning with a sorted edge list.

With these systems, we show that GraphSim is able to reproduce results as summarized in Fig. 4.3, denoting the simulation percentage error $e = \frac{100 \times |s-t|}{t}$, with simulation performance s, on equal memory configurations and graph data sets, compared to the performance numbers t taken from the respective paper grouped by accelerators and graph problems. We get a reasonable mean error of 22.63% with two outliers in BFS on ForeGraph and single-source shortest-paths (SSSP) on HitGraph caused by insufficient specification of root vertices which we will further explore in Sect. 4.3.2.

In the following, we first introduce the simulation environment GraphSim including a detailed explanation of FPGA memory hierarchies, memory access abstractions used in GraphSim, and DRAM simulators (Sect. 4.1). Thereafter, we provide a classification of existing graph processing accelerators from Sect. 3.2.1 and extract GraphSim memory access pattern implementations for AccuGraph, ForeGraph, HitGraph, and ThunderGP (Sect. 4.2). With the GraphSim memory access pattern implementations, we conduct a reproducibility study (Sect. 4.3.2) and uncover deficiencies in performance measurement practices and a performance comparison and analysis (Sect. 4.3.4) by comprehensively exploring the relevant *performance dimensions*: (i) accelerator design decisions, (ii) graph problems, (iii) data set characteristics, (iv) memory technology, and (v) memory access optimizations. Additionally, we show the reduced effort of engineering new ideas with GraphSim by example of two novel optimizations to AccuGraph (Sect. 4.4). Finally, we discuss related work (Sect. 4.5) and finish with a discussion (Sect. 4.6).

Notably, among other insights, we discover a *trade-off* in the asynchronous graph processing scheme of AccuGraph and ForeGraph compared to the synchronous graph processing scheme of HitGraph and ThunderGP. Additionally, we confirm that modern memory like HBM does not necessarily lead to better performance (cf. [SFB16; Wan+20b]).

Parts of this chapter have previously been published in the proceedings of BTW 2021 [DRF21b] (GraphSim simulation environment and reproducibility study) and GRADES-NDA 2021 [DRF21a] (extensive analysis of impact of graph processing accelerator design decisions on performance).

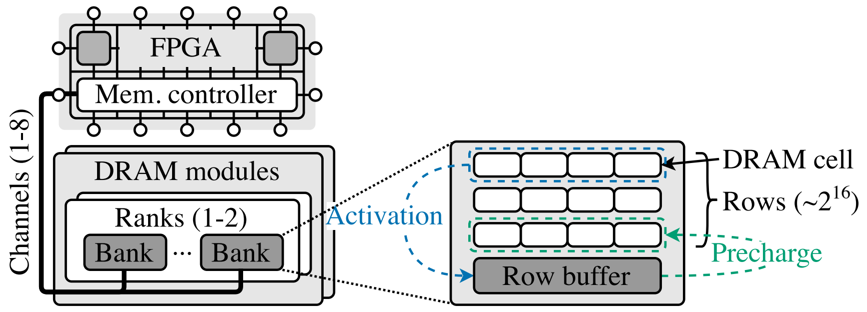


Fig. 4.4 FPGA and DRAM (adapted from [KYM16]).

4.1 Memory Access Simulation

We start this section by introducing the memory hierarchy of FPGAs and more specifically how DRAM works internally. Thereafter, we show the abstractions we developed to implement memory access patterns in GraphSim. Lastly, we motivate the selection of Ramulator [KYM16] as our DRAM simulator and briefly explain how Ramulator models memory and is configured for our purpose.

4.1.1 Memory Hierarchy of FPGAs

The memory hierarchy of FPGAs is split up into on-chip and off-chip memory. On-chip, FPGAs implement distributed memory that is made up of single registers and is mostly used as storage for working values and BRAM for fast storage of data structures. For storage of larger data, DRAM is attached as off-chip memory (e. g., DDR3[1], DDR4[2], or HBM[3]). Exemplary for DRAM organization, we show the internal organization of DDR3 memory in Fig. 4.4, which at the lowest level contains DRAM cells each representing one bit. The smallest number of DRAM cells (e. g., 16) that is addressable is called a column. Several thousand (e. g., 1,024) columns are grouped together into rows. Further, independently operating banks combine several thousand (e. g., 65,536) rows with a row buffer each. Requests to data in a bank are served by the row buffer based on three scenarios: *(1)* When the addressed row is already buffered, the request is served with low latency (e. g., t_{CL}: 11ns). *(2)* If the row buffer is empty, the addressed row is first activated (e. g., t_{RCD}: 11ns), which loads it into the row buffer, and

[1]JESD79-3 DDR3 SDRAM Standard
[2]JESD79-4 DDR4 SDRAM Standard
[3]JESD235D High-Bandwidth Memory (HBM) DRAM Standard

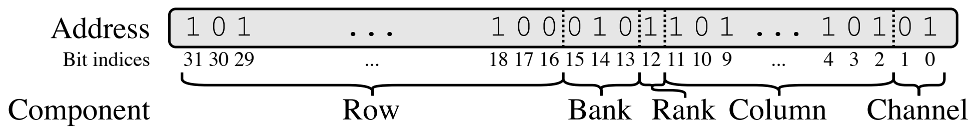


Fig. 4.5 DRAM addressing.

then the request is served. *(3)* However, if the row buffer currently contains a different row from a previous request, the current row has to be first pre-charged (e. g., t_{RP}: 11ns) and only then the addressed row can be activated and the request served. Additionally, there is a minimum latency between switching rows (e. g., t_{RAS}: 28ns). Thus, for high performance, row switching should be avoided as much as possible. As a reference point, a last-level cache miss on current Skylake Intel CPUs takes 17ns at 2.6GHz clock. Since one bank does not provide sufficient bandwidth, 8 parallel banks further form a rank. Multiple ranks operate in parallel but on the same I/O pins, thus increasing capacity of the memory, but not bandwidth. Finally, the ranks of the memory are grouped into channels. Each channel has its own I/O pins to the FPGA such that the bandwidth linearly increases with the number of channels. To achieve good performance, current DRAM additionally employ 8n prefetching (meaning 8 bursts). Thus, 64 bytes are returned for each request which we call a cache line in the following. DDR4 doubles total number of banks at the cost of added latency due to another hierarchy level called bank groups, which group two to four banks. HBM as a new stacked memory technology introduces very high bandwidth in a small package. The single channels have double as many banks (16) as DDR3 with half the prefetch (4n) which however transport double the data per cycle (128 bit). Additionally, HBM has smaller row buffers and can have many more channels in a confined space.

A common misconception might be that with FPGAs, off-chip memory can be accessed in smaller chunks than CPU cache lines. On FPGAs only the memory usage on-chip can be customized to fit the problem-to-solve. Access granularity cannot be reduced with FPGAs.

Data in DRAM is accessed by giving the memory a physical memory address that is split up into multiple parts internally representing addresses for each component in the DRAM hierarchy (cf. Fig. 4.5). Based on this, different addressing schemes are possible. An example addressing scheme that aids distribution of requests over channels might address the channels with the least

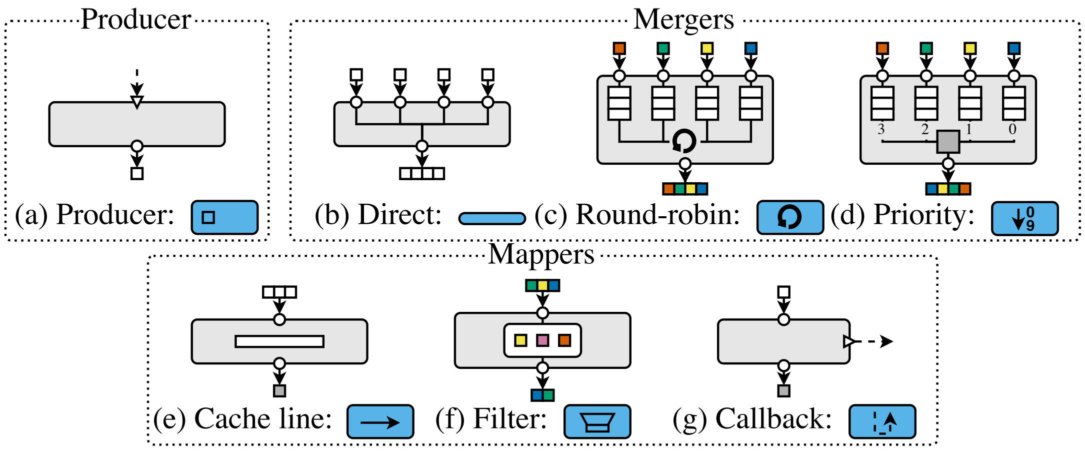


Fig. 4.6 Memory access abstractions.

significant bits of the address, meaning subsequent addresses go to different channels, then address columns, ranks, banks, and rows.

4.1.2 GraphSim Patterns

In GraphSim, since memory access is the dominating factor in graph processing, the necessity of cycle-accurate simulation of on-chip data flow is relaxed and only the off-chip memory access pattern is modeled (cf. Fig. 4.2). Thus, we reduce development time and complexity massively within reasonable error when compared to performance measurements on real hardware. Modelling the off-chip memory access pattern means modelling request types, request addressing, request amount, and request ordering. Request type modelling is trivial since it is clear when requests read and write data. For request addressing, we assume that the different data structures (e. g., edge list and vertex values) are stored adjacently in memory as plain arrays. We generate memory addresses according to this memory layout and the width of the array types in bytes. Request volume modelling is mostly based on the size n of the vertex set, the size m of the edge set, average degree deg, and partition number k of a given graph. Lastly, we only simulate request ordering through mandatory control flow caused by data dependencies of requests. We assume that computations and on-chip memory accesses are instantaneous by default. In the following we introduce memory abstractions we developed for modelling request and control flow.

Figure 4.6 shows an overview of the memory access abstractions and their icons grouped by their memory access role as *producer*, *merger*, and *mapper*.

Producer At the start of each request stream, a *producer* shown in Fig. 4.6(a) is used to turn control flow triggers (dashed arrow) into a request stream (solid arrow). The producer might be rate limited to avoid starving other producers, but if only a single producer is working at a time or requests are load balanced down-stream, the requests are created in bulk.

Mergers Multiple request streams might then be merged with *mergers*, since Ramulator only has one endpoint. We have deduced abstractions to merge requests in a *direct*- (Fig. 4.6(b)), *round-robin*- (Fig. 4.6(c)), and *priority*-based (Fig. 4.6(d)) fashion. If there are multiple request streams that do not operate in parallel, direct merging is applied. If request streams should be equally load-balanced, round-robin merging is applied. If request streams should take precedence over each other, priority merging is applied. For this, a priority is assigned to each request stream and requests are merged based on that.

Mappers Additionally to request creation with producers and ordering with mergers, we also found abstractions for request *mappers*. Thus, we introduce *cache line* buffers shown in Fig. 4.6(e) for sequential or semi-sequential accesses that merge subsequent requests to the same cache line into one request. We do buffering such that multiple concurrent streams of requests benefit from it independently by placing it as far from the memory as necessary to merge the most requests. For data structures that are placed partially in on-chip memory (e. g., prefetch buffers and caches), and thus partially not require off-chip memory requests, we introduce request *filters* shown in Fig. 4.6(f) that discard unnecessary requests. For control flow, we use a *callback* (Fig. 4.6(g)) abstraction. We disregard any delays in control flow propagation and just directly let the memory call back into the simulation. If requests are served from a cache line or filter abstraction, the callback is executed instantly.

Simulation environment In GraphSim, we instantiate a graph processing simulation and a Ramulator instance, and tick them according to their respective clock frequency. For the graph processing simulation we focus on configurability of all aspects of the simulation such that we can quickly run differently parameterized performance measurements. Our simulation works on multiple request streams that are merged into one and fed into Ramulator that calls back into the simulation whenever a request is served. This leads us to a immensely reduced implementation time and complexity, gives us more insight into the memory, and provides portability of ideas developed in the simulation environment.

4.1.3 DRAM Simulator – Ramulator

To speed up the engineering of graph processing on FPGA accelerators, a DRAM Simulator is an integral part of our simulation environment (cf. Fig. 4.2). For our purposes, we need a DRAM simulator that supports at least DDR3 and DDR4 memory. We chose Ramulator [KYM16] for this work over other alternatives like DRAMSim2 [RCJ11] and USIMM [Cha+12b] because – to the best of our knowledge – it is the only DRAM simulator which supports (among many others like LPDDR3/4 and HBM) both of those DRAM standards (DDR3 and DDR4).

Ramulator models DRAM as a tree of state machines (e. g., channel, rank, and bank in DDR3) where transitions are triggered by user or internal commands. However, Ramulator does not make any assumptions about data in memory. Purely the request and response flow is modelled with requests flowing into Ramulator and responses being called back. The Ramulator configuration parameters that are relevant to our work are: (i) DRAM standard, (ii) channel count, (iii) rank count, (iv) DRAM speed specification, (v) DRAM organization.

4.2 Selected Graph Processing Accelerators

We selected four graph processing accelerators from Sect. 3.2.1 to be simulated in GraphSim. The selection is based on if (i) they support at least BFS, PR, and WCC, (ii) they run on commodity FPGAs, (iii) the vertex set is not required to fit into on-chip memory, and (iv) the respective paper provides enough detail to model the memory access pattern. The accelerators fitting all but one of these criteria can be found in Tab. 4.1. The ones we chose to include in this chapter are highlighted in bold. We excluded the accelerators based on the Convey HC-2 and HMC systems because they cannot be implemented with commercially available FPGAs anymore. Additionally, support for BFS only restricts their usefulness. Furthermore, we excluded the systems by Ayupov et al. [Ayu+18] and Yang et al. [Yan+20a] because they did not provide sufficient detail to reproduce the results with the simulation environment. The set of accelerators we choose for this

Table 4.1 FPGA-accelerated graph processing systems in the context of the most relevant design decisions.

Identifier	Graph problems	System	Iter.	Flow	Partitioning	Binary representation	Prop.
ForeGraph [Dai+17]	BFS, PR, WCC	Simulation	Edge	n/a	Interval-shard	Compressed edge list	Async.
HitGraph [Zho+19]	BFS, PR, WCC, SSSP, SpMV	FPGA	Edge	n/a	Horizontal	Sorted edge list	Sync.
ThunderGP [Che+21b]	BFS, PR, WCC, SSSP, SpMV, ...	FPGA	Edge	n/a	Vertical	Sorted edge list	Sync.
AccuGraph [Yao+18]	BFS, PR, WCC	FPGA	Vertex	Pull	Horizontal	Inverse CSR	Async.
Betkaoui et al. [Bet+12a]	BFS	Convey HC-2	Vertex	Push	Horizontal	CSR	Sync.
CyGraph [Att+14]	BFS	Convey HC-2	Vertex	Push	None	CSR (in-place)	Sync.
Ayupov et al. [Ayu+18]	BFS, PR, WCC	Simulation	Vertex	Pull	None	Inverse CSR	Async.
TorusBFS [LRG15]	BFS	Convey HC-2	Vertex	Push	None	CSR	Sync.
Yang et al. [Yan+20a]	BFS, PR, WCC	FPGA	Hybrid	n/a	Vertical	Edge blocks & CSR	Sync.
Zhang et al. [ZL18]	BFS	HMC FPGA	Vertex	Hybrid	None	CSR	Sync.
ScalaBFS [Liu+21]	BFS	HBM FPGA	Vertex	Hybrid	None	CSR	Sync.

Iter.: Iteration scheme; Prop.: Update propagation; n/a: Not applicable; Async.: Asynchronous; Sync.: Synchronous

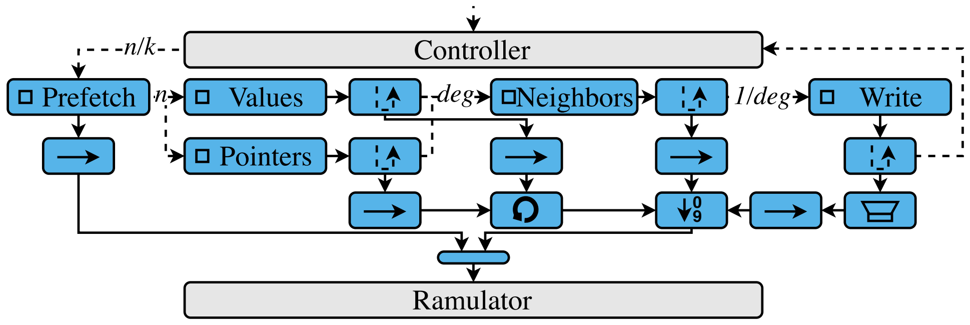


Fig. 4.7 AccuGraph request and control flow.

chapter represents the currently highest performing graph processing accelerators and all currently applied partitioning and iteration schemes. In the following, we introduce how they can be implemented in GraphSim in alphabetical order. Note that developing and verifying a complicated FPGA design usually takes weeks, while the implementation of a memory access simulation for a new graph processing accelerator in GraphSim takes only days or hours.

4.2.1 AccuGraph

AccuGraph [Yao+18] proposes a flexible accumulator based on a modified prefix-adder able to produce and merge updates to multiple vertices per cycle. Figure 4.7 shows the request and control flow modelling of AccuGraph. The controller is triggered and iterates over the graph until there are no more changes in the previous iteration. Each iteration triggers processing of all partitions. Processing of each partition starts with triggering the prefetch request *producer* which prefetches the partitions $\frac{n}{k}$ vertex values (with $n = |V|$ and the partition count k) sequentially which is passed through a *cache line* memory access abstraction merging adjacent requests to the same cache line into one. Thereafter, values and pointers of all destination vertices are fetched sequentially. Both of those request streams are annotated with *callbacks* which return control flow for each served request. Those two request streams are merged *round-robin*, because a value is only useful with the associated pointers. In parallel, neighbors are read from memory sequentially, annotated with their value from the prefetched vertex values, and one edge is materialized for each neighbor with its corresponding source vertex. The aforementioned accumulator thereafter produces updates for each edge. If the source vertex value changes, the result is written back

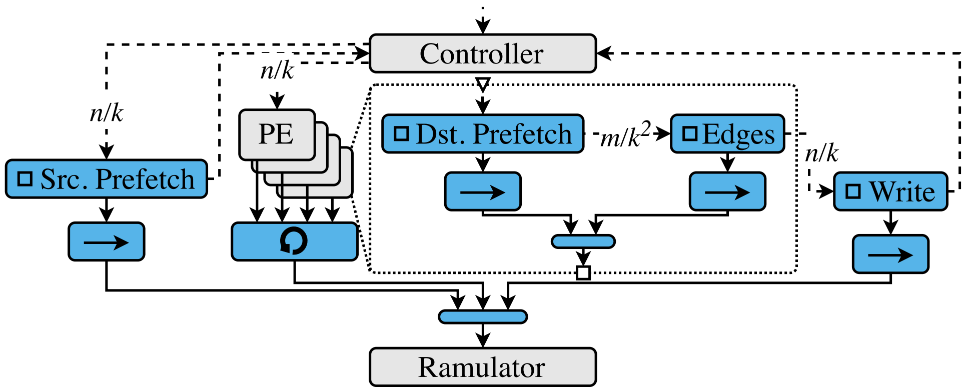


Fig. 4.8 ForeGraph request and control flow.

to off-chip memory (unchanged values are filtered by the *filter* memory access abstraction). All of these request streams are merged by *priority* with write request taking the highest priority and neighbors the second highest.

AccuGraph is parameterized by the number of vertex and edge pipelines (8 and 16 in the original paper) and the partition size. The original paper also describes an FPGA-internal data flow optimization which allows to approximate pipeline stalls, improving simulation accuracy significantly. The value cache used for the prefetched values is partitioned into 16 memory banks on the FPGA which can each serve one vertex value request per clock cycle. Since there are 16 edge pipelines in a standard deployment of AccuGraph, performance deteriorates quickly, when there are stalls. Thus, for this particular accelerator, we model stalls of the vertex value cache in the control flow between the neighbors and write producers. A neighbors request callback is delayed until the memory bank can serve the vertex value request.

4.2.2 ForeGraph

ForeGraph [Dai+17] stores the edges of the shards as compressed 32 bit edges with two 16 bit vertex identifiers each. This is possible due to the interval size being limited to 65,536 vertices. In each iteration, ForeGraph prefetches the source intervals one after another and for each source interval processes its corresponding shards by additionally prefetching the destination interval and sequentially reading and processing the edges (Fig. 4.8). This results in purely sequential off-chip requests with all random vertex value requests during edge processing being served by caches on-chip. After a shard has been processed, the

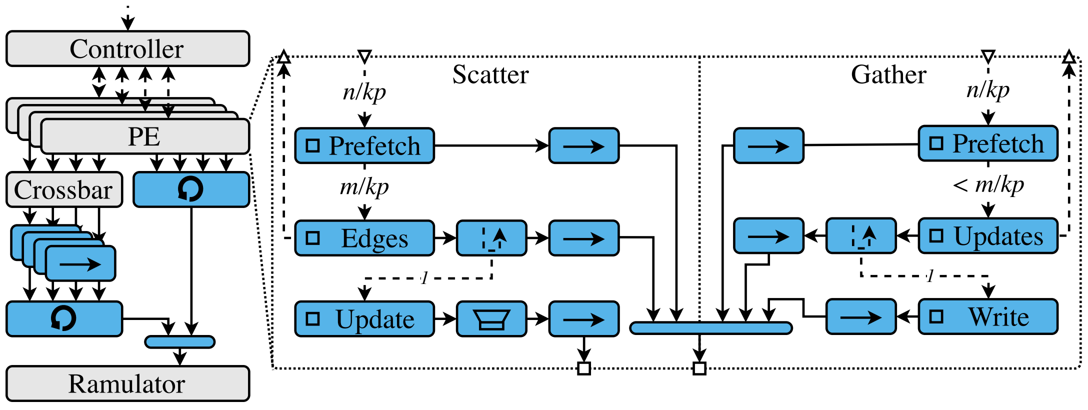


Fig. 4.9 HitGraph request and control flow.

destination interval is sequentially written back to off-chip memory. To achieve competitive performance, ForeGraph instantiates p processing elements (PE). These each work on their own set of source intervals and share the memory access in round-robin fashion. ForeGraph is parameterized with the number of PEs p and the partition size k.

4.2.3 HitGraph

HitGraph [Zho+19] execution starts with triggering a controller that itself triggers iterations of edge-centric processing until there were no value changes in the previous iteration (Fig. 4.9). In each iteration, the controller first schedules all k partitions for the scatter phase (producing updates), before scheduling all partitions to the gather phase (applying updates). Partitions are assigned beforehand to channels of the memory (four channels in [Zho+19]) and there are p processing elements (PE), one for each channel.

The scatter phase starts by prefetching the $\frac{n}{kp}$ values of the current partition. After all requests are produced, the prefetch step triggers the edge reading step that reads all $\sim\frac{m}{kp}$ edges of the partition. For each edge request, we attach a callback that triggers producing an update request. The target address depends on its destination vertex that can be part of any of the k partitions. Thus, there is a crossbar (unique to this accelerator) that routes each update request to a cache line access abstraction for each partition to sequentially write into a partition-specific update queue.

Similar to scatter, the gather phase starts with prefetching the $\frac{n}{kp}$ vertex values of the current partition sequentially. After value requests have been

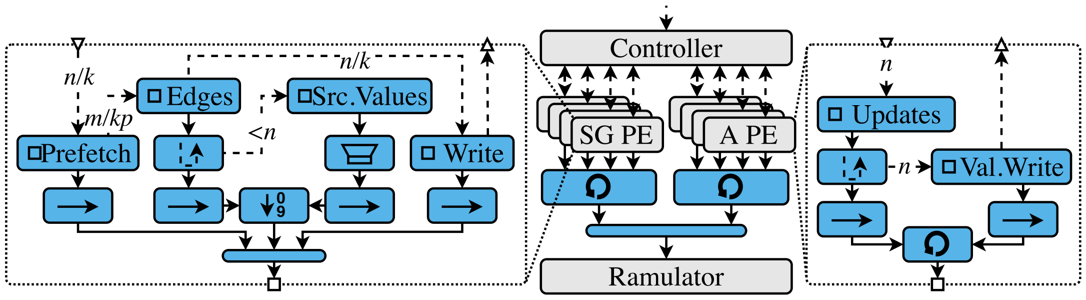


Fig. 4.10 ThunderGP request and control flow.

produced, the prefetch producer triggers the update producer, which sequentially reads the update queue written by the scatter phase before. For each update we register a callback that triggers the value write. The value writes are not necessarily sequential, but especially for iterations where a lot of values are written, there is a lot of locality.

All request streams in each PE are merged directly into one stream without any specific merging logic, since only one producer is producing requests at a time. Since all PEs are working on independent channels and Ramulator only offers one endpoint for all channels combined, we employ a round-robin merge of the PE requests in order not to starve any channel.

HitGraph is parameterized with the number of PEs p, pipelines and the partition size q. The number of PEs is fixed to the number of memory channels because each PE works on exactly one memory channel. The number of pipelines is limited by the bandwidth available per channel. Lastly, the partition size is chosen such that $p \times q$ vertices fit into BRAM.

4.2.4 ThunderGP

Like HitGraph, ThunderGP [Che+19] is based on a asynchronous update propagation scheme based on edge-centric iteration over the input graph. The graph is vertically partitioned into k partitions and each partition is split up into p chunks where p is equal to the number of memory channels. Each memory channel contains the whole vertex value set of the graph, its corresponding chunk of each partition and an update set. For each iteration, a scatter-gather phase (SG PE) is applied for each partition before an apply phase (A PE) is executed for each partition (Fig. 4.10). The scatter-gather phase starts by prefetching the partitions destination vertex value set into BRAM sequentially. Upon finalization, this triggers sequential edge reading. Each edge callback triggers loading its source vertex value. Since edge lists are sorted by source vertex, this is semi-sequential and a vertex value buffer filters duplicate source vertex value requests. When edge reading is finished, the updated values are written back to memory. The apply stage reads all updates sequentially and combines the updates produced by all channels in the previous phase into one value per vertex which is written back to all channels. This step may cause many duplicate value reads and writes.

ThunderGP is parametrized with the number of scatter-gather groups and apply PEs which matches the number of memory channels. Additionally, both of those PEs have a parameter for edges processed in parallel. Finally, ThunderGP has a parameter for partition size.

4.3 Evaluation

In this section, we introduce the data sets and GraphSim parameters used for the evaluation. We validate our simulation approach by reproducing the results reported for AccuGraph, ForeGraph, HitGraph, and ThunderGP, and explain the simulation error observed when comparing our measurements to those on real FPGA hardware. Thereafter, we, for the first time, comprehensively compare these graph processing approaches producing a number of insights.

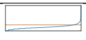
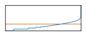
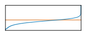
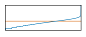
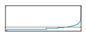
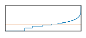
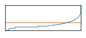
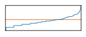
4.3.1 Setup: Data Sets & Simulation Environment

We take the graph data sets that are often used to benchmark the systems in Tab. 4.1 for our own evaluation (cf. Tab. 4.2). Two important aspects when working with these graphs are their directedness and the root vertices[4] used (e. g., for BFS or SSSP) because they can have a significant impact on performance.

Regarding data types, we use 32 bit data types for all vertex identifiers, CSR pointers, and values (integer or float). The only exception is ForeGraph which can use 16 bit vertex identifiers due to its interval-shard partitioning. An unweighted edge is always two vertex identifiers wide and a weighted edge is an additional 32 bits wider due to the attached edge weight. This is sensible for

[4]Root vertices: tw - 2748769; lj - 772860; or - 1386825; wt - 17540; pk - 315318; yt - 140289; db - 9799; sd - 30279; rd - 1166467; bk - 546279; r24 - 535262; r21 - 74764

Table 4.2 Graphs used often by systems in Tab. 4.1 (real-world graphs from SNAP [LK14]; Graph500 generator for R-MAT).

Name	$\|V\|$	$\|E\|$	Dir.	Degs.	D_{avg}	ø	SCC
twitter (tw)	41.7M	1,468.4M	👍		35.25	75	0.80
live-journal (lj)	4.8M	69.0M	👍		14.23	20	0.79
orkut (or)	3.1M	117.2M	👎		76.28	9	1.00
wiki-talk (wt)	2.4M	5.0M	👍		2.10	11	0.05
pokec (pk)	1.6M	30.6M	👎		37.51	14	1.00
youtube (yt)	1.2M	3.0M	👎		5.16	20	0.98
dblp (db)	426.0K	1.0M	👎		4.93	21	0.74
slashdot (sd)	82.2K	948.4K	👍		11.54	13	0.87
roadnet-ca (rd)	2.0M	2.8M	👎		2.81	849	0.99
berk-stan (bk)	685.2K	7.6M	👍		11.09	714	0.49
rmat-24-16 (r24)	16.8M	268.4M	👍		16.00	19	0.02
rmat-21-86 (r21)	2.1M	180.4M	👍		86.00	14	0.10

Dir.: Directed; Degs.: Degree distribution on log. scale; SCC: Ratio of vertices in the largest strongly-connected component to n; 👍: yes, 👎: no

Table 4.3 DRAM configurations.

Identifier	Type	#Chan.	Ranks	Data rate	BW	Size	RBS
AccuGraph	DDR4	1	1	2400 MT/s	19.2 GB/s	2 Gb	8 KB
ForeGraph	DDR4	1	1	2400 MT/s	19.2 GB/s	4 Gb	8 KB
HitGraph	DDR3	4	2	1600 MT/s	12.8 GB/s	8 Gb	8 KB
ThunderGP	DDR4	4	1	2400 MT/s	19.2 GB/s	16 Gb	8 KB
Default	DDR4	1 – 4	1	2400 MT/s	19.2 GB/s	16 Gb	8 KB
DDR3	DDR3	1 – 4	1	2133 MT/s	17.1 GB/s	8 Gb	8 KB
HBM	HBM	1 – 8	n/a	1000 MT/s	16.0 GB/s	4 Gb	2 KB

#Chan.: Number of channels; BW: Bandwidth per channel; RBS: Row buffer size; n/a: not applicable

all accelerators we encountered, since there are no excessively large benchmark graphs or requirements on more precision that do not fit into 32 bits.

Table 4.3 shows the DRAM configurations used in the respective paper of the selected accelerators as well as the DRAM configurations we use in this work. The default is the DDR4 configuration, since this is the most common in the systems the selected accelerators run on. The DDR3 and HBM configurations are used to compare performance on different memory technologies in Sect. 4.3.4.3. By default, we use one channel, but also do a scale test for HitGraph and ThunderGP due to their support for multiple channels.

As mentioned in Sect. 2.1.1.1, we consider the five graph problems BFS, PR, WCC, SpMV, and SSSP. However, SpMV and SSSP require edge weights which only HitGraph and ThunderGP support.

We use Ramulator[5] commit `dd326` and add a function to flush the statistics for multiple consecutive runs. To compile Ramulator and GraphSim we use `clang++` 5.0.1. For Ramulator with C++11 and for GraphSim with C++17 and `-D_FILE_OFFSET_BITS=64` to be able to read files larger than 2GB.

In subsequent performance measurements, we use the Graph500 benchmark's MTEPS definition, which specifies MTEPS as $|E|/t_{exec}$, where t_{exec} denotes the execution time. Notably, this is the definition of MTEPS that the Graph500 benchmark uses but is different to the measure that most graph processing accelerator papers report (i. e., total number of edges read during execution divided by execution time), which we call MREPS. MREPS do not normalize the runtime to graph size but rather denote raw edge processing performance. For both MTEPS and MREPS higher is better.

4.3.2 Reproducibility Study

We measure the quality of the simulation as the percentage error $e = \frac{100\times|s-t|}{t}$, with simulation performance s, on equal memory configurations and graph data sets, compared to the performance numbers t taken from the respective paper grouped by accelerators and graph problems. The AccuGraph numbers are taken from a chart and the ForeGraph, HitGraph, and ThunderGP numbers are extracted from table in the respective papers. To reproduce the experiments as closely as possible, we parameterized GraphSim according to configurations from the original papers (cf. Tab. 4.3).

Figure 4.11 shows performance measurements for AccuGraph for BFS, PR, and WCC as billions of read edges per second (GREPS). AccuGraph uses unweighted graphs and an optimized 8 bit unsigned integer for BFS (problematic for constant-degree graphs). For the reproducibility study, AccuGraph is assumed to fit all vertices in BRAM for BFS and only for PR and WCC measurements on live-journal and orkut, the partition size is set to $1,700,000$ vertices. The average error is very similar for all problems and fits the relative performance of the graph data sets well.

[5]Ramulator, visited 10/23: https://github.com/CMU-SAFARI/ramulator

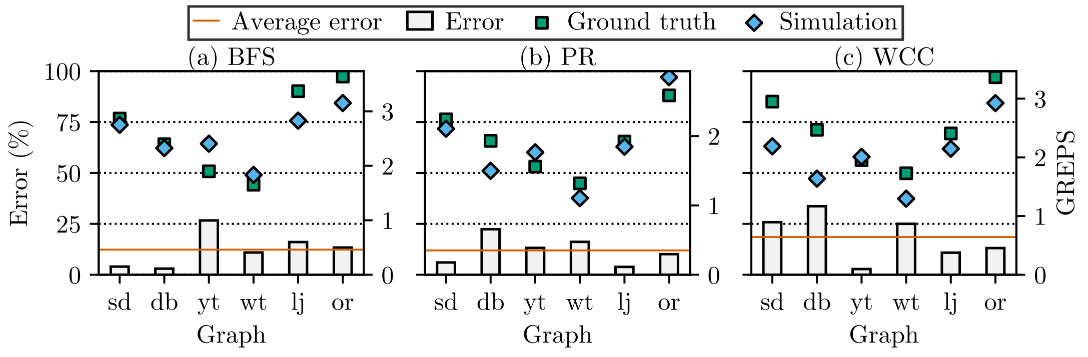


Fig. 4.11 AccuGraph reproducibility measurements.

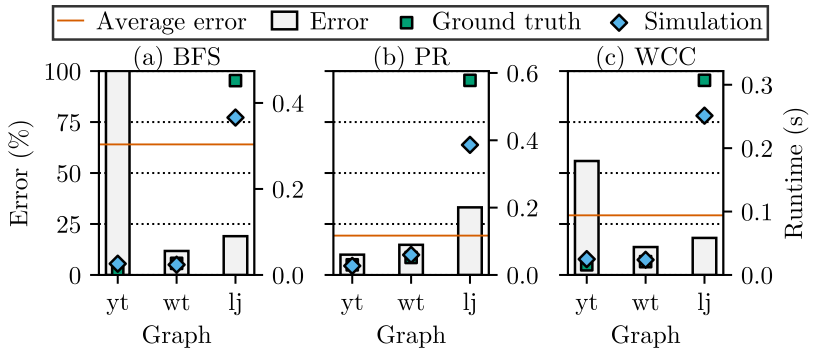


Fig. 4.12 ForeGraph reproducibility measurements.

The only consistent outlier is the youtube graph which relatively performs better in all simulation measurements than is suggested by the ground truth measurements (error source ❶). The original paper notes that the performance of AccuGraph logarithmically depends on the average degree of vertices. Thus, youtube should perform the way our measurements suggest, because it has a slightly higher average degree than the dblp graph. This may be an anomaly in the measurements performed by the AccuGraph authors. WCC is slightly slower in our simulations than they are on the accelerator and PR is slightly faster. There may be a fixed overhead that we are measuring in our experiments and is not measured in theirs. The better performance of PR, however, is expected, since we do not take the longer latencies and incurred pipeline stalls of floating point arithmetics into account (error source ❷).

Figure 4.12 shows performance measurements for ForeGraph for BFS, PR, and WCC as runtime in seconds. Overall, we are able to reproduce the ground truth well with our GraphSim measurements. However, BFs shows by far the worst error, with a big difference for the youtube graph. This can be explained by the problem's dependence on the input root vertex (error source ❸). The

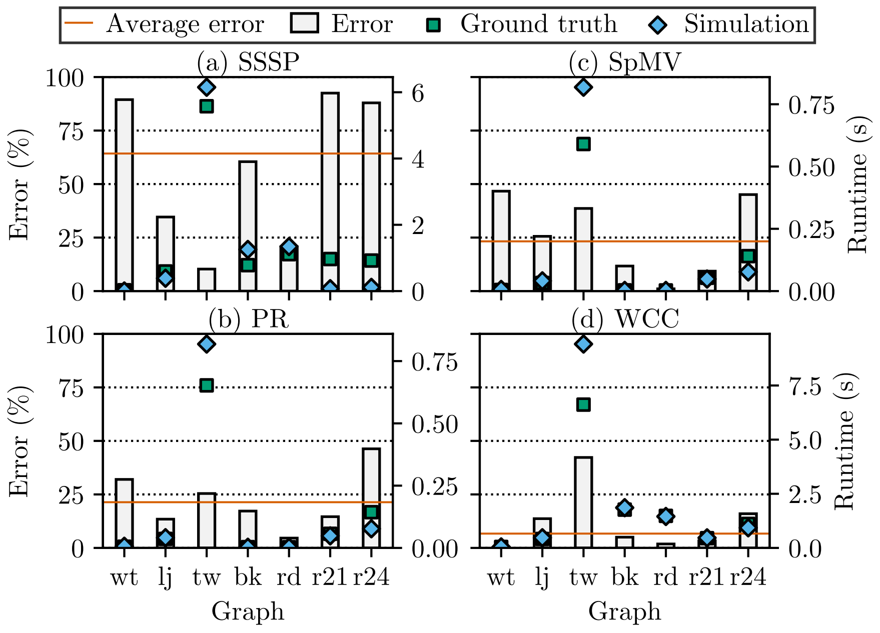


Fig. 4.13 HitGraph reproducibility measurements.

ForeGraph authors randomly choose only one root vertices which they do not report in the paper. However, BFS runtime especially for asynchronous graph processing based accelerators depends heavily on the source vertex because it influences the number of iterations over the graph until convergence on the result.

Figure 4.13 shows the HitGraph performance measurements for SpMV, PR, SSSP, and WCC as runtime in seconds. HitGraph uses weighted graphs and uniformly wide value types for all problems. Overall, we observe a consistent outlier in the twitter graph. However, we notice that the HitGraph paper reports the diameter of the twitter graph as being 15, while we report it as being 75 (cf. Tab. 4.2). Thus, we assume that our version of the graph is different and exclude it from all error averages in this paper while still showing it in the plots for completeness (error source ❹). SpMV and PR result in the same simulation performance, but since ground truth values are slightly different, we get a different error. In the original paper, the authors measure only a single iteration of SpMV and PR. However, we found that for very short runtimes of single iteration executions differences of a few cycles can already cause large deviations leading to the errors we observe for SpMV and PR (error source ❺). We advise using multiple iterations of such algorithms in benchmarks in the future.

SSSP shows by far the worst error, with some executions running much shorter in simulation than in the ground truth measurements. This can again be

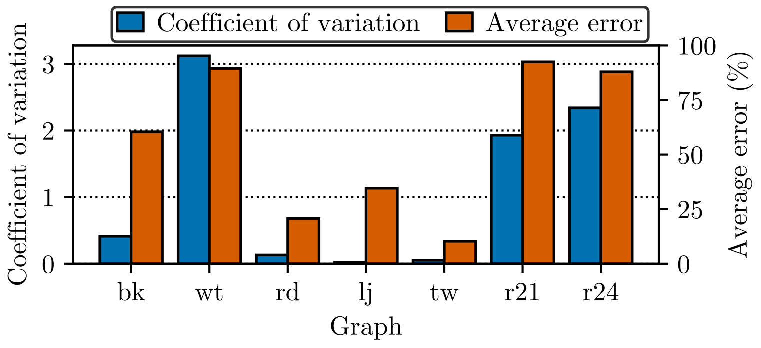


Fig. 4.14 HitGraph SSSP runtime variation study.

explained by the problem's dependence on the input root vertex (error source ❸). The HitGraph authors randomly choose 20 root vertices and report the average runtime. However, wiki-talk and the rmat graphs have many strongly-connected components (SSCs) with just one or a few vertices (cf. Tab. 4.2). This causes algorithms like SSSP and BFS to immediately terminate after one iteration over the graph with very little runtime which results in large variation in performance measurements for root vertices from many small and few big SSCs shown in Fig. 4.14. The error is strongly correlated to the coefficient of variation in the runtimes (given by $\frac{\sigma}{\mu}$ with the standard deviation σ and the mean μ). This leads us to the conclusion that 20 random root vertices are not enough to stabilize the runtime measurements for graphs with such structure. We advocate for sharing how roots are picked in the future.[6] Moreover, the HitGraph paper does not specify how edge weights are set in the graph, which can also influence runtimes of SSSP (error source ❻). We initialized all weights to 1. We regard WCC as the most reliable indicator for simulation quality because it does not depend on input variables and runs long enough so fixed overheads are irrelevant. We observe a low simulation error for WCC, which reassures us that the off-chip memory access modelling works well for HitGraph. Besides the twitter graph (which we explicitly excluded), the simulation almost perfectly matches the ground truth.

Figure 4.15 shows the simulation reproducibility measurements for ThunderGP for BFS, SpMV, SSSP, and WCC as MREPS. We excluded PR for brevity in this particular case because the results are very similar to WCC. For BFS and SSSP, the root vertices that ThunderGP used for the ground truth are always

[6]We generated the 20 random root vertices with the `mt19937` generator in C++ with seed 3483584297.

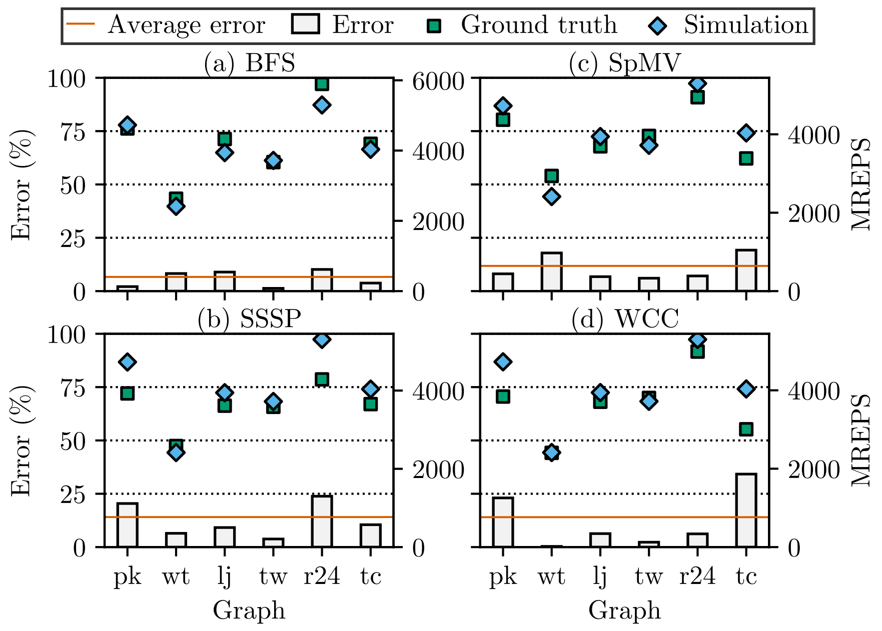


Fig. 4.15 ThunderGP reproducibility measurements.

zero. Thus, overall, we are able to reproduce the performance measurements of ThunderGP with great accuracy using GraphSim. There are no patterns of errors that are particularly pronounced.

4.3.3 Error Analysis

We saw that GraphSim is able to reproduce the ground truth performance measurements of the original papers with reasonable error (Sect. 4.3.2). This is possible for bandwidth-bound accelerators despite the radical approach of disregarding FPGA internals. Especially if relative performance behaviour of approaches is so significantly different (cf. Sect. 4.3.4), an average error of e. g., 8.997% for WCC is reasonable to make sound relative comparisons. However, we also identified six sources of errors which we discuss in the following. For measurements with insufficiently specified input parameters like start vertices (error source ❸) and edge weights (error source ❻) we see large errors for some graphs. Additionally, we attribute at least some of the error to noise in the measurements. For example, very low runtime measurements like individual iterations of SpMV and PR (error source ❺) can lead to significant noise. We see underestimation of runtime due to missing modelling of pipeline bubbles that slow down request generation or missing modelling of e. g., floating point units that perform complicated calculations (error source ❷). Lastly, there remain two graphs in twitter and youtube for which we cannot explain performance differences based on our simulation but rather attribute these differences to different data sets or different usage of them (error sources ❹ and ❶).

One not easily quantifiable, possible error source (error source ❼) we want to add here is interpretation based on understanding of the original paper's description of their approach. This was e. g., especially necessary for data structures with missing data type specifications. To aid researchers trying to understand the approaches we specified all data types and advise to completely specify such parameters in the future to aid reproduction of results.

We get a reasonable mean error of 22.63% with only two major outliers in BFS on ForeGraph and single-source shortest-paths (SSSP) on HitGraph caused by insufficient specification of root vertices We advise that the simulation should be used in use cases where relative performance behaviour is compared rather than where absolute performance should be estimated. Additionally, if the relative performance behaviour is close for the compared approaches our simulation approach might lead to inaccurate conclusions.

4.3.4 Accelerator Comparison

So far, graph accelerators were compared by looking at their absolute reported performance numbers, and thus making comparisons based on different FPGAs, memory architectures, number of memory channels, and consequently off-chip memory bandwidth available (e. g., [Zho+19; Yao+18]). In this section, we compare AccuGraph, ForeGraph, HitGraph, and ThunderGP along performance dimensions relevant for graph processing: (i) accelerator design decisions and (ii) graph problems, (iii) data set characteristics, (iv) memory technology, and (v) memory access optimizations.

4.3.4.1 Design Decisions & Graph Problems

Figure 4.16 shows a comparison of the four graph processing accelerators on all graphs from Tab. 4.2 for BFS, PR (one iteration), and WCC. While we start with a general performance analysis with respect to accelerator design decisions and graph problems (performance dimensions (i) and (ii)) in this subsection, we refer back to Fig. 4.16 throughout this section for more particular performance effects. We also introduce four critical performance metrics in Fig. 4.17 to explain

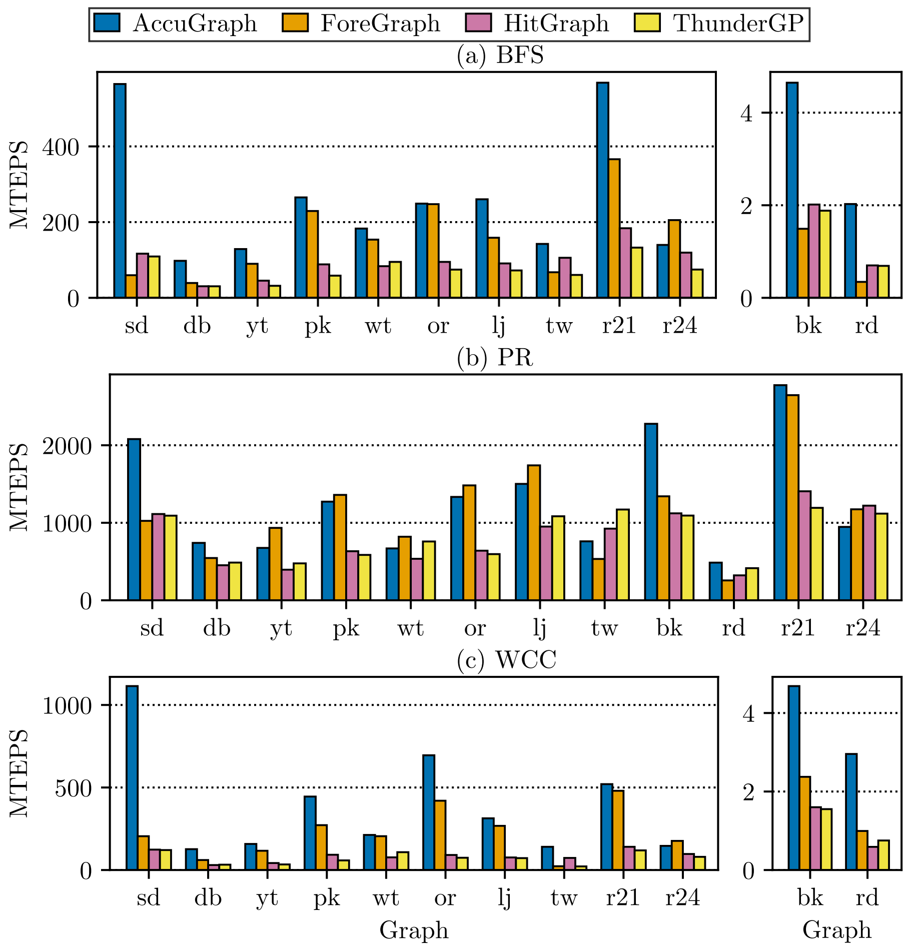


Fig. 4.16 Comparison of graph processing accelerators grouped by graphs for BFS, PR, and WCC (DDR4, single-channel).

our observations throughout this section. *Number of iterations* has the biggest impact on performance since each iteration entails prefetching, edge, and value reading and writing (Fig. 4.17(a)). Edge reading is the dominating factor of each iteration. Thus, *bytes per edge* (Fig. 4.17(b)) and *edges read per iteration* (Fig. 4.17(d)) are the two second most important performance metrics. Finally, *values read per iteration* (Fig. 4.17(c)) can especially play a role for sparse and large graphs. We only show BFS plots in Fig. 4.17 for brevity because the numbers are very similar for PR and WCC.

Overall, performance of PR is the highest because only one iteration is performed (cf. Fig. 4.16). BFS and WCC performance is overall similar. For BFS and WCC, performance on the berk-stan and roadnet-ca graph – which we break out in separate plots with a difference y-axis scale – is significantly lower. This, however, is expected as berk-stan and roadnet-ca are graphs with a

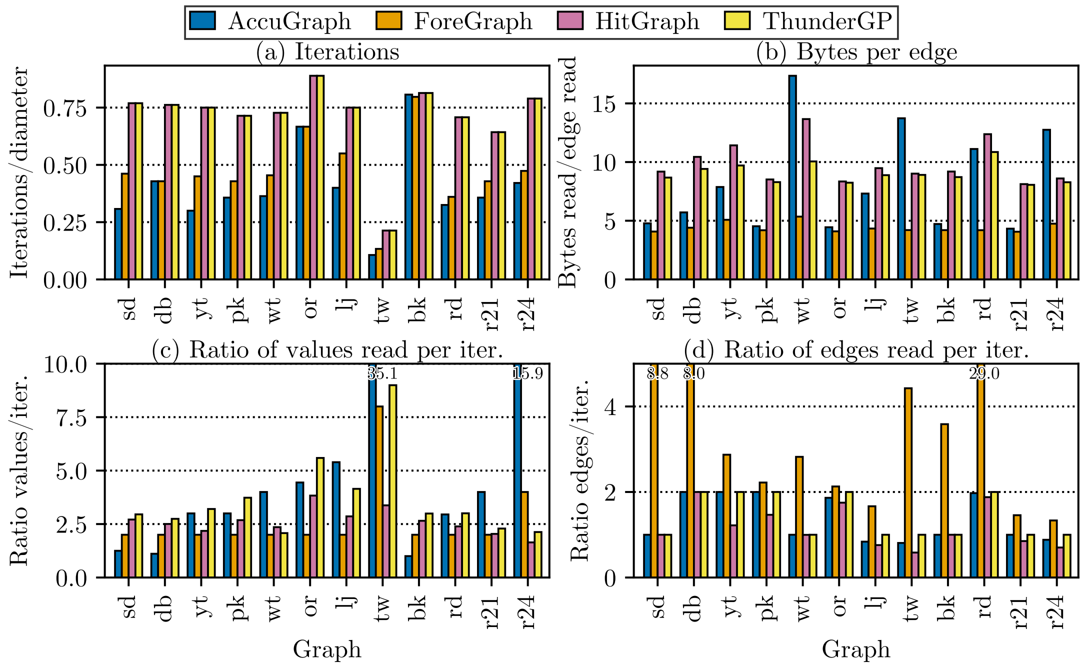


Fig. 4.17 Critical performance metrics (exemplary for BFS).

large diameter and thus require many more iterations to complete. We notice that AccuGraph and ForeGraph on average perform better than HitGraph and ThunderGP. To explain this, we additionally notice that AccuGraph and ForeGraph finish in significantly less iterations over the graph for BFS and WCC (not shown) than HitGraph and ThunderGP relative to the graph's diameter (cf. Fig. 4.17(a)). This is possible due to the asynchronous graph processing scheme of AccuGraph and ForeGraph leading to convergence to the result in less iterations (*insight 1*). The iteration reduction of asynchronous graph processing is even more pronounced for WCC leading to a more pronounced performance advantage of AccuGraph and ForeGraph over HitGraph and ThunderGP for WCC when compared with BFS (cf. Fig. 4.16). Additionally, AccuGraph and ForeGraph read significantly less bytes per edge on average (cf. Fig. 4.17(b)). This is due to the CSR data structure of AccuGraph and the compressed edges for ForeGraph (*insight 2*). For AccuGraph the exact number of bytes depends on the density of the graph which we discuss in Sect. 4.3.4.2. ForeGraph always needs 4 bytes per edge, 2 bytes per vertex identifier, and some additional bytes to prefetch the value intervals. HitGraph and ThunderGP need 8 bytes for each edge plus prefetching values and reading updates.

We also measured performance on weighted graphs for HitGraph and ThunderGP on SSSP and SpMV. However, there were no significant differences in performance besides overall longer runtimes due to bigger edge size (because of edge weights) compared to BFS and PR respectively. Thus, we do exclude SSSP and SpMV plots for brevity.

4.3.4.2 Data Set Characteristics: Graph Properties

In this section, we discuss performance effects observable in Fig. 4.16 due to the graph properties (performance dimension (iii)) size ($|E|$), density (D_{avg}), and skewness of the degree distribution (as Pearson's moment coefficient of skewness $\mathbb{E}[(\frac{D-\mu}{\sigma})^3]$ with D the degrees of the graph). The first trend we notice is that AccuGraph and ForeGraph performance decreases relative to HitGraph and ThunderGP for large graphs like rmat-24-16 and twitter. For the asynchronous graph processing scheme, destination vertex values need to be present when processing an edge which leads to loading these values many times instead of just once for update application (cf. Fig. 4.17(c)). Thus, asynchronous graph processing leads to disproportionately more value reads for large graphs (*insight 3*). Particular to AccuGraph, we see that performance is especially good for small graphs with only one partition ($|V| < 1{,}024{,}000$ for our configuration) such as slashdot, dblp, and berk-stan. AccuGraph saves vertex value reads for these graphs with skipping the prefetch step because the values are already in on-chip memory (cf. Fig. 4.17(c)). However, for large graphs, AccuGraph still needs $n+1$ CSR pointers for each partition leading to less savings in bytes per edge with horizontal partitioning (*insight 4*). HitGraph and ThunderGP performance is very similar in general. We only see a significant difference in performance for the twitter graph due to ThunderGP reading many more values because of vertical partitioning scheme. With an optimization described in Sect. 4.3.4.4, HitGraph counteracts excessive value reads.

All accelerator approaches benefit from dense graphs, with the effect being more pronounced for AccuGraph and ForeGraph (only working a full potential when $D_{avg} > 16$) due to a significant amount of pipeline stalls for sparse graphs (cf. Fig. 4.18). For asynchronous graph processing accelerators and sparse graphs like dblp, youtube, and roadnet-ca, vertex value reads make up significantly more of the runtime (addition to *insight 3*). Additionally, AccuGraph performance suffers for sparse graphs because the ratio between pointers and neighbors in the CSR data structure is higher.

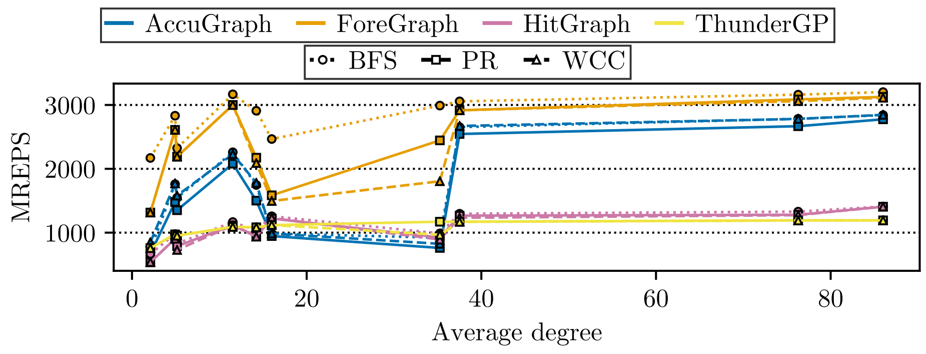


Fig. 4.18 Performance by average degree of the graph.

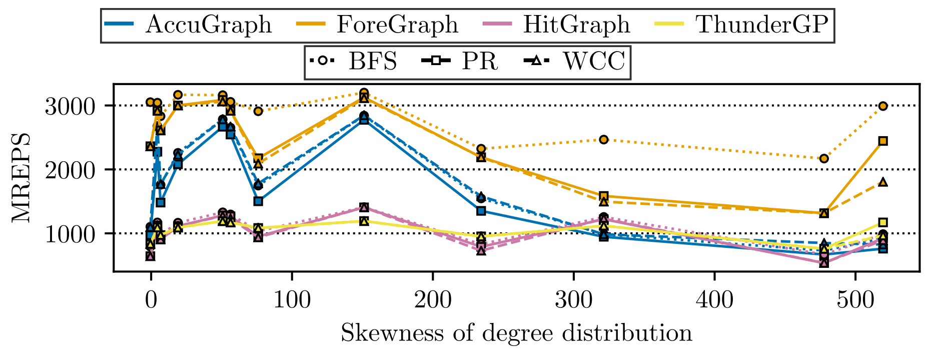


Fig. 4.19 Performance by skewness of degree distribution.

Figure 4.18 and Fig. 4.19 show raw edge processing performance in MREPS by average degree of the graph (D_{avg} in Tab. 4.2) and skewness of degree distribution.

We also observe performance differences by by skewness of degree distribution (cf. Fig. 4.19). Performance for AccuGraph and ForeGraph drops for graphs with high skewness (e. g., wiki-talk and twitter). The accelerators are only working at their full potential at low to moderate skewness. We also identify pipeline stalls caused by edge materialization on the CSR data structure as a problem for AccuGraph for high degree distribution skewness, similar to sparse graphs (*insight 5*). For ForeGraph we identify partition skew as the main cause of reduced performance which we discuss in more detail in Sect. 4.3.4.4 but can be observed in Fig. 4.17(d).

4.3.4.3 Memory Technology: DRAM Types

In Fig. 4.20(a), we show average speedup of DRAM types (DDR3 and HBM) over DDR4 for all four accelerators (performance dimension (iv)). We observe that modern memory (e. g., DDR4 or HBM) does not necessarily perform better

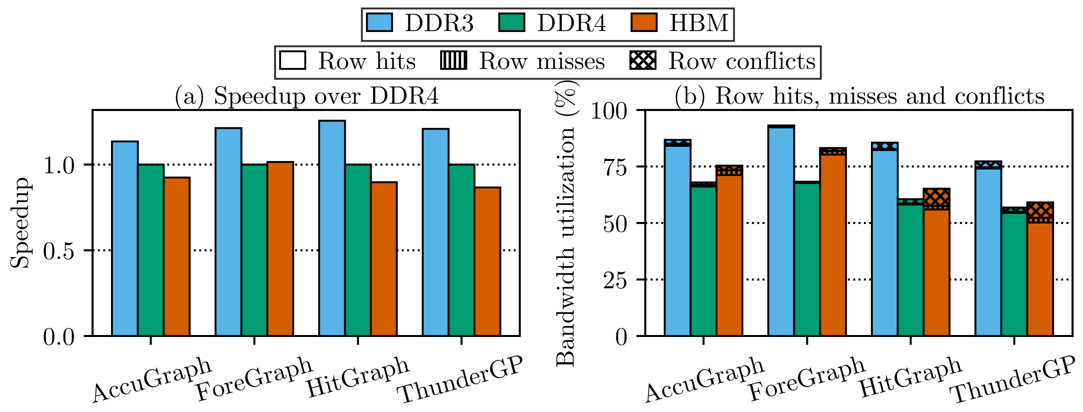


Fig. 4.20 DDR3 and HBM speedup over DDR4 and bandwidth utilization for BFS (single-channel).

than the older DDR3, despite higher theoretical throughput. This stems from lower bandwidth utilization and higher latency of requests (*insight 6*) which is explained by extremely low utilization of parallelism in the memory due to mostly sequential reading of very few data structures at once. AccuGraph and ForeGraph show more row hits due to write requests reusing rows in the row buffer of read requests (cf. Fig. 4.20(b)). To achieve very good bandwidth utilization, approaches have to utilize either even more locality (ForeGraph and AccuGraph) or more memory parallelism (AccuGraph with its CSR data structure). Additionally, there is no inherent benefit in using HBM for graph processing accelerators when the accelerator does not scale to multi-channel setups or scales poorly. The bandwidth utilization goes up slightly for HBM (cf. Fig. 4.20(b)) but at a cost of significantly more latency inducing row misses and conflicts due to HBM's smaller row buffers. Thus, we conclude that HitGraph and ThunderGP have higher potential adapting to HBM.

Figure 4.21 shows multi-channel scalability of HitGraph and ThunderGP. AccuGraph and ForeGraph are not enabled for multi-channel operation and thus excluded. For measurements on more than one channel, we assume the clock frequency (reported by the respective paper) achieved with the 4-channel designs for both HitGraph and ThunderGP for GraphSim simulation. However, as a limitation of these measurements this may not be exactly representative of the performance on real hardware because the clock speed could be slightly higher for two channels and lower for eight channels. For HitGraph, we see almost linear performance improvements when increasing the number of channels and see super-linear improvements for the roadnet-ca graph. This is due to improved effect

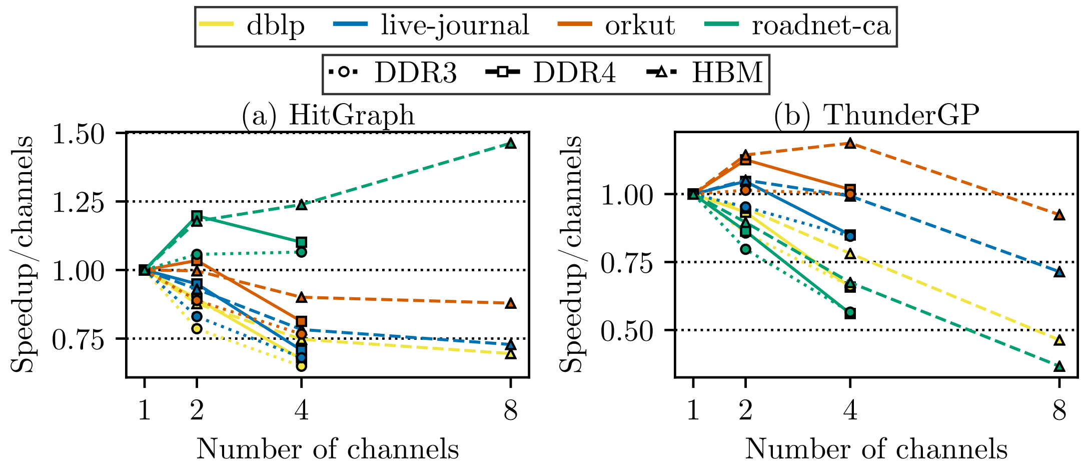


Fig. 4.21 Scalability over number of channels for BFS.

of partition skipping resulting in significantly less requests to memory (*insight 7*). For ThunderGP, we see mostly sub-linear improvement in performance. We explain the ThunderGP performance with its vertical partitioning scheme which leads to every PE working on values from all vertices such that all updates have to be applied to all channels, limiting performance (*insight 8*). However, the scaling seems to benefit from dense input graphs like the orkut graph. An effect that we also observe is that DDR4 performance scales very well for two channels due to better bank parallelism utilization compared to DDR3 and HBM leading to better latency.

Another point we want to highlight is that HitGraph scales linearly in memory footprint with the number of memory channels and ThunderGP scales sub-linearly. HitGraph needs $n+m+n$ while ThunderGP requires $n*c+m+n*c$ space in memory where n is the size of the vertex value array, m is the size of the edge array and c is the number of channels. This not only means higher memory usage but also number of reads and writes not scaling linearly with number of channels for vertical partitioning (*insight 9*).

4.3.4.4 Memory Access Optimizations

ForeGraph, Hitgraph, and ThunderGP propose a set of optimizations to reduce load on the memory or partition skew (performance dimension (v)). In the following, we describe the different optimization approaches.

ForeGraph has three optimizations. Edge shuffling (Shuf.) is a preprocessing step that repacks the edges such that the edge lists of p shards are zipped into

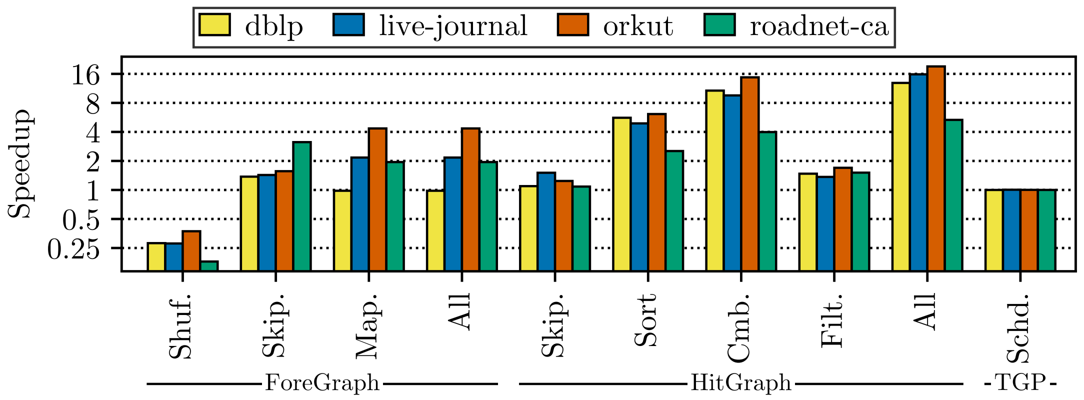


Fig. 4.22 Optimizations for BFS (TGP: ThunderGP).

one (where p is the number of PEs). This alone leads to reduced performance due to aggravated load imbalance with partitions (due to padding in the form of null edges) but improves PE utilization when combined with stride mapping (Map.). Shard skipping (Skip.) is employed for shards with unchanged source intervals compared to the previous iteration (equal to partition skipping for AccuGraph). Finally, stride mapping renames vertices such that intervals are sets of vertices with a constant stride instead of consecutive vertices. In total, the optimizations improve performance for all graphs we tested on (cf. Fig. 4.22). However, we observe lower than average performance for slashdot, dblp, and roadnet-ca. For those graphs, among others, the interval-shard partitioning introduces a lot of partition skew (especially in combination with edge shuffling) leading to many more edges read than necessary to process the graph (cf. Fig. 4.17(d)). Additionally, ForeGraph performance is higher for WCC on berk-stan and roadnet-ca when compared to BFS which is explained by less edges read due to more partition skipping (addition to *insight 7*).

HitGraph, like AccuGraph and ForeGraph, employs partition skipping (Skip.) with similar effectiveness. As a second optimization, HitGraph applies edge sorting by destination vertex (Sort), increasing locality to the gather phases value writing. The edge sorting for HitGraph prepares the data structure for update combining (Cmb.). Updates with the same destination vertex are combined into one in the shuffle phase which reduces the number of updates u from $u = |E|$ to $u < |V| \times p$ with number of PEs p. As a second optimization to update generation, a bitmap with cardinality n in BRAM saves for each vertex if its value was changed in the last iteration. This enables update filtering (Filt.) of updates from inactive vertices, saving a significant number of update writes.

ThunderGP proposes an offline scheduling of chunks to memory channels (Schd.) based on a heuristic predicting execution time. Chunks are greedily scheduled such that the overall predicted execution time is as similar as possible. Based on our measurements in Fig. 4.22 this however does not make a big difference. Additionally, ThunderGP does zero-degree vertex removal which we disabled for all runs because it is a pre-processing step applicable to all graph processing systems but hides performance ramifications of highly skewed degree distributions, e. g., for wiki-talk or rmat-21-86.

4.3.5 Insights & Open Challenges

In summary, from this comprehensive analysis of the four accelerators we gained nine insights that we categorize and group as trade-offs where possible in this section. Our biggest finding is a trade-off between lower iteration count of asynchronous graph processing for graph problems like BFS and WCC when compared to synchronous graph processing (*insight 1*, similarly on CPU [Wha+15]), and reading vertex values many more times for large graphs (*insight 3*). As an open challenge we propose finding an approach to reduce vertex value reads for asynchronous graph processing (e. g., similar to update filtering for HitGraph) to lower the impact of graph size on the accelerator performance (*open challenge (a)*). As a second trade-off we found that CSR significantly reduces bytes per edge and values read for small and dense graphs (*insight 2*) at a trade-off of reading more bytes per edge and values for large and sparse graphs when using horizontal partitioning (*insight 4*). As a last trade-off we found that large partitions reduce partition overhead while small partitions can significantly benefit partition skipping leading to super-linear performance increases (*insight 7*). We noticed that high skewness in degree distribution can lead to performance degradation, e. g., for accelerators using CSR or interval-shard partitioning (*insight 5*). Additionally, vertical partitioning leads to poor channel scalability (*insight 8*) and memory footprint for multi-channel setups (*insight 9*). Regarding modern memory (e. g., HBM), we saw that trading of more latency with higher bank-level parallelism does not necessarily lead to better performance (*insight 6*, generally for modern memory [Gho+19b]). Thus, we propose as an open challenge to investigate schemes to improve utilization of bank-level parallelism in modern

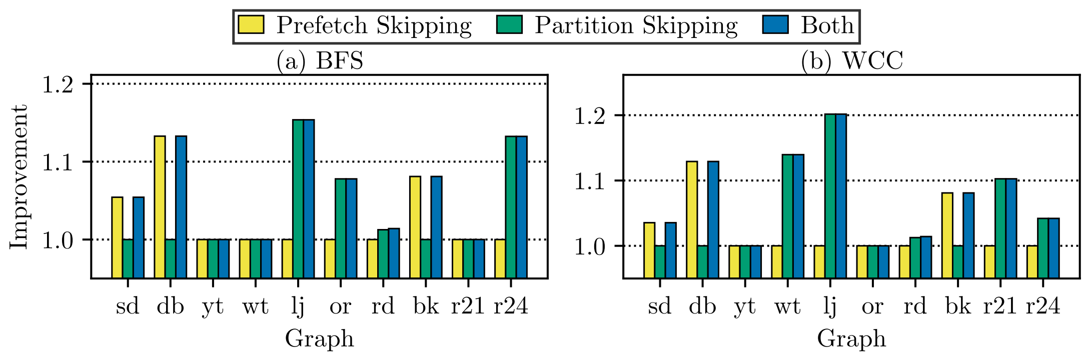


Fig. 4.23 Runtime improvement of optimizations over baseline.

memories (*open challenge (b)*). Lastly, we see and open challenge on enabling the asynchronous graph processing scheme for multi-channel (*open challenge (c)*).

4.4 Example: Faster Graph Accelerator Engineering

In this section, we illustrate how our approach helps to speed up graph processing accelerator engineering by example of two optimizations for AccuGraph that we found while analyzing the performance in the previous section. Note that instead of implementing the optimizations on the FPGA itself, our simulation approach is used to quickly assess the altered designs for the different data sets as well as potentially different DRAM types, thus reducing the overall engineering time by a form of rapid graph accelerator prototyping.

Optimization ideas AccuGraph writes all value changes through to off-chip memory and also applies them to BRAM if they are in the current partition. Thus, BRAM and off-chip memory are always in sync. Nevertheless, at the beginning of processing a partition, the value set is prefetched even if the values are already present in BRAM. Thus, the first optimization we propose is prefetch skipping in this case. For some graphs we also saw the effectiveness of partition skipping with ForeGraph and HitGraph (cf. Fig. 4.22). Thus as a second optimization, we propose adding partition skipping to AccuGraph. Both optimizations can easily be added to AccuGraphs control flow by directly triggering the value and pointer reading producers or completely skipping triggering of execution for certain partitions respectively. For prefetch skipping we compare the currently fetched partition with the next partition to prefetch and skip prefetching if they are the same. For partition skipping we keep track if any value of the vertices of a partition were changed and skip the partition if none changed. The optimizations also work in combination.

Results To prove their effectiveness, we measure the effect of both optimizations for BFS and WCC separately and combined (Fig. 4.23). For all small graphs with only one partition we see an improvement based on prefetch skipping. Partition skipping is not applicable to those graphs. For some other graphs we see an improvement based on partition skipping. Prefetch skipping only sometimes contributes a small improvement but only when combined with partition skipping. PR as a stationary algorithm is not shown, since no partitions can be skipped by definition. For prefetch skipping, there are similar performance improvements on PR compared to BFS and WCC. Overall we see no decrease in performance, suggesting that both optimizations should always be applied.

Discussion Note that these insights on the two enhancement ideas were possible in a relatively short amount of time, compared to engineering on an actual FPGA. Developing and verifying a complicated FPGA design usually takes weeks, while the implementation of a new graph accelerator approach in our simulation environment takes days or even just hours if the approach is well understood before. Additionally, the iteration time is much improved. Synthesis runs for compiling hardware description code to FPGA take hours up to a day without many possibilities of incremental synthesis, while a complete compilation of our simulation environment takes 33.5 seconds on a server with the possibility of easily utilizing parameters and incremental compilation. As a downside, the simulation runs longer than a synthesized design on an FPGA would. However, the user is not limited by special hardware that is only available in limited numbers (FPGAs). Many runs can be executed in parallel on one or even multiple servers. Especially for the very fragmented FPGA market, virtualized offers for FPGAs might not be available for specific boards.

4.5 Related Work

To the best of our knowledge, there is no prior related work on using the off-chip memory requests paired with a DRAM simulator to simulate and make graph processing accelerators more comprehensible and performance measurements reproducible and comparable.

Cache miss runtime estimation [MBK02] describe a cost model to approximate query runtimes in relational databases based on cache misses of memory requests. They focus on CPU cache hierarchies which allow much less fine-granular data placement than FPGA memory hierarchies (cf. Sect. 4.1.1). Additionally, they do not perform simulations of requests but model performance theoretically based on the model parameters of number of cache misses and cache latency not applicable to FPGAs.

Modern memory technologies In line with our findings on HBM, Schmidt et al. [SFB16] found that it is not trivial to attain good performance with the novel memory technology hybrid memory cube (HMC). Wang et al. also confirm in [Wan+20b] that it is crucial to consider how HBM should be used in FPGA-based accelerator designs.

Comprehensibility [ZCP15b] introduces a DRAM model and simulation for HitGraph. The simulation also generates the sequence of requests, but instead of simulating DRAM runtime, it assumes that every request results in a row buffer hit and models the performance along the cycles needed for processing the data and approximated pipelines stalls. However, they do not show performance numbers generated with this simulation. [Yan+19] uses Ramulator as the underlying DRAM simulator for a custom cycle-accurate simulation of the accelerator Graphicionado [Ham+16]. However, this incurs very high implementation time.

Reproducibility Regarding reproducibility, there is prior work on ways to report performance results such that it suits the own approach on parallel computing systems [Dav95; HB15]. The graph processing accelerator domain seems to suffer from similar problems and lack of widely accepted standards in benchmarking. Particularly, declarations of input parameters are often incomplete.

Comparability Ramulator [KYM16] was previously used in a work studying the interactions of complex workloads and DRAM types [Gho+19a]. They uncovered how the internal structure and characteristics of DRAM (DDR3 and DDR4 in our work) relate to performance gains or losses on otherwise fix benchmarks. This may be a future angle to improve graph processing accelerator performance by fitting the DRAM type to the algorithms and data sets. Similarly to our work, [Xu+17] raises awareness for lacking comparability in graph processing approaches on CPU-based cloud platforms. They find tradeoffs in approaches between different workloads and differently structured graphs.

4.6 Conclusion

In this chapter, we answer research question RQ2.1 *"Which crucial graph processing accelerator properties contribute to good performance?"*. To this end, we propose GraphSim, a simulation environment for graph processing accelerators. The simulation environment models request flow fed into a DRAM simulator (i. e., Ramulator [KYM16]) and control flow based on data dependencies. We develop a set of memory access abstractions and apply these to four representative graph processing accelerators: AccuGraph, ForeGraph, HitGraph, and ThunderGP.

Even though the simulation environment disregards large parts of the graph processing accelerator, we showed that it is able to reproduce ground truth measurements with a reasonable error for most workloads. In our analysis of the large errors for some workloads we found insufficiencies in benchmark setups and attribute some error to our radical approach. This chapter addresses an important shortcoming of graph processing accelerators, namely *comparability* of graph processing *performance*. We approach this matter by comparing the performance of four well-known graph accelerators (i. e., AccuGraph, ForeGraph, HitGraph, and ThunderGP) along performance dimensions relevant for graph processing (cf. dimensions (i)–(v)). We found performance effects based on accelerator design decisions (*insights 5, 8, 9*), issues in utilization of modern memory technologies (*insight 6*), and several interesting trade-offs (*insights 1–4, 7*). Additionally, we show that our simulation approach significantly reduces the iteration time to develop and test graph processing approaches for hardware accelerators by example of two optimizations for AccuGraph that we propose.

We propose to conduct future work on the identified *open challenges (a)–*

(c), i. e., further improving the asynchronous graph processing scheme for large graphs, leveraging the potential of HBM for graph processing, and multi-channel scalability of the asynchronous graph processing scheme. Additionally, we see a need for standardization of benchmark techniques in the field of graph processing accelerators, as sketched in this chapter.

Chapter 5

GraphScale: Scalable Bandwidth-Efficient Graph Processing on FPGAs

As we concluded in Chapter 4, Graph processing accelerators like AccuGraph [Yao+18] and FabGraph [Sha+19] utilize *graph compression* and *asynchronous graph processing* to reduce the load on the memory sub-system which is essential for good performance. For graphs with large average degree, a compressed graph data structure, like CSR, almost halves the number of Bytes per edge to be processed. Asynchronous graph processing, in turn, leads to a significant decrease in iterations over the graph. However, these approaches have not yet been *scaled to multiple memory channels*, as we show in Tab. 5.1. While AccuGraph and FabGraph enable the potential of a compressed graph data structure and asynchronous graph processing, they do not scale to multiple memory channels. In contrast, HitGraph [Zho+19] and ThunderGP [Che+22] scale to multiple memory channels but do not exploit that potential.

Thus, in this chapter, we address research question RQ2.2 *"How can the crucial graph processing accelerator properties be combined in one system?"*. We propose GraphScale, the first scalable graph processing framework for FPGAs based on asynchronous graph processing on a compressed graph. While, for asynchronous graph processing, the challenge is handling the high-bandwidth data flow of vertex label reads and writes to on-chip scratch pads at scale, the CSR-compressed graph adds design complexity and higher resource utilization for materializing compressed edges on-chip that is challenging for scaling to multiple

Table 5.1 Related FPGA-based graph processing accelerators with classification and feature set.

Identifier	Iter.	Partitioning	Data structure	Compr.	Async.	Scales	Labels	Frame.
AccuGraph [Yao+18]	Vertex	Horizontal	Inverse-CSR	👍	👍	👎	👍	👍
FabGraph [Sha+19]	Edge	Interval-shard	Compr. edge list	👍	👍	👎	👍	👍
HitGraph [Zho+19]	Edge	Horizontal	Sorted edge list	👎	👎	👍	👍	👍
ThunderGP [Che+22]	Edge	Vertical	Sorted edge list	👎	👎	👍	👍	👍
GraphScale	Vertex	Custom	Inverse-CSR	👍	👍	👍	👍	👍

Iter.: Iteration scheme; Compr.: Compressed; Async.: Asynchronous graph processing; Frame.: Framework; n/a: Not applicable; 👍: yes, 👎: no

memory channels. We tackle these challenges with a novel two level crossbar design to handle the data flow of vertex labels and a two-dimensional partitioning scheme to distribute the graph over the available memory channels, respectively. To support these, GraphScale features a vertex-centric iteration scheme for more flexibility in compression of the data structure than edge-centric and pull data flow that is beneficial for iterations with many vertex label updates which we expect especially for PR and WCC and more sequential memory accesses when compared to the push data flow. For the underlying data structure, we chose CSR because it is easy to construct with little preprocessing and allows for efficient sequential memory accesses, too.

GraphScale shows promising scalability with a maximum speedup of 4.77× on dense graphs and an average speedup of 1.86× over AccuGraph, FabGraph, HitGraph, and ThunderGP on a four DDR4 channel FPGA. Additionally, we address research question RQ2.3 *"How can accelerators be scaled to unlock the potential of novel memory technologies like HBM?"* and observe that scaling beyond four channels with modern high-bandwidth memory (HBM) provides a further speedup of 1.53× over GraphScale running on DDR4 memory. Overall, we conjecture that asynchronous processing on a compressed graph with multi-channel memory scaling improves the graph processing performance, however, leads to interesting trade-offs (e. g., for large graphs where partitioning overhead dominates performance). Motivated by this, we show how this partitioning overhead can be tackled with an inline binary packing decompressor with a resulting average performance improvement of 1.25× on synthetic graphs.

We start this chapter by introducing the design of GraphScale in Sect. 5.1 with a focus on our novel two level vertex label crossbar and novel two-dimensional partitioning scheme for graph processing on multiple memory channels. To tackle partitioning overhead on large graphs, we analyze which integer compression

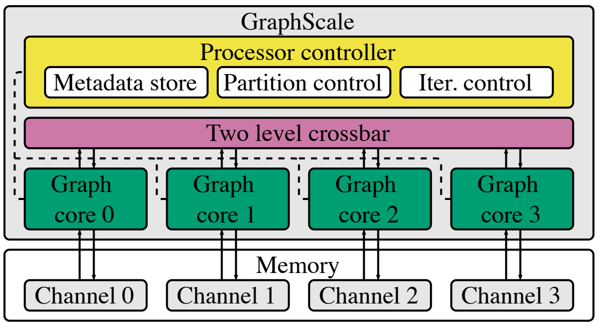


Fig. 5.1 GraphScale architecture.

technique is best suited to be implemented on the FPGA and provides the best compression ratio and propose a novel inline decompressor design based on binary packing compression in Sect. 5.2. In Sect. 5.3, we evaluated our approach compared to state-of-the-art graph processors AccuGraph, FabGraph, HitGraph, and ThunderGP an propose an OpenCL wrapper to enable scaling of GraphScale to HBM. We scale GraphScale to up to 16 channels of HBM and show the resulting performance improvements in Sect. 5.3.3, including a discussion of the technical limitations of scalability. In Sect. 5.3.4, we evaluate the binary packing decompressor but can only observe the performance improvements on synthetic graphs. For real-world graphs, inherent imbalances of requests to the GraphScale crossbar hide the performance improvements. Thus, we also propose a performance model based on a theoretical performance bound for memory and for the crossbar that is able to explain performance characteristics of GraphScale based on the provided graph (Sect. 5.3.5).

Parts of this chapter have previously been published in the proceedings of FPL 2022 [DRF22] (GraphScale base system for up to four memory channels) and ACM Transactions on Reconfigurable Technology and Systems [DRF23a] journal (binary packing decompressor and scaling to HBM).

5.1 GraphScale System Architecture

In this section, we describe our graph processing framework GraphScale (cf. Fig. 5.1) at an abstract processor-scale level and subsequently explain its design concepts. In principle, a GraphScale graph processor consists of p graph cores (explained in Sect. 5.1.1) matched to the number of p memory channels (four in this example). Each graph core is only connected to its own memory channel

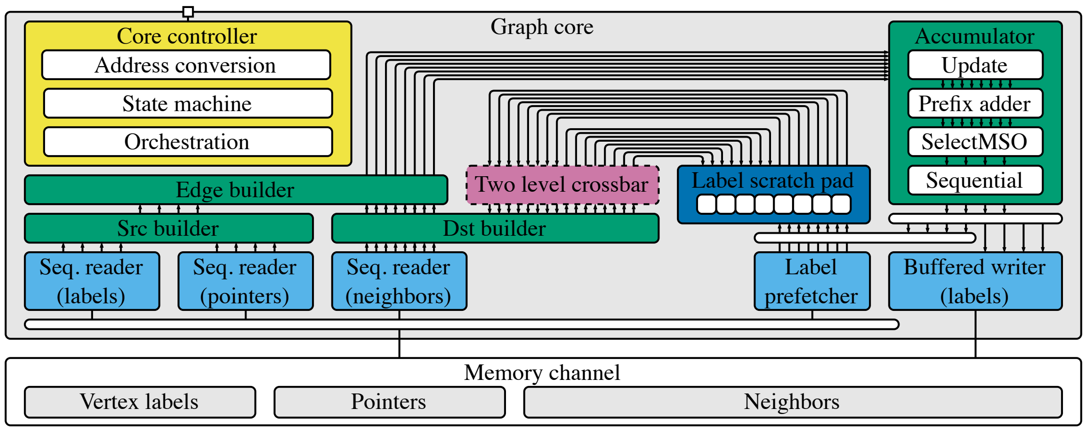


Fig. 5.2 Graph core architecture.

and can thus only directly read and write data on this channel. This requires partitioning of the graph into at least p partitions. The details of partitioning and how the graph is distributed over the memory channels is discussed in Sect. 5.1.3. However, since graph partitioning does not eliminate data dependencies between partitions, the graph cores are connected via a high-performance crossbar for exchange of vertex labels enabling the scaling of the approach. The crossbar will be explained in Sect. 5.1.2. The whole execution is governed by a processor controller. Before execution starts, the host code passes parameters for each partition and optimization flags to the processor controller which stores them in a metadata store. When execution is triggered by the host code, the processor goes through a state machine, orchestrating the control signals for the execution of iterations over the graph.

5.1.1 Graph Core

A graph core (cf. Fig. 5.2), as the basic building block of GraphScale, processes graphs based on the vertex-centric iteration scheme and pull data flow. It works on a partitioned inverse-CSR data structure of the graph (cf. Sect. 2.1.1) consisting of one vertex labels array and for each partition one pointers and neighbors array. Furthermore, processing of the graph is structured into two phases per iteration: prefetching and processing. In the prefetching phase, the vertex label prefetcher reads a partition specific interval of the vertex label array into the vertex label scratch pad, implemented as e (set to 8 in this example) banks of on-chip BRAM. The vertex label scratch pad is used to serve all non-sequential read requests that occur during an iteration instead of off-chip DRAM since BRAM has much higher bandwidth and single clock cycle latency.

Starting the data flow of the processing phase, the source builder reads vertex labels and pointers sequentially. Vertex labels and pointers are zipped to form v source vertices in parallel with a vertex index (generated on-the-fly), vertex label, inclusive left bound, and exclusive right bound of neighbors in the neighbors array each. The destination builder reads the neighbors array of the current partition sequentially and puts e neighbor vertex identifiers in parallel through the two level crossbar passing the vertex identifiers to the correct label scratch pad bank of the correct graph core and returning the resulting vertex labels in the original order (discussed in more detail in Sect. 5.1.2). The vertex label annotated with the neighbor index is then passed to the edge builder which combines source and destination vertices based on the left bound l and right bound r of the source vertex and the neighbor index j of the destination vertex as $l \leq j < r$. Thus, we get up to e edges with a maximum of v source vertices as output from the edge builder per clock cycle.

The accumulator takes the e edges annotated with their source and destination vertex labels as input and updates vertices in four steps. First, updates are produced in the update stage depending on the graph problem's update function for each edge in parallel. For BFS, this means taking the minimum of the source vertex label and destination vertex label plus 1. If the latter is smaller, the output is flagged as an actual update of the source vertex label. This is crucial for algorithms that terminate when no more updates are produced in an iteration (e. g., BFS and WCC). The pairs of source vertex identifier and updated vertex labels are then passed to the prefix adder which reduces the updates to the most significant element with the same source vertex identifier for each source vertex. The most significant entry is then selected by the v selectors in the SelectMSO stage of the accumulator and passed on to the last stage. Each selector already only selects pairs with $i \mod v = I$ for source vertex index i and selector index I. The sequential stage consists of v sequential operators that reduce updates to the same vertex from subsequent cycles into one. The accumulated update is put out when a new source vertex identifier is encountered or a higher source vertex identifier is encountered. Thus, in total, the accumulator produces updates only when the new label is different based on the annotated edges and reduces them into a maximum of one update per source vertex.

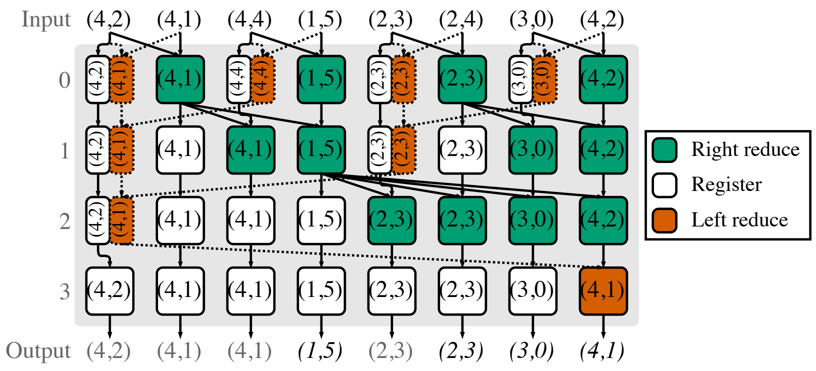

Input (4,2) (4,1) (4,4) (1,5) (2,3) (2,4) (3,0) (4,2)
0
1
2
3
Output (4,2) (4,1) (4,1) (1,5) (2,3) (2,3) (3,0) (4,1)
Right reduce
Register
Left reduce

Fig. 5.3 Prefix adder with wrap-around suffix sub-accumulator (with minimum reduce operator).

Figure 5.3 shows the parallel prefix-adder vertex-update accumulator. We add a suffix sub-accumulator (dotted outlines) necessary to attain correct results in some edge cases and a merged signal for non-idempotent reduce operators like summation. The accumulator takes e pairs of source vertex identifier and updated vertex label (split with a comma) and returns one updated vertex label as the right-most identifier-label pair per incoming source vertex identifier (italicized). The prefix-adder accumulator consists of $\log_2(e)+1$ pipelined levels of registers (white) and reduce processing elements (PE). The registers take one identifier-label pair as an input and pass this value on to the next level in the next clock cycle. The reduce PEs (green and orange) take two identifier-label pairs as an input and combine them according to a user defined function if the source vertex identifiers are equal. The result is again put out in the next clock cycle. Right reduce PEs (green) pass on the right identifier-label pair unmodified if the identifiers are unequal and left reduce PEs (left) pass on the left pair. In this particular example, the parallel accumulator could either be used e. g., for BFS or WCC because it uses minimum reduce PEs which put out the minimum of the vertex labels if they should be combined. The connection pattern of the first $\log_2 e$ levels of the accumulator represent a Ladner-Fischer prefix-adder.

Additional to the prefix adder, we also introduce a suffix sub-adder which reduces all identifier-label pairs with source vertex identifier equal to the first one to the first element. In an additional pipeline step, this suffix accumulation result is reduced with the last prefix accumulation result if there have been multiple different source vertex identifiers in the input. We do this because the sequential source vertex identifiers can overlap from the last one to the first one as a result of how the edge builder works. In this edge case updated vertex labels might be missed because only the right-most vertex label of a source vertex identifier is further processed. Finally, we only reduce two identifier-label pairs if all pairs in between have the same source vertex identifier which we keep track of with a merged signal mentioned above.

The resulting updates are fed back to a buffered writer and into the label scratch pad so they can immediately be used in the same iteration. The buffered writer collects all updates to the same cache line and writes them back to memory when an update to a new cache line is encountered.

All different parts of this design are orchestrated in their execution by a core controller. The core controller gets graph-wide parameters of the execution like number of vertices, number of edges and address of the buffer in memory and dynamic parameters like iteration number from the processor controller. Based on this, it starts the prefetch phase and then the processing phase and therefore calculates the addresses to the data structure arrays. Finally it also flushes the pipeline so all updates are written back to memory before asserting the ready signal such that the next iteration can be started.

5.1.2 Scaling Graph Cores – Vertex Label Crossbar

To scale this very effective single-channel design with limited overhead, we propose the graph core to memory channel assignment shown in Fig. 5.1. Each graph core works on exactly one memory channel. However, annotating edges with vertex labels requires communication between graph cores. Therefore, we propose a scalable resource-efficient two level crossbar. In this section, we will describe how we achieve the necessary high throughput of this crossbar to saturate the accumulators of multiple graph cores with annotated edges.

We show the multi stage design of the crossbar for two cores and $e = 4$ in Fig. 5.4. The first level (bank shuffle) receives e neighbors from the destination builder each cycle for each core with a FIFO buffer for each lane in between to reduce stalls. The neighbor indices serve as addresses to vertex labels in the labels array. Before the processing of a partition starts, the partition's vertex label are prefetched to the label scratch pad. Since the on-board memory returns e neighbors per graph core per cycle at maximum throughput, $e \cdot p$ requests have to be served by the label scratch pad per cycle. The label scratch pad of each

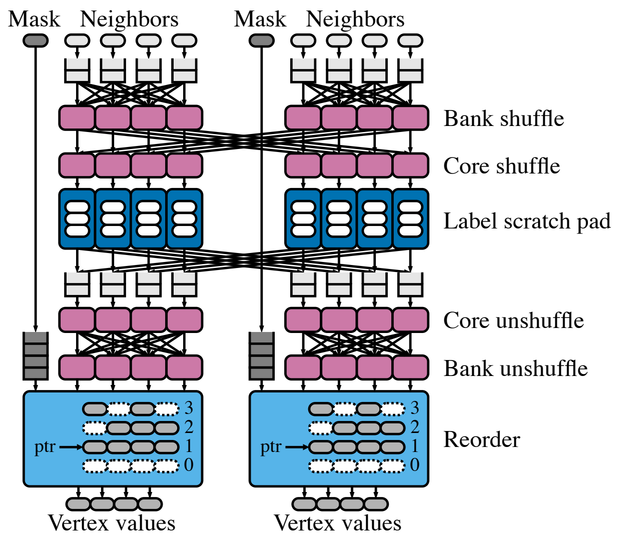


Fig. 5.4 Stages of scalable two level crossbar.

graph core is split up into b banks which is at least equal to e (larger values for b may be used to reduce contention on banks) that can serve requests in parallel. The vertex labels are striped over these banks. The last $\log_2 b$ bits of each neighbor index are used to address the bank of the label scratch pad that this vertex label can be requested from. Thus, the bank shuffle level puts each neighbor index into the right bank lane based on its last $\log_2 b$ bits. This can introduce stalls because multiple neighbors from one line can go to the same bank (i. e., a multiplexer has to output entries from the same input cycle in multiple output cycles). However, since each neighbor only goes to one bank we decouple the b bank shufflers and let them consume r full lines before stalling such that labels from later lines can overtake in other banks. For most graphs, this reduces stalls because the load is approximately balanced between banks.

In a second level, we introduce a core crossbar that shuffles the neighbor indices annotated with their line and lane they originally came from to the core that contains the vertex label. Core addressing is done by the first $\log_2 p$ bits of the neighbor index. However, since the neighbor indices are already in the correct lane, this only requires $p \cdot b$ core shufflers with p inputs. The results are additionally annotated with the core they originally came from and fed into the label scratch pad. The core shufflers work independently from each other too, allowing neighbor indices to overtake.

The banked vertex label scratch pad (each bank is 2^c entries deep) returns the vertex labels with a one clock cycle latency but keeps the annotations. A

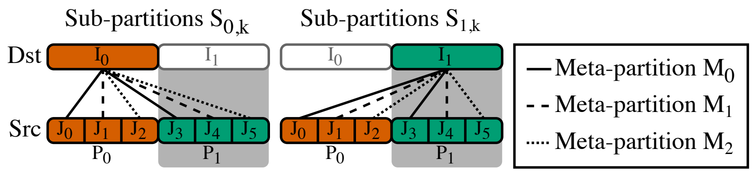


Fig. 5.5 Two-dimensional partitioning (two cores and three sub-intervals).

second layer of core shufflers routes the vertex labels back to their original graph core. Thereafter, the vertex labels are unshuffled to the lane they originally came from and fed into a final reorder stage to restore the original sequence of the data. The sequence possibly changes because requests and responses overtook each other in the previous steps.

The reorder stage has a fixed number of lines it can keep open at the same time (4 in this example) which we will call reorder slots. It is passed the valid signals of the incoming neighbors when they first enter the crossbar and puts them into a mask FIFO to check if all responses are present. The unshuffled labels are then still annotated with the line they originally came from modulo the number of reorder slots which is used as the address to put them in a BRAM. There is one BRAM for each lane of reorder slots because in each cycle we possibly write one label and read one label per lane. The reorder stage also maintains a pointer pointing to the next output line and compares the valid signals of this line to the mask FIFO output. If the mask FIFO valid output and the valid signals form the line are equal, the labels are put out, the pointer is incremented, the mask FIFO is popped, and the valid signals of the line are cleared. When the pointer is incremented above the last line it overflows to 0.

Finally, the mask FIFO of the reorder stage is also used to exert backpressure. If the mask FIFO contains as many elements as there are reorder slots, the ready signal is deasserted and all stages stop. To handle the one cycle latency of the label scratch pad there is also an additional overflow register where the label scratch pad result can overflow to.

5.1.3 Graph Partitioning and Optimizations

Figure 5.5 shows the partitioning of the input graph as the last missing part of our graph processing accelerator GraphScale and why it is able to scale well. The partitioning in done in two dimensions. In the first dimensions, the set of vertices is divided into p equal intervals I_q (I_0 and I_1 in this examples for $p = 2$), one stored on each memory channel and processed by its corresponding graph core P_q. The second dimension of partitioning divides each vertex interval into l equal sub-intervals (J_0 to J_5 in this example for $l = 3$) that fit into the label scratch pad of the graph core. We generate one sub-partition $S_{i,j}$ for each pair of interval I_i and sub-interval J_j containing all edges with destination vertices in I_i and source vertices in J_j and rewrite the neighbor indices in the resulting CSR data structure such that the requests are shuffled to the correct graph core by the two level crossbar (i. e., first $\log_2 p$ bits are graph core index) and subtract the offset of the respective sub-interval J_j. Sub-partitions $S_{q,q \cdot l}$ for each $q \in [0,q)$ additionally form a meta-partition M_q. During execution, all sub-intervals $J_{q \cdot l}$ are prefetched by their respective graph core q before processing of all sub-partitions of meta-partition M_q is triggered. During processing, edges whose source and destination vertex reside in different intervals I_i produce a data dependency between partitions and thus graph cores such that a vertex label has to be communicated over the two-level crossbar to process that edge. This light-weight graph partitioning, however, may introduce load imbalance because it works on simple intervals of the vertex set.

Each graph core writes all label updates to off-chip memory through the buffered writer while processing a partition. As a first optimization, immediate updates, we also immediately write back the updates to the label scratch pad if they are part of the current partition [Yao+18]. Thus, with this optimization, BRAM and off-chip memory are always in sync. Nevertheless, at the beginning of processing a partition, the vertex label set is unnecessarily prefetched even if the labels are already present in BRAM. Thus, we utilize prefetch skipping in this case as a light-weight control flow optimization which skips the prefetch phase if the vertex label set is already present in the label scratch pad [DRF21b]. This optimization only works in conjunction with immediate updates. As a third optimization, we apply stride mapping to improve partition balance which we identified as a large issue during testing [Dai+17]. Because the graph cores work in lock-step on the meta-partitions, imbalanced partitions lead to a lot of idle time. Stride mapping is a light-weight technique for semi-random shuffling of the vertex identifiers before applying the partitioning which creates a new vertex ordering with a constant stride (100 in our case which results in $v_0, v_{100}, v_{200}, \ldots$).

Table 5.2 Integer compression techniques with feature set important to implementation of FPGAs.

Identifier	Non-recursive	No dictionary	Blocked
Variable length	👍	👍	👎
Simple-8 [AM10]	👍	👍	👍
Binary packing [AM10]	👍	👍	👍
PFor [Zuk+06]	👍	👍	👍
Golomb [RP71]	👍	👍	👎
Huffman [Huf52]	👍	👎	👎
Interpolated [MS00]	👎	👍	👎
RePair [LM00]	👎	👎	👎

👍: yes, 👎: no

5.2 Pointer Array Compression

In this section, we first show an analysis of the different compression techniques (cf. Tab. 5.2) to determine which technique provides the highest compression ratio and thereafter present the GraphScale inline decompressor subsystem based on binary packing compression.

5.2.1 Integer Compression Techniques

Table 5.2 shows a selection of integer compression techniques that are often mentioned in related work and their feature set. To make sure that inline decompression at line rate can be achieved on the FPGA for the compression technique we choose, we only pick those that are not recursive (meaning we do not have to do multiple passes over the data to get the decompressed integers) and do not require a dictionary (which would require BRAM resources that are already very limited). This excludes Huffman [Huf52], Interpolated [MS00], and RePair [LM00] compression. Additionally, we prefer compression techniques which work with blocks to avoid data dependencies between values and simplify decompression of many values per clock cycle. All of these techniques can additionally be combined with difference encoding if the original sequence of integers is sorted. Difference encoding means that every encoded value v'_i represents the difference of its original value v_i to its previous value v_{i-1}. Subsequently, we will shortly introduce the five remaining compression techniques (variable length, Simple-8, binary packing, PFor, and Golomb).

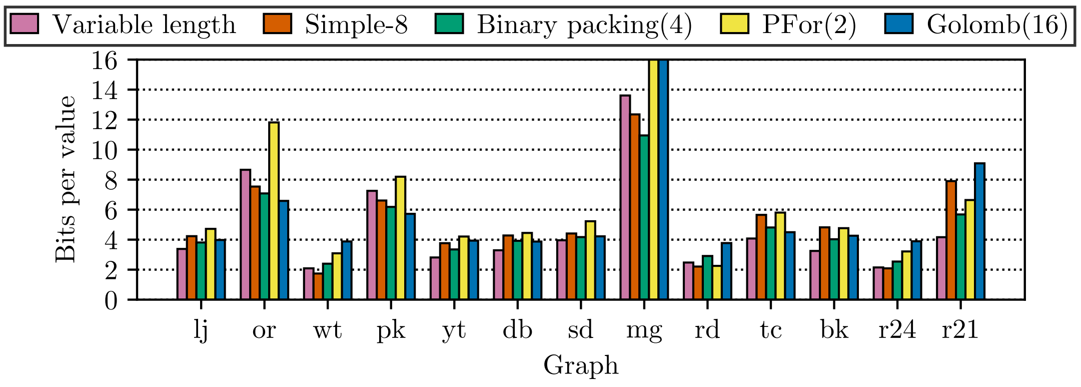


Fig. 5.6 Compression ratio of pointers array for different compression techniques.

Variable length For each value, there is a prefix of ones of the same length as the subsequent packed value. This works well for very small values but can waste a lot of bits for large values.

Simple-8 As many values as possible are packed into 64 bit blocks where the first four bits of each block are used to determine the width of the individual values [AM10]. One drawback is that this technique possibly wastes bits as padding but all blocks are fixed width making FPGA implementation easier.

Binary packing and PFor Binary packing [AM10] takes a fixed number of values as a block and determines the maximum of the minimum number of bits needed to represent each value. Each block contains a header with this bit width and all values encoded with that bit width. PFor [Zuk+06] works similar to binary packing but the header of each block additionally stores a base value against which each value is diffed. There are a number of possible optimizations that we do not cover here.

Golomb Each value is divided by a parameter b where the quotient is encoded in unary code and the remainder is encoded into $\log_2(b)$ bits binary code [RP71].

5.2.2 Analysis

In Fig. 5.6, we show a benchmark of the five compression techniques (variable length, Simple-8, binary packing, PFor, and Golomb) from Tab. 5.2 that we determined to be suitable for integration into GraphScale. On the x-axis we show the different graphs that we also use in our performance benchmarks in Sect. 5.3

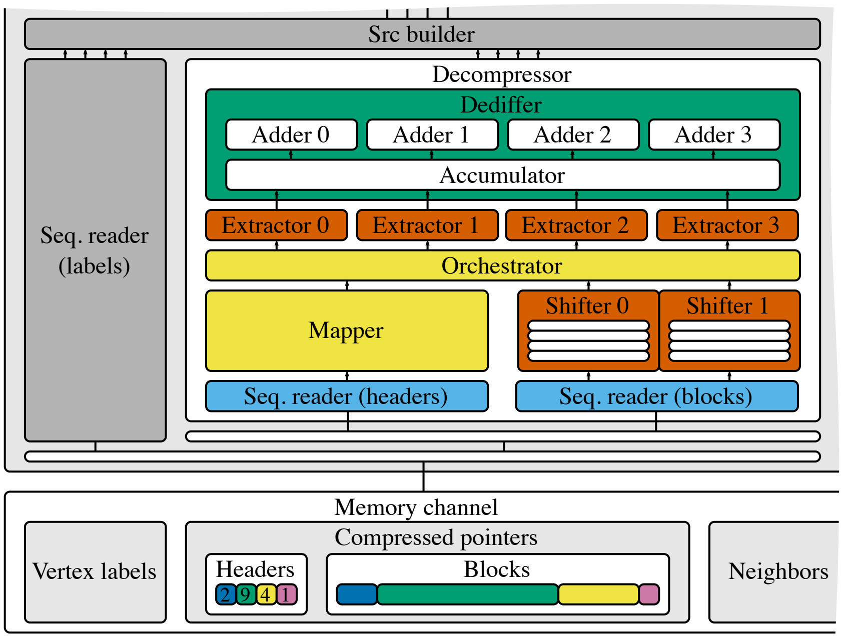


Fig. 5.7 Inline binary packing decompressor subsystem.

and on the y-axis we show the number of bits that are required per value in the pointers array after compression. We aim for lowest average bits per value as this provides the best compression ratio. For PFor and binary packing, we tried out different block sizes with 2 and 4 respectively being the optimum. For Golomb we determined $b = 16$ to provide the best results. Overall, we confirm that the compression techniques are most effective for very sparse graphs and large graphs (e. g., wiki-talk and rmat-24-16) and least effective for very dense graphs (e. g., mouse-gene). Variable length and binary packing perform the best with almost the same average. Thus, due to variable length not supporting blocks, we chose to implement binary packing.

5.2.3 Concept

Figure 5.7 shows our inline decompressor design for binary packed pointers as a cutout of the lower left part of the graph core from Fig. 5.2. The parts that stay the same are greyed out. We replace the sequential reader for pointers with the inline decompressor. For the memory layout, we split off the headers of the blocks into their own array. Otherwise it would require a lot more logic to just extract the headers from the blocks because of the variable block length in bits. From the bottom up, on the left side of the decompressor, the headers are read and fed into a mapper. To make the compression even more effective and make header reading easier, each header is only four bits wide. However, pointer values can be up to 32 bits wide such that we expand the 16 possible values of the four bit wide headers to five bits. All values i are mapped to $i+1$ besides the value 15 which is mapped to 32, which we determined to be the best distribution. On the right side, the decompressor reads the blocks as full lines of memory and puts two of them through two multi-stage shifters (similar to the shifter in [Sie+19]). Shifter 0 shifts its memory line such that position 0 of the result contains the first bit of the to be extracted block. If the block spans two memory lines, shifter 1 additionally shifts the next memory line such that it can be joined with the result of shifter 0. The orchestrator takes headers from the mapper and keeps track of the next block position. With that information it controls the shifters and pops memory lines if all blocks have been processed. The orchestrator then feeds the joined memory line from shifter 0 and shifter 1 and the mapped header to a number of extractors matching the number of vertex pipelines. Each extractor extracts its corresponding value from the memory line and feeds into a dediffer. The dediffer accumulates the values from the extractors in multiple stages and at the end adds them to the current value to attain the original data.

5.3 Evaluation

In this section, we introduce the system used for evaluation and setup for the performance measurements together with the system parameters and the graph data sets used for the experiments. Thereafter, we comprehensively explore performance in multiple dimensions. First, we look at the effects of the optimizations introduced in Sect. 5.1.3, before highlighting the scalability of the GraphScale framework. We compare GraphScale with the performance of the competitors that do not scale to multiple memory channels (i. e., AccuGraph and FabGraph) to show the scalability of GraphScale and those that do scale (i. e., HitGraph and ThunderGP) to show the advantages and disadvantages of asynchronous graph processing and a compressed graph. Next, we propose an OpenCL wrapper for GraphScale and show the resulting scalability to an HBM-enabled FPGA. Lastly, we show the performance improvements from the binary packing compression and propose a performance model to explain data-dependent performance characteristics of GraphScale.

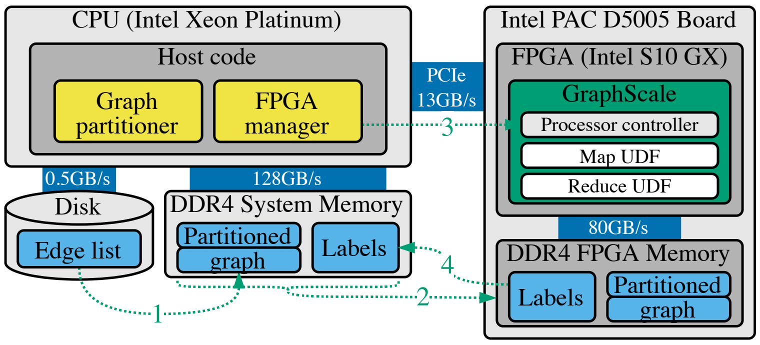


Fig. 5.8 Overall system architecture (incl. host, device, and memory).

Figure 5.8 shows the system context in which GraphScale is deployed. In principle, the system features a CPU and an accelerator board which hosts the FPGA, running GraphScale itself, and memory, used as intermediate data storage for the graph during processing. The CPU manages the execution on the FPGA and is also responsible for loading and partitioning the graph. To execute a particular workload with a particular graph, the GraphScale framework, first, is synthesized with user defined functions (UDFs) for the map and reduce operators in the accumulator. Map produces an update to the source vertex label for each edge, while reduce aggregates updates into one value for each to be updated vertex. For a switch from BFS to WCC, the reduce UDF stays the same while only one line has to be changed for the map UDF. PR requires changing the map UDF significantly and replacing the reduce UDF with summation. Additionally, PR alternatingly works on two separate vertex label arrays. Secondly, the synthesized design is programmed to the FPGA.

For execution of the programmed algorithm on a particular graph data set, the edge list (or any other representation) of the graph is read from disk to the CPU and partitioned by the graph partitioner in the host code according to the GraphScale framework parameters. Additionally, the vertex labels of the graph are initialized with graph problem specific values. The graph partitions and vertex labels are then transferred to the respective channels of the FPGA memory. Thereafter, the parameters of the graph are passed to GraphScale via a control interface which triggers execution. After the execution finished, the results can be read back to CPU memory and used for further processing. If desired, the partitioned graph can be used multiple times in a row by loading new vertex labels and again triggering the execution.

For our experiments, we are working with a server equipped with an Intel

Table 5.3 Resource utilization and clock frequency by system, graph problem, and system parameters (D5005: 933.1K LUTs, 1.87M registers, 30.5 MB BRAM, 5760 DSPs; S10MX: 702.7K LUTs, 1.41M registers, 17.5 MB BRAM 3960 DSPs).

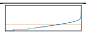
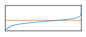
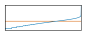
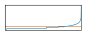
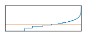
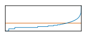
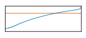
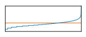
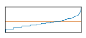
System	Platform	Problem	p	b	c	r	LUTs	Regs.	BRAM	DSPs	Clock freq.
GraphScale	D5005	BFS	1	16	17	4	19%	13%	40%	0%	192 MHz
			2	16	16	4	30%	23%	41%	0%	186 MHz
			4	16	15	4	58%	47%	43%	0%	170 MHz
		PR	1	16	17	4	26%	14%	66%	<1%	174 MHz
			2	16	16	4	43%	43%	67%	<1%	162 MHz
			4	16	15	4	82%	69%	72%	1%	143 MHz
		WCC	1	16	17	4	20%	14%	40%	0%	191 MHz
			2	16	16	4	30%	23%	41%	0%	183 MHz
			4	16	15	4	55%	45%	43%	0%	161 MHz
GraphScale-HBM	S10MX	BFS	1	16	17	1	28%	25%	71%	0%	248 MHz
			2	16	16	1	30%	28%	72%	0%	256 MHz
			4	16	15	1	41%	38%	75%	0%	241 MHz
			8	16	14	1	65%	64%	80%	0%	186 MHz
			16	8	13	0	75%	78%	58%	0%	142 MHz
		PR	1	16	16	1	27%	26%	64%	<1%	211 MHz
			2	16	15	1	34%	34%	65%	<1%	211 MHz
			4	16	14	1	48%	50%	68%	<1%	203 MHz
			8	16	13	1	89%	91%	74%	2%	148 MHz
		WCC	1	16	17	1	28%	24%	71%	0%	248 MHz
			2	16	16	1	30%	29%	72%	0%	257 MHz
			4	16	15	1	39%	39%	75%	0%	249 MHz
			8	16	14	1	65%	66%	80%	0%	210 MHz
			16	8	13	0	75%	78%	58%	0%	137 MHz
GraphScale-BinPack	D5005	BFS	1	16	17	1	16%	9%	40%	0%	180 MHz

LUTs: Look-up-tables; Regs.: Registers; BRAM: Block RAM; DSPs.: Digital signal processors; Clock freq.: Clock frequency

FPGA Programmable Accelerator Card (PAC) D5005 attached via PCIe version 3. The system features two Intel Xeon Gold 6142 CPUs at 2.6 GHz and 384 GB of DDR4-2666 memory. The D5005 board is equipped with four channels of DDR4-2400 memory with a total capacity of 32 GB and a resulting bandwidth of 76.8 GB/s. The design itself is based on the Intel Open Programmable Execution Engine (OPAE) platform and is synthesized with Quartus 19.4.

For the HBM experiments, we are working with a workstation equipped with an Intel FPGA S10MX development kit attached via PCIe version 3. The system features an Intel Xeon Gold E5-2667 CPU at 2.9 GHz and 96 GB of DDR3-1600 memory. The S10MX development kit board is equipped with 32 channels of HBM 2 memory with a total capacity of 8 GB and a resulting bandwidth of up to 512 GB/s. GraphScale itself is embedded into the Intel OpenCL platform as a kernel and is synthesized with Quartus 21.2.

Table 5.4 Graphs used often by systems in Tab. 5.1 (real-world graphs from [LK14] and [RA15]; Graph500 generator for R-MAT).

Name	$\lvert V\rvert$	$\lvert E\rvert$	Dir.	Degs.	D_{avg}	ø	SCC
live-journal (lj)	4.8M	69.0M	👍		14.23	20	0.79
orkut (or)	3.1M	117.2M	👎		76.28	9	1.00
wiki-talk (wt)	2.4M	5.0M	👍		2.10	11	0.05
pokec (pk)	1.6M	30.6M	👎		37.51	14	1.00
youtube (yt)	1.2M	3.0M	👎		5.16	20	0.98
dblp (db)	426.0K	1.0M	👎		4.93	21	0.74
slashdot (sd)	82.2K	948.4K	👍		11.54	13	0.87
mouse-gene (mg)	45.1K	14.5M	👎		643.26	11	0.95
roadnet-ca (rd)	2.0M	2.8M	👎		2.81	849	0.99
top-cats (tc)	1.8M	28.5M	👍		15.92	258	1.00
berk-stan (bk)	685.2K	7.6M	👍		11.09	714	0.49
rmat-24-16 (r24)	16.8M	268.4M	👍		16.00	19	0.02
rmat-21-86 (r21)	2.1M	180.4M	👍		86.00	14	0.10
perfect-v-e (p-v-e)	2^v	$2^v \cdot e$	👍	-	e	-	1.00

Dir.: Directed; Degs.: Degree distribution on log. scale; SCC: Ratio of vertices in the largest strongly-connected component to n; 👍: yes, 👎: no

Table 5.3 shows the different system configurations used for our experiments for GraphScale as well as AccuGraph, FabGraph, HitGraph, and ThunderGP. Besides the three graph problems BFS, PR, and WCC, we synthesized system variants which utilize different numbers of memory channels p. All DDR4-based variants (i. e., GraphScale on the D5005 platform) have a total vertex label scratch pad size of 2^{21}, 16 scratch pad banks, and 8 vertex pipelines. All types including pointers, vertex identifiers, and vertex labels are 32 bit unsigned integers, except PR vertex labels which are 64 bit and consist of the degree of the vertex and its PR value. Lastly, the depth of the reorder stage is set to 32. For HBM, we synthesized system variants up to 16 memory channels except for PR which did not fit on the FPGA. All variants except those for 16 channels have a total vertex label scratch pad size of 2^{20}, 4 vertex pipelines, and an overprovisioned crossbar with 16 vertex label scratch pad banks. The GraphScale-BinPack variant implements binary packing compression. This parameterization results in a moderate resource utilization with rising LUT and register utilization and almost constant BRAM utilization because vertex label scratch pad size is shared between the graph cores. The PR configuration has significantly higher resource utilization due to the doubled vertex label size.

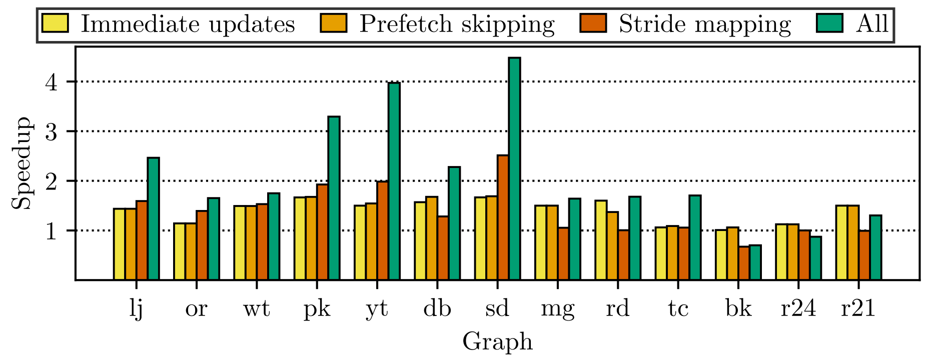


Fig. 5.9 Effects of GraphScale optimizations for BFS.

Graph data sets that are used to benchmark our system are listed in Tab. 5.4. This selection represents the most important graphs, currently considered, found by a recent survey [DRF23b]. Two important aspects when working with these graphs are their directedness and the choice of root vertices[1] (e. g., for BFS or SSSP), because they can have a significant impact on performance. We also show graph properties like degree distribution and average degree that are useful in explaining performance effects observed in the following. For the binary packing compression experiments, we generated different configurations of the perfect-v-e graph. This graph is generated in a way to perfectly utilize the GraphScale crossbar to isolate graph loading from memory.

5.3.1 Effects of GraphScale Optimizations

Figure 5.9 shows the effects of different optimizations from Sect. 5.1.3, when applied to the base framework. The measurements are performed on a four channel GraphScale system and normalized to measurements with all optimizations turned off. The *immediate updates* optimization ensures that updates to the vertex labels of the current partition interval are written back to the scratch pad immediately, instead of just being written back to memory. This makes updates available earlier and leads to faster convergence for almost all graphs. Only the berk-stan graph does not benefit from this optimization which is due to a specific combination of graph structure and selected root vertex. The *prefetch skipping* optimization skips the prefetch phase of each iteration if intermediate updates are enabled. Hence, the prefetch skipping measurements

[1]Root vertices used: lj - 772860; or - 1386825; wt - 17540; pk - 315318; yt - 140289; db - 9799; sd - 30279; mg - 20631; rd - 1166467; tc - 1405263; bk - 546279; r24 - 535262; r21 - 74764

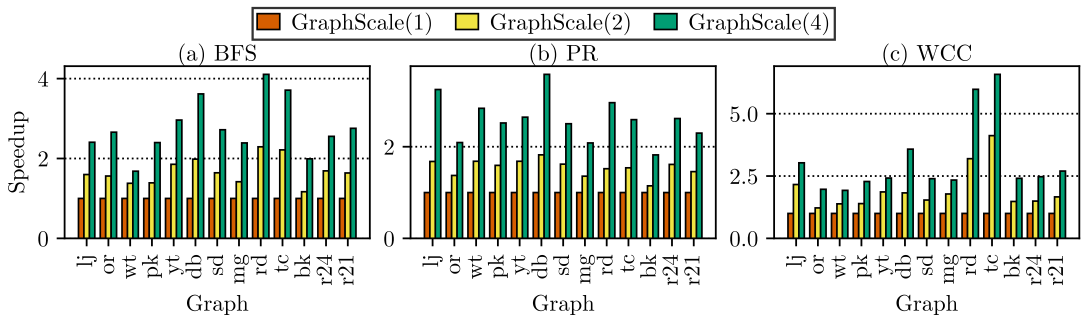


Fig. 5.10 GraphScale memory channel scalability from one to four channels.

have intermediate updates enabled. Additionally, prefetch skipping only works on graphs with a single partition. Prefetch skipping is a lightweight control flow optimization that sometimes leads to small performance improvements. Lastly, *stride mapping* tries to optimize partition balance. Whenever partitions can be balanced (e. g., youtube or slash-dot graphs), the performance improves most significantly. However, in rare cases (e. g., berk-stan graph) this may lead to performance degradation because with asynchronous graph processing, result convergence is dependent on vertex order and a beneficial vertex order may be shuffled by stride mapping. From our observation, it is beneficial if high degree vertices are at the beginning of the vertex sequence for faster convergence. In single channel measurements, single channel performance was better without stride mapping for almost all graphs. This is expected because partition balance is only important between channels but not between sub-partitions.

5.3.2 GraphScale Scalability

Figure 5.10 shows the scalability of GraphScale from a single-channel up to four memory channels as speed-up over the baseline of single-channel operation for BFS, PR, and WCC. For single-channel, the stride mapping optimization is disabled. Otherwise, all optimizations discussed in Sect. 5.1.3 are always enabled. The measurements show that there is some scaling overhead and speedup is dependent on the data set. This may be due to partition balance but is mainly influenced by density (i. e., average degree) of the graph for BFS. This can e. g., be observed for the orkut, dblp, and rmat-21-86 graphs. Two interesting exceptions are the roadnet-ca and top-cats graphs which show super-linear scaling. This is due to stride mapping changing the vertex ordering and thus leading to convergence on the result in significantly less iterations. Scalability speedups for

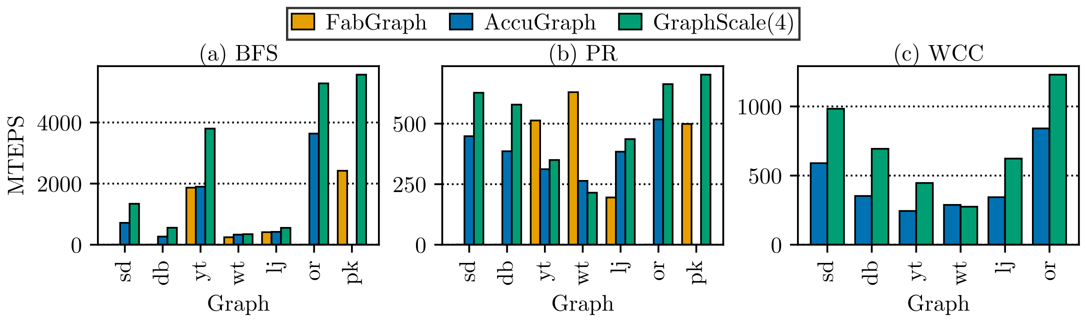


Fig. 5.11 Comparison of GraphScale with AccuGraph and FabGraph.

WCC are similar to the BFS measurements besides the even more pronounced super-linear scaling for roadnet-ca and top-cats.

For the comparison of GraphScale against AccuGraph, FabGraph, HitGraph, and ThunderGP, we use the performance measure millions of traversed edge per second (MTEPS) defined by the Graph500 benchmark as $|E|/t_{exec}$ with runtime t_{exec}. More is better for this performance metric. This is different than the MTEPS* definition $|E| \cdot i/t_{exec}$ with number of iterations i used by HitGraph and ThunderGP. MTEPS* eliminates number of iterations in favor of showing raw edge processing speed. However, faster convergence to results due to lower number of iterations has more impact on actual runtime than usually smaller differences in raw edge processing speed [DRF21a].

Figure 5.11 shows AccuGraph and FabGraph compared to GraphScale scaled to four memory channels. FabGraph was only measured for BFS and PR on the youtube, wiki-talk, live-journal, and pokec graphs and the AccuGraph measurements did not include the pokec graph. For the BFS measurements, we use 0 as the root vertex as was done for AccuGraph and FabGraph in their respective papers. Overall, we show an average performance improvement over AccuGraph of 1.48× and over FabGraph of 1.47×. Especially AccuGraph benefits from much higher clock frequency due to lower design complexity. For wiki-talk, GraphScale scales the worst (cf. Fig. 5.10) and is, thus, not able to provide a large improvement. FabGraph performs exceptionally well for PR on very sparse graphs (i. e., youtube and wiki-talk).

Figure 5.12 compares the four channel GraphScale system to HitGraph. Because HitGraph does not provide BFS performance numbers, the GraphScale BFS results in Fig. 5.12(a) are compared to single-source shortest paths results from HitGraph which has the same output for edge weights 1. We were not able

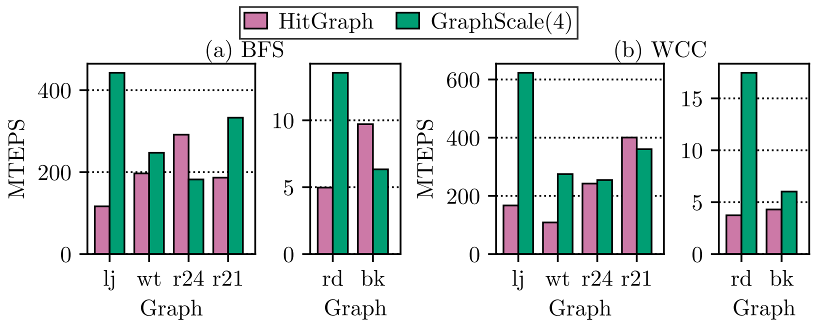


Fig. 5.12 Comparison of GraphScale and HitGraph.

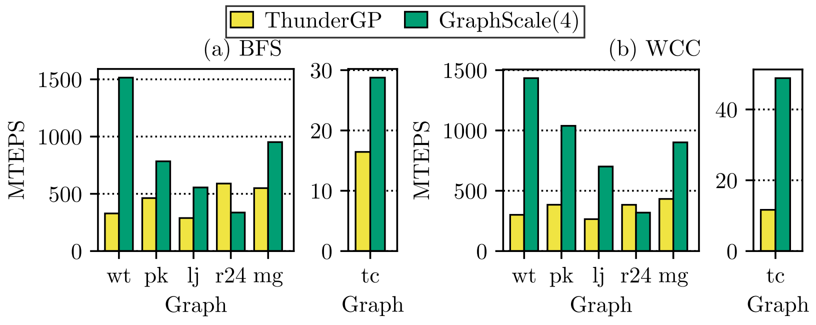


Fig. 5.13 Comparison of GraphScale and ThunderGP.

to obtain the root vertices that were used for the HitGraph measurements thus we measure with our own. Overall, we show an average performance improvement over HitGraph of 1.89× for BFS and 2.38× for WCC. As already shown in Fig. 5.10, GraphScale benefits from denser graphs like live-journal in contrast to a sparse graph like wiki-talk. We also again observe the superior scaling of our approach for the roadnet-ca graph. For graphs with a large vertex set like rmat-24-16, our approach requires increasingly more partitions (9 for rmat-24-16), introducing a lot of overhead.

Figure 5.13 compares the four channel GraphScale system to ThunderGP. For this experiment, we implemented a vertex range compression proposed by ThunderGP which removes any vertex without an outgoing edge from the graph before partitioning it. While we apply this for the purpose of comparing the approaches on equal footing, we criticise this compression technique because it returns wrong results. Taking BFS as an example, vertices that only have incoming edges also receive updates even though they do not propagate them. ThunderGP uses random root vertices generated with an unseeded random generator. Thus, we reproduce their root vertices and measure on the exact same input parameters. Overall, we achieve a speedup over ThunderGP of 2.05×

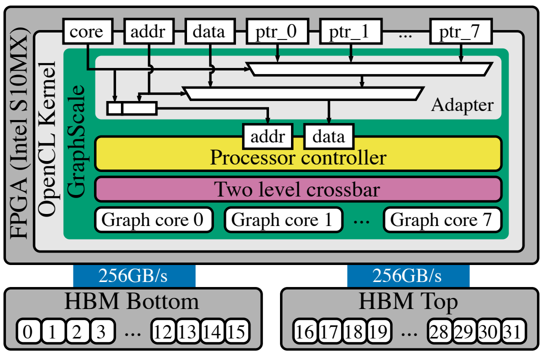


Fig. 5.14 GraphScale adapter for OpenCL.

and 2.87× for BFS and WCC respectively. The vertex range compression makes the wiki-talk graph much denser which our approach benefits from. The only slowdown we observe is again for the rmat-24-16 graph due to partition overhead.

5.3.3 HBM

Figure 5.14 shows the integration of GraphScale into an OpenCL kernel enabling usage with the S10MX development kit because Intel OPAE is not available for this system. The integration is handled by an adapter converting the function call interface of OpenCL kernels to the address and data based register interface of the processor controller. This works because GraphScale, which is added to the OpenCL kernel as a VHDL library function, retains its state between kernel function calls. The reset signal of GraphScale is only triggered when the OpenCL kernel is created. Additional to a core, address, and data parameter, the OpenCL kernel also has one pointer parameter for each memory channel. For HBM, pointer parameters are bound to a memory channel during compile time and have a different data type than the data parameter. The core parameter multiplexes the pointers and the address parameter multiplexes the data parameter and the pointer parameters which is then passed to the data port of the processor controller of GraphScale. The core and address parameters are additionally concatenated and passed to the address port.

Figure 5.15 shows the scalability of GraphScale up to 16 channels on the HBM-enabled system (hatched) compared against GraphScale scaled over up to four channels of DDR4 memory (solid color). We directly compare HBM configurations with double the number of graph cores to the DDR4 configurations because the HBM memory channels are half as wide as the DDR4 memory channels

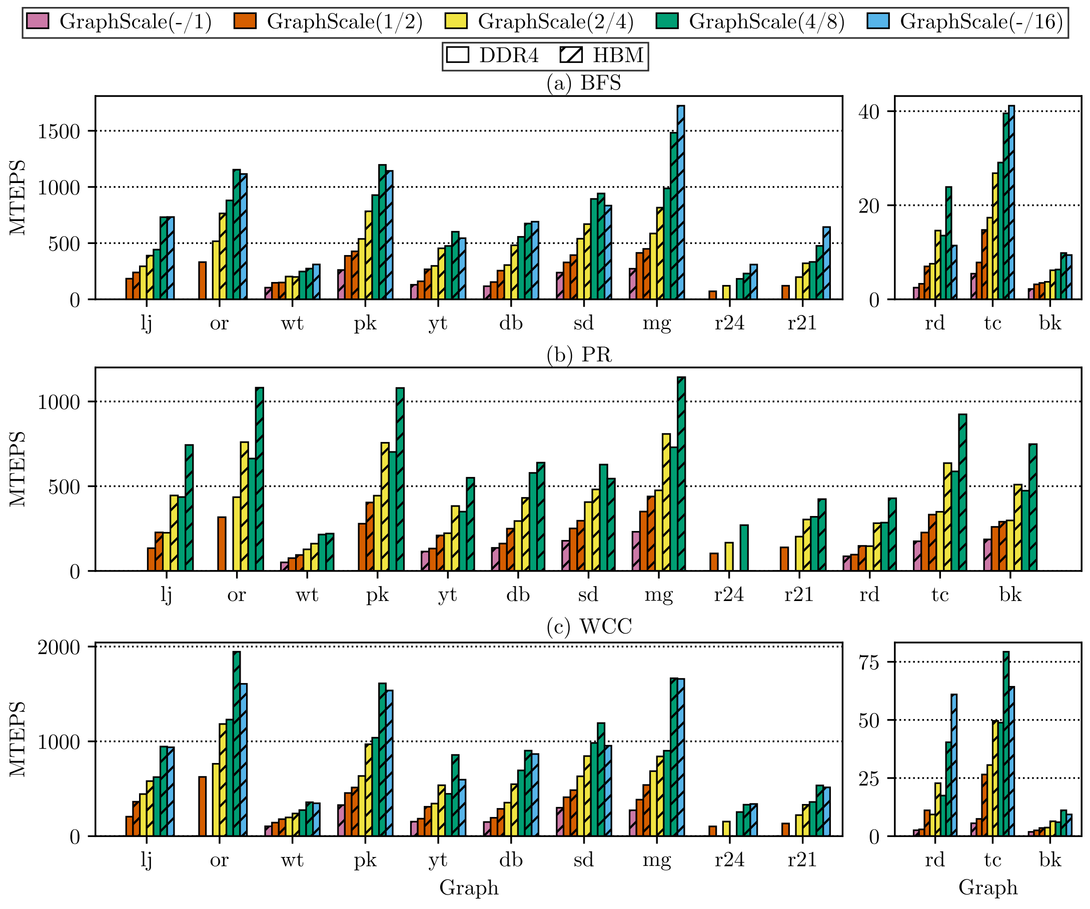


Fig. 5.15 GraphScale HBM memory channel scalability from one to eight channels.

resulting in half as many edges processed per core simultaneously. There are no DDR4 configurations matching the 1 core and 16 core HBM configurations. The narrower data paths of the HBM channels, however, also lead to significantly higher clock frequencies (cf. Tab. 5.3). Additionally, we overprovision the crossbar with 16 label scratch pad banks per graph core while the graph cores only do a maximum of 8 requests per clock cycle due to the narrower data path. If there are no measurements shown for certain graphs (e. g., no measurement for the HBM 1 graph core configuration for the live-journal graph), this means that the graph did not fit into the memory of this configuration. Each HBM channel only has 256MB of memory and different to DDR4, there is no datapath connecting all memory channels, limiting graph size.

To fit 16 graph cores, we had to remove the ready signal decoupling registers in the crossbar due to resource utilization constraints leading to much reduced clock frequency, halve the label scratch pad size, and remove the overprovisioned label scratch pad banks. For some graphs, the reduced clock frequency and

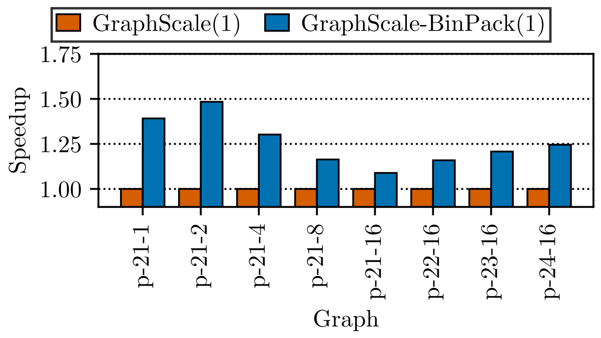


Fig. 5.16 Compression speedup on perfect graphs.

default number of label scratch pad banks means that GraphScale-HBM with 8 graph cores runs faster than GraphScale-HBM with 16 graph cores. Further scaling to more memory channels is limited by resource utilization of the FPGA. For scalability to 32 channels, each graph core of GraphScale would have to be smaller than 2% of the LUT resources. In principle, having a number of cores that is not a power of two is also possible but does not make sense here because 16 cores already is the limit. In the future, FPGAs with more resources and possibly smaller board support packages (providing basic functionality like memory and PCIe controllers) will enable scaling the system even further. When it eventually is possible to saturate every memory channel with a graph core, there are more methods to extract even higher performance from HBM [Hol+21]. The rmat-24-16 graph did not fit into memory at all for PR.

On average, GraphScale on HBM is able to provide a speedup of 1.53× with a maximum of 3.49× for WCC on the road-net graph. Compared to the theoretical HBM bandwidth of 512GB/s this may seem low. However, we are limited by resources such that we are not able to scale to the full 32 channels and do not achieve the best clock frequency for 16 channels due to the same reason. Additionally, for HBM we would have to reach an unrealistic 500 MHz clock frequency to utilize the full memory bandwidth compared to only 300 MHz required to reach the maximum bandwidth of a DDR4 channel.

5.3.4 Compression

Figure 5.16 shows the effect of the binary packing compression on the GraphScale performance on generated graphs for an increasing average degree and on the other hand increasing number of partitions. The experiments are run with all vertex labels set to -1 to prevent updates having to be written back to memory and the number of iterations over the graph is fixed to 1 to isolate the performance of loading the graph. The measurements show a performance improvement for GraphScale-BinPack(1) compared against the baseline of GraphScale(1) the lower the average degree of the graph is (cf. perfect-21-2) and the more partitions the graph has (cf. perfect-24-16). For the perfect-21-1 graph, we observe a drop in performance because there are only half as many vertex pipelines in our design as edge pipelines such that with an average degree of below 2 we are limited in the edge builder by the number of vertex pipelines. On average, we observe a speedup of 1.26× with the binary compression enabled and a maximum of 1.48× for the perfect-21-2 graph achieving our goal of reducing partition overhead for large graphs. However, for the real-world graphs that we use in our other experiments, we do not observe this improvement.

5.3.5 Understanding Data-dependent Performance Characteristics

To better understand the performance characteristics of GraphScale on different graphs, we propose a performance model based on two components. We model the theoretical maximum performance based on memory accesses alone (memory bound) and also simulate the theoretical maximum performance based on the crossbar (crossbar bound) which is mainly influenced by imbalances in accesses to the label scratch pad both locally and globally producing bubbles in the processing pipelines. As we observed in [DRF21a], the biggest influence on performance are number of iterations over the graph which we optimize for with asynchronous graph processing. Thus, we will normalize all performance numbers in this experiment to one iteration over the graph. Additionally, imbalance of number of edges between partitions can also cause lower performance which we will also exclude from the experiment by only looking at one graph core.

The memory bound M_{max} is mostly influenced by the average degree of the graph and the number of meta partitions as we observed for the compression too. It can be modeled as follows (i is the number of iterations over the graph and $d=1$ if the number of sub-intervals l is exactly one and $d=i$ otherwise):

$$M_{max} = n \cdot d + (2 \cdot n \cdot l + m) \cdot i$$

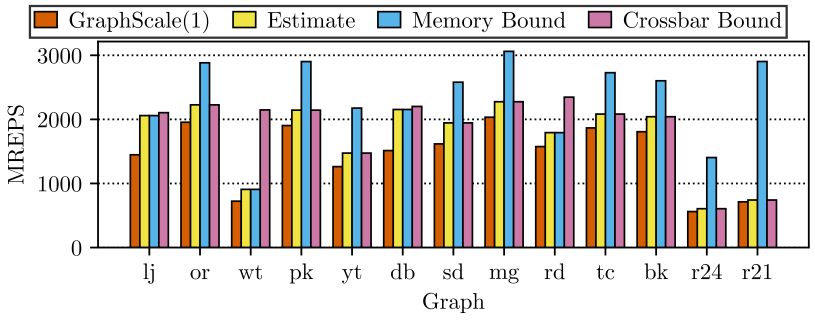


Fig. 5.17 Theoretical maximum performance.

We additionally simulate the crossbar bound C_{max} by going over the edge list in cycles and consuming an edge if the corresponding simulated label scratch pad bank has spaces left in an internal queue. The lookahead is limited to the FIFO depth from Fig. 5.4 regulated with a parameter in the simulation. We count the cycles that the simulation needs to consume all edges (a maximum of 16 per cycles with perfect balancing of requests) and calculate the theoretical maximum millions of read edges per second (MREPS) from that.

Figure 5.17 shows the result of our performance modeling compared against the actual performance measurements of GraphScale with one core. The final estimate is the minimum of the memory bound M_{max} and the crossbar bound C_{max}. We observe that we can closely model the actual performance with this estimate. For most real-world graphs, the performance is significantly limited by the crossbar bound because there are local imbalances in the label scratch pad accesses. This is especially prevalent for the synthetic rmat graphs. However, the graphs do not exhibit global imbalances of accesses. This suggests that optimizing the partitioning of the graphs taking our performance model as a measure of quality could improve performance and unlock the performance gains shown for the binary packing compression.

5.3.6 Discussion

Overall, we observe an average speedup of 1.48× over AccuGraph and FabGraph and 2.3× over HitGraph and ThunderGP with a maximum speedup of 4.77× for BFS on the wiki-talk graph over ThunderGP, confirming the potential of scaling asynchronous graph processing on compressed data. For optimizations, we show the importance of partition balance with stride mapping being very effective for graphs like slash-dot and youtube at the trade-off of shuffling the potentially beneficial natural vertex ordering of real world graphs. In our scalability measurements, we observe that GraphScale performance benefits from denser graphs in general (e. g., orkut and dblp graphs). This was especially pronounced compared against FabGraph for PR on very sparse graphs. Additionally, compared to HitGraph and ThunderGP, we observe a significant slowdown for graphs with a large vertex set, like rmat-24-16. This results in a trade-off in the compressed data structure between less data required for dense graphs and more partitioning overhead for graphs with a large vertex set. We tackle both these challenges (sparse and large graphs) with our binary packing decompressor with an average speedup of 1.26× on synthetic graphs. To explain the lacking performance improvement from binary packing compression on real-world graphs, we show a performance model forming a theoretical upper bound for GraphScale performance. Lastly, we scale GraphScale to HBM resulting in a 1.53× average speedup with a maximum of 3.49× speedup for WCC on the road-net graph. We are limited by resource utilization and clock frequency by the hardware available to us preventing us from utilizing the full HBM bandwidth. In this particular case, turning off compression could lead to increased performance because we are not limited by memory bandwidth.

5.4 Conclusion

We proposed GraphScale, an FPGA-based, scalable graph processing framework. GraphScale is inspired by the insight that current graph processing accelerators do not combine compressed graph representations, asynchronous graph processing, and scalability to multiple memory channels in one design (cf. Chapter 4). We showed the potential of such a design by defining a scalable two level crossbar and a two-dimensional graph partitioning scheme that enable all three accelerator properties in one system, addressing research question RQ2.2 *“How can the crucial graph processing accelerator properties be combined in one system?”*. Our experimental performance evaluation showed scalability and superior performance of GraphScale compared to state-of-the-art graph processing accelerators AccuGraph, FabGraph, HitGraph, and ThunderGP with an average speedup of 1.86× and up to 4.77× on dense graphs. Additionally, we answer research question RQ2.3 *“How can accelerators be scaled to unlock the potential of novel memory technologies like HBM?”*. To this end, we scale GraphScale to HBM resulting in a 1.53× average speedup with a maximum of 3.49× speedup for WCC on the road-net graph. We conclude that it is difficult to fully utilize the bandwidth provided by novel memory like HBM due to clock frequency and resource utilization constraints. To decrease partition overhead for large graphs, we implement binary packing compression which is able to improve the performance by another 1.26× speedup.

Based on our performance model, we conclude that it is possible to further improve performance by exploring partitioning schemes to improve partition balance (e. g., as in [BL18]) in future work. Additionally, we think that baking the workload into the bitstream limits the usefulness of graph processing accelerators. A remaining challenge is making these kinds of accelerators dynamic such that workloads can be switched with low latency.

Part II

Flexible Data Processing on FPGAs

Chapter 6

PipeJSON: Parsing JSON at Line Speed on FPGAs

In recent years, JavaScript Object Notation (JSON) [Bra+14] and its variants have gained popularity for both data exchange and storage. Their appeal lies in the flexible and semi-structured way they represent data (e. g., [Abi+18]). This makes them especially interesting for analytical data processing systems handling vast and diverse data sets (e. g., [JQZ20; DLN21; NJ03]). Although these systems utilize efficient internal binary representations of the data for processing and storage, ingesting raw JSON documents in the first place is expensive due to low parsing performance [Li+17b; LL19]. Thus, due to the relevance of semi-structured data, a number of JSON parsers have been developed (cf. Tab. 6.3), starting with traditional libraries such as RapidJSON [Yip+15] and sajson [Aus+12]. More recently, through advances in modern CPU technology, performance improvements have been made by new approaches like Mison [Li+17b] and more recently simdjson [LL19]. These parsers leverage vectorized AVX instructions for data parallel SIMD processing.

Nonetheless, we conjecture that CPU-based parsers are still constrained in parsing efficiency, particularly because of the rigid instruction set and limited pipelining capabilities of CPUs (cf. Fig. 6.1). These restrictions mean that they only achieve a fraction of the parsing performance, possible with pipelining on FPGAs (e. g., attached via PCIe (cf. PipeJSON (PCIe))), constraining parsing performance from reaching the practical limit of memory bandwidth. Addressing

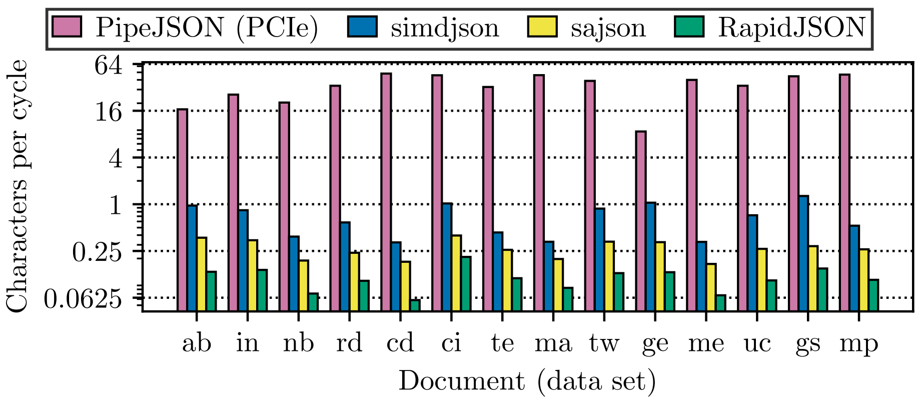


Fig. 6.1 Parallelism extracted by PipeJSON compared to CPU JSON parsers in characters per clock cycle (log-scale).

research question RQ3.1 *"How can line-speed data ingestion of arbitrary input data be achieved on FPGAs?"*, we present PipeJSON, the first standard-compliant JSON parser to process tens of gigabytes of data per second, by parsing multiple characters per clock cycle. PipeJSON validates structural correctness of documents during parsing and is flexible in the sense of not requiring parser generation, thus being able to handle arbitrary JSON documents during runtime. Instead of relying on data parallel AVX instructions of modern CPUs, PipeJSON leverages FPGA hardware to make extensive use of AVX-equivalent loop unrolling complemented by deep pipelining. Conceptually, PipeJSON is based on the data parallel simdjson tokenizer, that we extend in this chapter. We combine the tokenizer with multiple fully pipelined components on the FPGA[1]. To ensure easy integration into software projects, PipeJSON is implemented with Data Parallel C++ [Rei+21], accessible to software developers, and compatible with various programming languages, such as C++ and Go.

Our PipeJSON system achieves an average speed-up of 7.95× over state-of-the-art CPU-based JSON parsers, *including* data transfer to the FPGA and back via PCIe. We show that the flexible pipelining and data parallelism, achievable with FPGAs, is ideally suited for tasks like JSON parsing and is able to overcome CPU performance bottlenecks, such as number parsing. While, in this chapter, we neither consider task parallelization on CPUs nor on FPGAs, in the future, multiple CPU cores and unused FPGA resources could be used to parse multiple documents at once. Furthermore, while we have to use the FPGA as a CPU-attached offload accelerator in this work, due to hardware restrictions, it may

[1]We utilize pipeline parallelism instead of processing one block after the other as in [LL19]

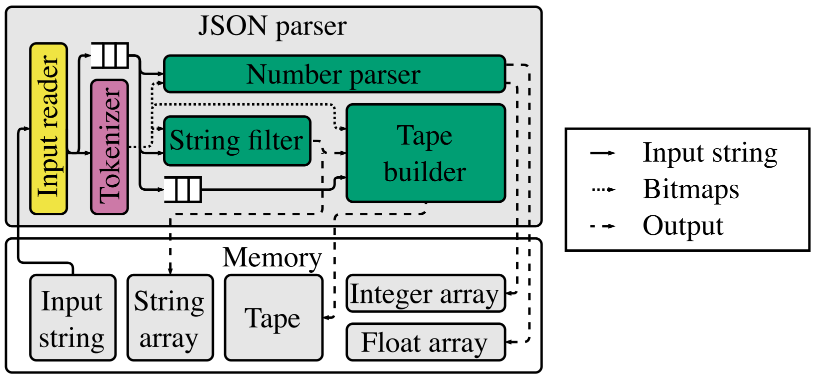


Fig. 6.2 PipeJSON architecture overview.

also be directly attached to the network to completely alleviate the CPU from parsing and possibly other tasks in a data analytics context.

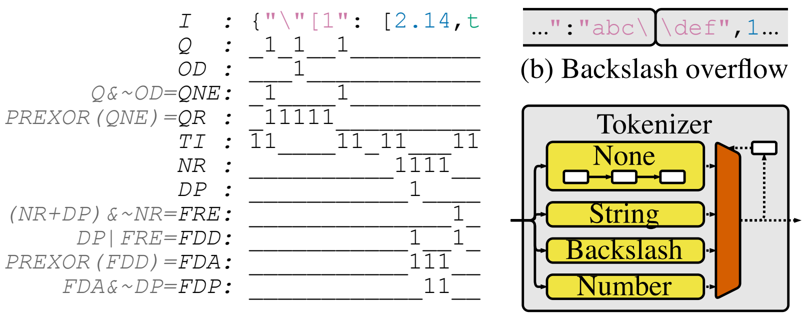
In this chapter, we first present the PipeJSON JSON parser concept with data parallelism matched to the width of the off-chip memory interface (Sect. 6.1). Additionally, we show how PipeJSON can be implemented as a prototype on an FPGA with a simdjson-compatible interface, as a drop-in replacement for CPU parsers (Sect. 6.1.5). In Sect. 6.2, we evaluate PipeJSON performance compared against state-of-the-art CPU parsers and along relevant JSON-specific document properties. We wrap this chapter up with a conclusion in Sect. 6.4.

Parts of this chapter have previously been published in the proceedings of DaMoN 2022 [Dan+22].

6.1 PipeJSON System Architecture

In this section, we provide an architecture overview of the PipeJSON concept which we completely designed with FPGAs in mind and define its most important components in more detail (i. e., tokenizer, number parser, tape builder). Additionally, we introduce the PipeJSON prototype used for performance measurements in the system context.

PipeJSON consists of the parser modules shown in Fig. 6.2. Each module denotes a pipeline on the FPGA, where each step in the pipeline processes different data in parallel. Additionally to building deep pipelines, we utilize the FPGA's flexibility by adding data parallelism where state-of-the-art CPU architectures cannot use SIMD to solve a problem in a data parallel fashion (e. g., string filtering and number parsing). A raw JSON document is read as an


```
                I   : {"\"[1": [2.14,t
                Q   : _1_1__1_________
                OD  : ___1____________
        Q&~OD=QNE: _1____1_________
  PREXOR(QNE)=QR  : _11111__________
                TI  : 11____11_11___11
                NR  : __________1111__
                DP  : ___________1____
 (NR+DP)&~NR=FRE: ______________1_
        DP|FRE=FDD: ___________1__1_
  PREXOR(FDD)=FDA: ___________111__
       FDA&~DP=FDP: ____________11__
```
(a) Bitmap examples

(b) Backslash overflow

(c) Pipelining overview

Fig. 6.3 Tokenizer.

input string from memory and output is written back in the form of a binary representation as a combination of a tape, with JSON tokens, and arrays for string, integer, and float values. While the output could be written in any format, we have chosen the particular tape representation from [LL19] to guarantee comparable results in our evaluation. The input reader reads the input string and splits it up into blocks of 64 characters, matching the cache line width of a single memory channel of 64B of current DRAM technology, before passing each block into the tokenizer and a FIFO queue (i. e., on-chip memory and logic) for later usage. The tokenizer annotates the characters in each block with bitmaps for further processing, which are then passed to data type-specific modules. Based on the bitmaps and the 64B data blocks from the FIFO, the string filter selects and writes all string characters to the string array and passes the number of strings in the block and their lengths to the tape builder. The tape builder reads the input characters, bitmaps, and information from the string filter, and writes corresponding tokens to the tape in memory. Lastly, the number parser reads from the FIFO and receives bitmaps from the tokenizer, then transforms numeric characters to integers or floats, and writes them to the corresponding arrays. Subsequently, we discuss all components in more detail.

6.1.1 Tokenizer

The tokenizer computes the bitmaps from a given raw JSON input string block. Figure 6.3(a) shows a subset of the bitmap computation done in the tokenizer. In this simplified example, `I` denotes the input string, `Q` indicates the quotes, `OD` the ends of odd sequences of backslashes (cf. [LL19]), `TI` the tokens, `NR` the number characters, and `DP` the decimal points.

To attain the main bitmap used for string filtering (i. e., quoted ranges `QR`), the escaped quotes are removed from `Q` with bitmap `OD` and a prefix exclusive-or is applied to the resulting `QNE`. The token bitmap `TI` is used by the tape builder to write tokens to the tape. For the number parser, we mainly use the `NR` bitmap and the `FDP` bitmap marking the decimal digits of floats. The latter is computed by first adding `DP` to the `NR` bitmap and only taking the non `NR` bits, resulting in the float ends `FPE`. To get `FDP`, we then again apply the prefix exclusive-or operator on the ranges delimited by `DP` and `FPE` (i. e., `FDD`) and finally remove the decimal point again from the resulting `FDA` bitmap.

Because the input string is split up into fixed sized blocks in the input reader, a JSON value like a string might be part of multiple blocks (cf. Fig. 6.3(b)). Since this changes the tokenization behavior, we introduce the following main overflow types: (i) string, (ii) backslash, (iii) number, (iv) none. If the last character of the current block was in a string, escaped by a backslash (as in the example), part of a number, or neither of those, we use the respective overflow type as the context for the next block. We implement this by appending one bit at the end of the bitmaps to capture overflows and by pre-pending the last bit and the overflow bit from the end of the previous block to pass this information. For simplicity, we re-use overflow types derived from the number overflow type for negative numbers, the decimal part of a number and the decimal part of a negative number (not shown).

Figure 6.3(c) gives an abstract view of the tokenizer module. The sub-modules are pipelined for better throughput, but have to wait until the overflow type of the previous block is known. Since the bitmap computation for a block would have to wait for the previous block to appear in the pipeline, this would degrade performance. To circumvent this, we pre-compute the bitmaps for all overflow types in parallel and only decide on which bitmaps to pass on based on the then known overflow type from the previous block with a multiplexer.

6.1.2 String Filter

Figure 6.4 shows the pipelined structure of the string filter module. As input, the string filter takes the quoted range (`QR`) and quoted range end (`QRE`) bitmaps and a block of characters. For brevity, we only show an 8-character string filter in Fig. 6.4. During processing, the string filter compacts all string characters of

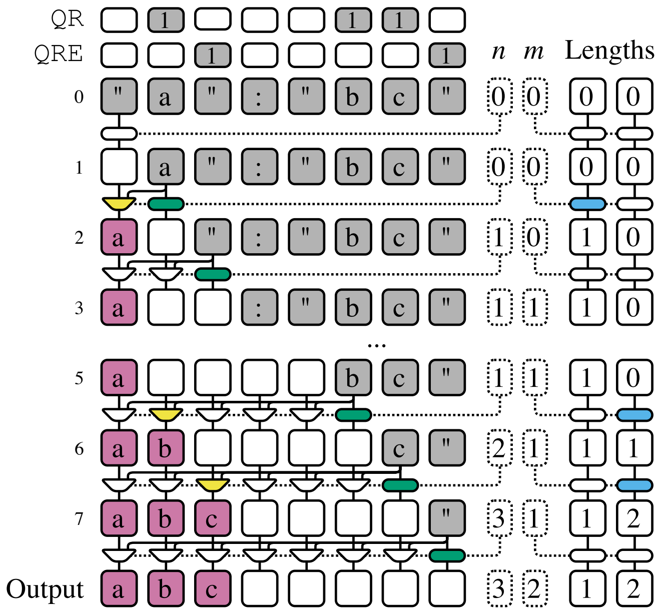


Fig. 6.4 String filter.

a block into one contiguous sequence and additionally keeps track of the number of total characters, number of different strings, and for each string its length. We achieve a throughput of one block of size b per clock cycle by having b pipeline steps each consuming one input character and extending the output sequence by one output character. For each step i, we have i multiplexers either forwarding the output character from step $i-1$ or forwarding the currently to-be-consumed character for multiplexer n if it is a string character. In case of a string character (steps 1, 5, and 6), we increment the current string length pointed to by m by one. Whenever we encounter a quoted range end (steps 2 and 7), we increment m by one. The results of the string filter are written to the string array and the number of strings in the block and their lengths is passed to the tape builder.

6.1.3 Number Parser

An important parser module is the two-staged number parser pipeline shown in Fig. 6.5. In the first stage, it takes the input characters and bitmap information, such as number ranges (`NR`), number range ends (`NRE`), and floating point decimal parts (`FDP`), as input and transforms them to integers. We build a pipeline that processes one input character of a 64 character block in each step and has an internal counter n to keep track of the current number that is parsed. When a number character is encountered (e. g., step 0), the temporary number currently

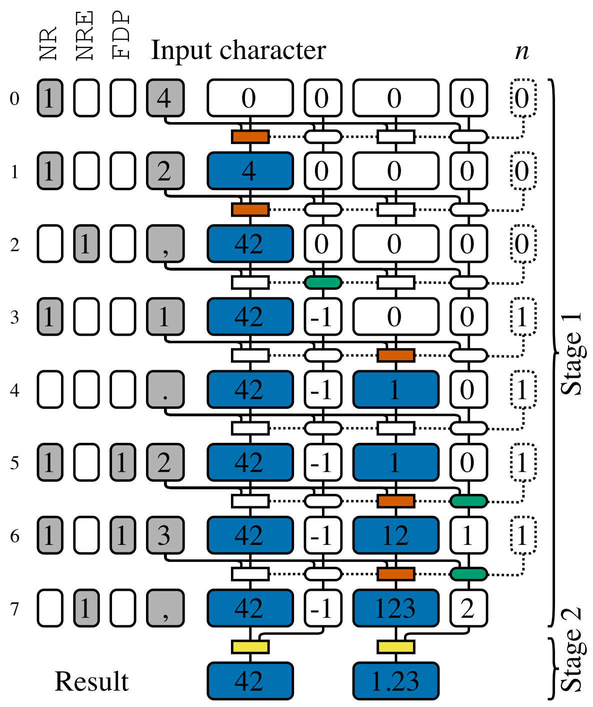


Fig. 6.5 Number parser.

pointed to by n is multiplied by 10 (i. e., shift) and added to the new number character. The resulting new temporary number is passed to the next pipeline step. If no number character is encountered, the old temporary value is passed on. If a number range end is encountered (steps 2 and 7), n is incremented because the parsing of the current number is finished. After a decimal point is encountered, an auxiliary decimal digit counter is incremented for each subsequent number character (steps 5 and 6). In PipeJSON, this pipeline is at least 75 steps deep to also account for possible number overflows where characters from the previous block are pre-pended. In a second stage, numbers with a decimal point are multiplied by 10^{-d} (i. e., shift) where d denotes the number of decimal digits to transform them into float values. The results are written to the integer and float arrays, respectively.

6.1.4 Building the Tape

For comparison reasons, we kept the binary representation generated by PipeJSON similar to the tape from simdjson [LL19], shown in Fig. 6.6. The tape contains tokens of one Byte and nested object and array structures are marked by a begin and end token. There are tokens for strings, integers, Boolean, null

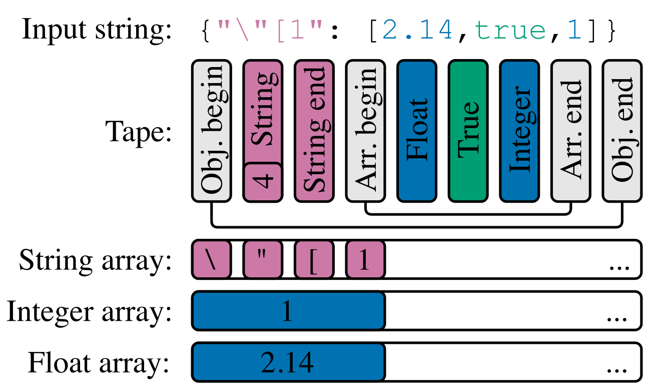


Fig. 6.6 Resulting binary data representation.

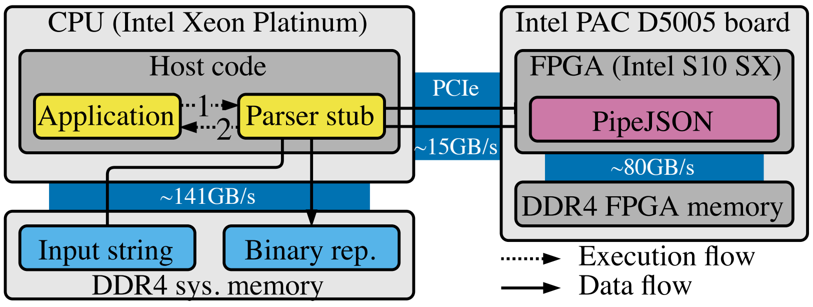


Fig. 6.7 Parser system architecture.

(not shown), and floats. The six least significant bits of a string token are used to store the length of the string. Long and overflowing strings are stored as a series of string tokens, each with a maximum length of 64, and a delimiting string end token. The characters of the string itself are stored in the string array, by omitting tokens for commas and colons because they can be derived from the context of being in an object or array.

6.1.5 Summary – PipeJSON Prototype

Figure 6.7 depicts the system architecture of our PipeJSON prototype. We implement PipeJSON in Intel OneAPI with Data Parallel C++ [Rei+21] as a parser stub on the host CPU that is used to communicate with the FPGA and PipeJSON itself on a PCIe-attached FPGA. In the PipeJSON (PCIe) configuration of the design that we also show in Fig. 6.1, an application directly calls the API of the parser stub, forwarding the input string. The parser stub triggers PipeJSON, which directly accesses the input string through PCIe and returns the resulting binary representation. The result can then be passed back to the calling application without any buffer copying or unnecessary detour through FPGA memory. This makes PipeJSON a drop-in replacement for existing CPU-based JSON parsers, which is PCIe bandwidth-bound. Hence, we also benchmark PipeJSON (D2D), where the input string is placed on the FPGA memory beforehand and the binary representation is written to FPGA memory only. This is used to show the potential of our parser but not as conveniently usable as the PipeJSON (PCIe) configuration.

6.2 Evaluation

In this section, we experimentally evaluate PipeJSON in the drop-in replacement (PCIe) and FPGA on-board, device-to-device (D2D) variants on different JSON data sets, compared to related JSON parsers. We discuss the results in the context of the theoretical maximum bandwidth of our design.

6.2.1 Setup

For the experiments, we use our prototype (cf. Fig. 6.7), integrated with a server equipped with two Intel Xeon Platinum 8260 CPUs at 2.40 GHz and 377 GB of DDR4-2933 memory. The server also features an Intel programmable acceleration card (PAC) D5005 board attached via PCIe v3 x16. The D5005 contains one Intel Stratix 10 SX 2800 FPGA with 933,120 adaptive logic modules as the basic logic building blocks, 3,732,480 registers, 30.5 MB of BRAM, and 5,760 DSP blocks. We use Intel OneAPI 2022.1 with the standard board support package and Quartus 19.2 for synthesis of the FPGA bitstream. The host code for PipeJSON is compiled with dpcpp 2022.1. For the measurements of the CPU parsers, we set up a container with Docker version 20.10.12 and compiled the parsers with gcc 9.1.0.

We use the same JSON parsers (cf. Tab. 6.3) and data sets (i. e., gsoc-2018 to github_events in Tab. 6.1) for our comparisons that [LL19] use, but add synthetic documents that we generated to explore effects of document structure on performance. For example, deep_objects contains nested JSON objects with depth 16 and big_arrays contains nested arrays with multiple entries. The documents ints_n, floats_n, strs_n are fixed at $m = 8\text{M}$, but vary by the number of digits n of integers and floats, respectively, and string length n. Table 6.1

Table 6.1 Documents used to benchmark JSON parsers by [LL19] and synthetic documents (for m is 8M).

Name	#Ints	#Floats	#Strings	#Objects	øDepth	#Arrays	øArray Size	Size
gsoc-2018 (gs)	0	0	34128	3793	2.67	0	0.00	3.33 MB
marine_ik (ma)	130225	114950	38268	9680	4.25	28377	8.64	2.98 MB
canada (cd)	46	111080	12	4	3.00	56045	2.98	2.25 MB
citm_catalog (ci)	14392	0	26604	10937	3.42	10451	1.14	1.73 MB
mesh.pretty (mp)	40613	32400	11	3	1.00	3610	21.22	1.58 MB
mesh (me)	40613	32400	11	3	1.00	3610	21.22	0.72 MB
twitter (tw)	2108	1	18099	1264	3.23	1050	0.54	0.63 MB
twitterescaped (te)	2108	1	18099	1264	3.23	1050	0.54	0.56 MB
update-center (uc)	0	0	27229	1896	3.25	1937	1.04	0.53 MB
random (rd)	5002	0	33005	4001	2.47	1001	4.00	0.51 MB
instruments (in)	4935	0	6889	1012	2.46	194	4.24	0.22 MB
nu MBers (nb)	0	10001	0	0	0.00	1	10001.00	0.15 MB
apache_builds (ab)	0	5289	26	884	1.99	3	293.33	0.13 MB
github_events (ge)	149	0	1891	180	2.56	19	2.53	0.07 MB
deep_objects_k (do$_k$)	$k \cdot 55903$	0	$k \cdot 83800$	$k \cdot 55903$	8.02	$k \cdot 27897$	2.00	k MB
big_arrays_k (ba$_k$)	$k \cdot 55904$	0	$k \cdot 70725$	$k \cdot 31516$	3.85	$k \cdot 3564$	8.10	k MB
ints_n	$m/(n+2)$	0	0	0	0.00	1	$m/(n+2)$	mB
floats_n	0	$m/(n+3)$	0	0	0.00	1	$m/(n+3)$	mB
strs_n	0	0	$m/(n+4)$	0	0.00	1	$m/(n+4)$	mB

Table 6.2 Resource utilization, clock frequency, and theoretical bandwidth by memory interface.

Configuration	LUTs	Registers	BRAM	DSPs	f	T_{max}
PipeJSON (PCIe)	72.2%	34.6%	8.1%	8.6%	242 MHz	13.00 GB/s
PipeJSON (D2D)	71.1%	34.8%	7.9%	8.9%	302 MHz	19.33 GB/s

LUTs: Look-up-tables

shows all of the documents we use in the evaluation, their abbreviation used in plots, and their properties.

6.2.2 Theoretical Performance and Resources

We consider the maximum theoretical performance for the throughput T_{max} of our design as:

$$T_{max} = \min\left(\frac{CL_{size}}{II} \cdot f, DTR\right),$$

with JSON documents of size larger than cache line size CL_{size}, pipeline initialization interval II (i. e., input rate of pipeline in cycles), achieved clock frequency of the design f, and data transfer rate from buffer location to FPGA (PCIe or FPGA memory) DTR. Overall, our design achieves an II of 1, which is the smallest possible value. This means that a new input can be passed to all pipelines in each clock cycle. Additionally, we fixed CL_{size} at 64 B which is the interface width for PCIe or a single channel of DDR4.

Table 6.2 shows our two PipeJSON configurations with their respective resource utilization, clock frequency f, and resulting maximum theoretical performance T_{max}. Even though the PCIe v3 x16 of our benchmark system has a theoretical bandwidth limit of 16 GB/s, we measured a maximum throughput of 13 GB/s with Intel OneAPI. Thus, we use that as the DTR for PipeJSON (PCIe) in turn limiting T_{max}, making PipeJSON (PCIe) I/O-bound. PipeJSON (D2D) has a maximum theoretical performance of 19.33 GB/s (i. e., compute-bound). To achieve higher performance we would either have to increase the number of interface channels (i. e., data parallelism), or clock frequency up to a theoretical maximum of 400 MHz for our Intel OneAPI setup on Stratix 10.

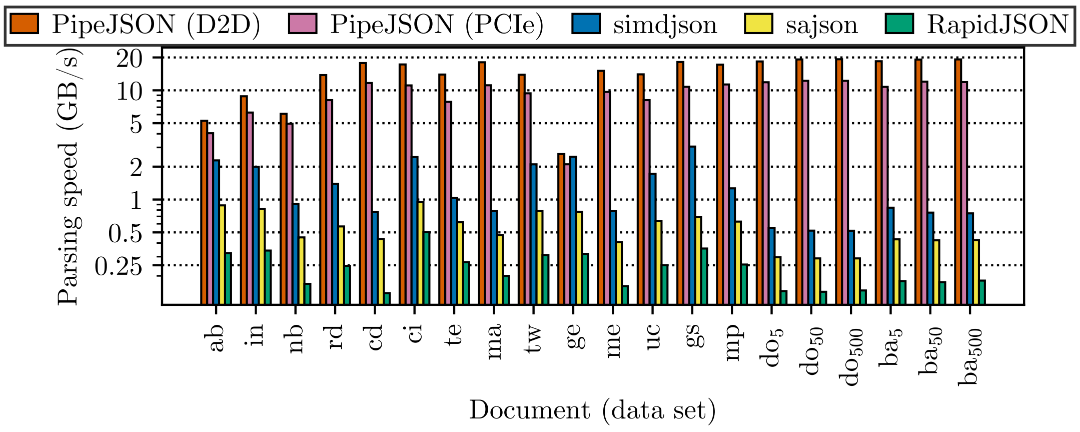


Fig. 6.8 Comparison of PipeJSON against CPU parsers (GB/s).

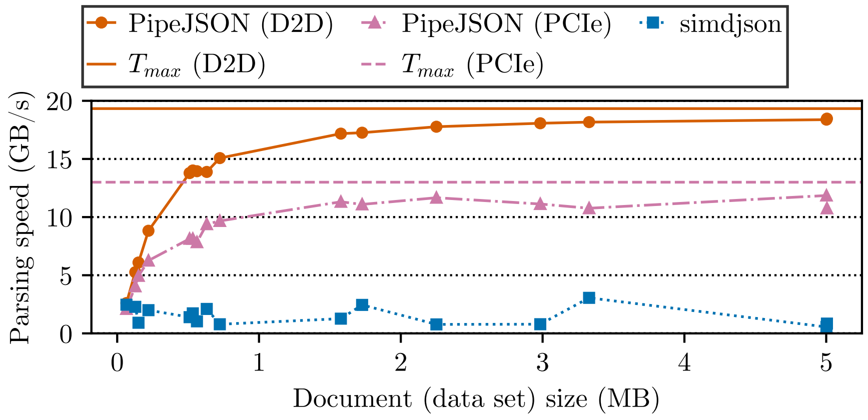


Fig. 6.9 Parsing speed by document size.

6.2.3 Experiments

In a first experiment, shown in Fig. 6.8, we compare our PipeJSON parser to the state-of-the-art CPU parsers simdjson, sajson, and RapidJSON (cf. Tab. 6.3) on all data sets from [LL19] (cf. gsoc-2018 to github_events), deep_objects_*k*, and big_arrays_*k*. The results show that while sajson performs better than the frequently used RapidJSON, the data parallel implementation of simdjson is even faster. Overall, PipeJSON outperforms the CPU parsers with a parsing speed of up to 12.22 GB/s for our PCIe-attached and 19.26 GB/s for our D2D, on-board variant. However, on the tiny 0.07 MB github-events document, simdjson is slightly faster than our PipeJSON (PCIe) variant. We observe the biggest improvements over CPU parsers for the marine_ik, deep_objects_*k*, and big_arrays_*k* documents which are the documents with the by far highest number of integers and floats.

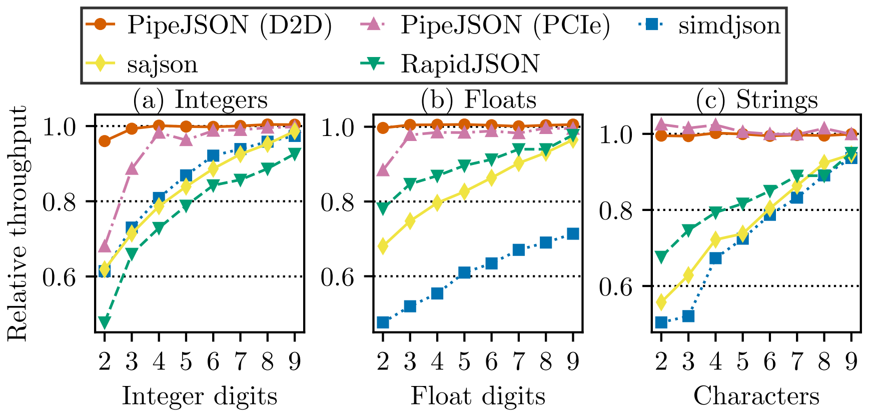


Fig. 6.10 Throughput relative to ten digit measurements.

To gain a better insight into the correlation between document size and performance, we sort the documents (excluding the 50 MB and 500 MB variants of deep_objects_k and big_arrays_k by their size in MB ascending in Fig. 6.9 and compare PipeJSON (D2D) and PipeJSON (PCIe) to the fastest CPU parser (i. e., simdjson). We added the theoretical performance limits from Tab. 6.2 as horizontal lines, to show by how much PipeJSON deviates. We can see that the PipeJSON performance logarithmically approaches the theoretical performance limit with document size for both configurations and we significantly lose performance below 0.5 MB document size. This stems from an estimated latency, introduced by Intel OneAPI, of calling PipeJSON of 25.29μs for PipeJSON (D2D) and 28.47μs for PipeJSON (PCIe) and an estimated pipeline depth of 900. Regarding pipeline depth, this means that PipeJSON does not operate at peak parallelism for the first and last 900 blocks of input data.

In a second experiment, we further explore the relatively low CPU parser performance (cf. Fig. 6.8) for marine_ik, deep_objects_k, and big_arrays_k documents. To this end, we plot the slowdown we observe for all parsers and increasingly smaller elements of integers, floats, and strings in Fig. 6.10. We measure performance on the synthetic documents ints_n, floats_n, and strs_n for two to ten digits – a maximum of ten digits fit into 32 bit integers – and normalize the measurements to the ten digit measurement of its respective parser. For PipeJSON, the results suggest a robustness regarding element sizes, while the CPU parsers are vulnerable to smaller element sizes of integers, floats, and strings. Notably, simdjson is slightly better for integers, but slower for float and character parsing. PipeJSON (D2D) is robust in performance, even for very

Table 6.3 CPU and FPGA JSON parsers.

Identifier	Platform	Validating	Data parallel
sajson [Aus+12]	CPU	👍	👎
RapidJSON [Yip+15]	CPU	👍	👎
Mison [Li+17b]	CPU	👎	👍
simdjson [LL19]	CPU	👍	👍
PipeJSON	FPGA	🖒	👍

Supported 👍: yes, 🖒: partially, 👎: no

small integers, but PipeJSON (PCIe) slows down slightly for smaller integers and floating point numbers. For this particular combination, PipeJSON has to write back more data to memory than is read from the input string in the first place, degrading performance with the limited PCIe bandwidth.

6.2.4 Discussion

In summary, from the analysis of PipeJSON and three state-of-the-art CPU-based JSON parsers we gained several insights about their performance on various data sets. The experiments on common JSON data sets, also used in [LL19], show superior results for data parallel parsing compared to conventional approaches. In particular, PipeJSON (D2D) achieved an average speedup of 12.56× over simdjson, the fastest data-parallel CPU parser, with a maximum speedup of 33.37×, and PipeJSON (PCIe) 7.95× and 21.17×, respectively. We found that PipeJSON performed especially well for larger document sizes, bound by I/O for PCIe and compute-bound for D2D, which could be circumvented through an increased frequency or PipeJSON instance parallelization on the FPGA. For tiny documents, however, PipeJSON cannot reach its peak performance for the first and last $900 \cdot 64$ characters, due to latency and an estimated pipeline depth of 900 clock cycles. When looking into the robustness of the parsers regarding digit scaling of integers, floats, and strings, PipeJSON is more robust than CPU parsers, especially compared to simdjson on floats and strings.

6.3 Related Work

Due to the relevance of semi-structured data, many JSON parsers like RapidJSON [Yip+15], or sajson [Aus+12] have been proposed. Through advances in modern CPUs, these conventional parsers have been improved by new approaches like Mison [Li+17b] and more recently simdjson [LL19], which leverage AVX instructions for data parallel SIMD processing. Additionally, there have also been efforts to take pressure off the parser by filtering the raw JSON data [Pal+18].

To overcome CPU limitations, Peltenburg et al. [Pel+21] propose a parser generator for JSON-template-specific accelerators, which denotes the only known related work on FPGAs[2]. However, their approach scales in the number of JSON document object model (DOM) nodes and requires a new generation for any structural change in the JSON document which is impractical for semi-structured data with flexible schema. However, prior to JSON, XML has been widely adopted, which resulted in several FPGA-based XML parsing accelerators (e. g., [CHL08; DNZ10; Sid13; Hua+14]). While early work focuses on accelerating recurring idioms of XML parsing to achieve 1 Byte per cycle performance [DNZ10], later approaches propose tree automata for XML schema validation [Hua+14].

PipeJSON Table 6.3 lists the discussed JSON parsers, compared to PipeJSON. While [Pel+21] and PipeJSON are both built for FPGAs, PipeJSON can parse any document without a separate parser generation step, thus being flexibly applicable to arbitrary JSON documents and having a constant resource utilization independent of document templates. We designed PipeJSON on data parallel concepts similar to the currently fastest CPU parsers, i. e., Mison, simdjson (cf. Fig. 6.1), as the first general-purpose, validating (i. e., structural correctness) FPGA-based JSON parser to combine line-rate performance with flexibility in the sense of not requiring parser generation.

6.4 Conclusion

This chapter addresses the important topic of JSON parsing for a growing number of applications and data processing systems that process and store semi-structured data. We propose a highly pipelined, data parallel JSON parser design on FPGAs that outperforms the fastest CPU-based JSON parsers by 12.56× with data already on the FPGA and 7.95× for our drop-in parser including data movement, connected via PCIe, only limited by the PCIe v3 bandwidth of our

[2]We do not consider stream filtering or JSON field extraction on FPGAs (e. g., Fleet [THZ20]) as parsing

setup. Thus, PipeJSON addresses RQ3.1 *“How can line-speed data ingestion of arbitrary input data be achieved on FPGAs?”*. Even though we are limited by the hardware platform available to us, PipeJSON could fully saturate current network bandwidths and effectively reduces data movement in the system when placed as a SmartNIC in the system (cf. Sect. 3.3.2). Additionally, PipeJSON shows that for some workloads, HLS languages (e. g., OneAPI) simplify integration of FPGAs into the system context especially from a software engineer’s perspective.

For future work, we propose extending PipeJSON to multiple memory channels, which is expected to improve the parsing speed by several factors, investigating optimizations for improved performance on smaller documents, exploring more complex output formats, and employing operator push down for filtering during parsing [Pal+18] for more efficient bandwidth utilization.

CHAPTER 7

GraphMatch: Subgraph Query Processing on FPGAs

Subgraph query processing, used e. g., for graph pattern matching, is an important workload in many application areas [Sah+20], like social network analysis [Sni+06] and protein interaction network analysis [PCJ06], where all embeddings identical to a given query graph are found in a data graph.

Subgraph query processing was mainly approached with two kinds of algorithms in related work on CPUs, namely backtracking [Ull76; Bi+16] and join-based approaches [Abe+17; MS19]. A recent study by Sun et al. [SL20] has shown that backtracking is efficient for large and sparse query graphs, and join approaches for smaller, dense query graphs. We focus on join-based subgraph

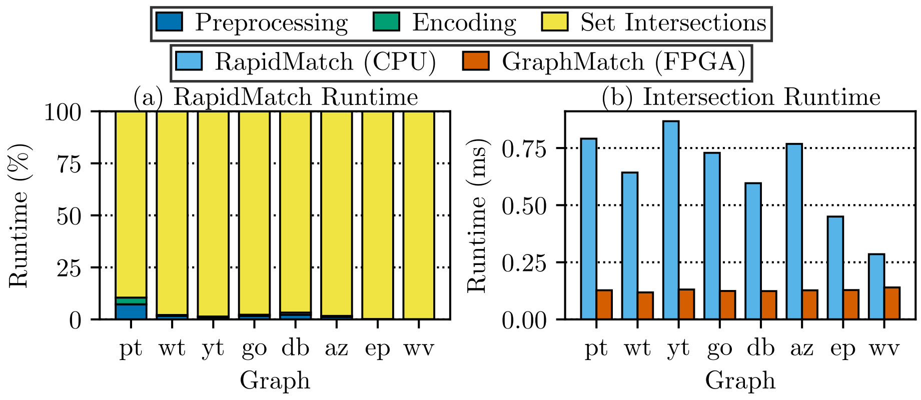


Fig. 7.1 RapidMatch (CPU) runtime, and RapidMatch and GraphMatch (FPGA) intersection runtime.

query processing on FPGAs using worst-case optimal joins (WCOJs) because recent work on WCOJ-based subgraph query processing on CPUs [MS19] was very promising. For instance, one of the most advanced CPU-based systems, RapidMatch [Sun+20a], is mainly based on joins, while considering graph structural information for candidate set filtering as in backtracking. When looking into join-based approaches like RapidMatch, set intersection is the most expensive operation despite vectorized processing, as shown in Fig. 7.1(a). This is due to challenges like limited parallelism, expensive round trips to main memory, and cache pollution in current general-purpose CPU hardware.

FPGAs showed that they can solve these challenges with massive, unstructured data and pipeline parallelism and flexible data movement through configurable data flow architectures (e. g., in relational join processing [Las+22]). For subgraph query processing, data movement could be more efficient by keeping partial matchings mostly on the FPGA chip. The benefits of FPGAs are shown in Fig. 7.1(b), which denotes a comparison of RapidMatch using SIMD-based set intersection (cf. [HZY18]) and an FPGA-based implementation that will be explained in more detail subsequently (cf. Sect. 7.1).

Thus, in this chapter, we address research question RQ3.3 *"How can flexible subgraph query processing benefit from FPGAs?"*. We propose GraphMatch, a flexible, WCOJ-based accelerator for subgraph query processing fully implemented on an FPGA. We first conceptually adapt the widely-used LeapFrog algorithm [Vel12] to FPGAs, before specifying the novel, FPGA-native AllCompare algorithm, which leverages the FPGA's massive pipelining. With this, we address research question RQ3.2 *"How can algorithmic software approaches be useful to hardware design?"*. GraphMatch itself is designed for (i) dense, small query graphs, (ii) supporting subgraph homomorphism and isomorphism, (iii) parallel, efficient set intersections, and (iv) directed and undirected graphs. For that, GraphMatch implements a number of set intersection operators with configurable number of input sets which are connected with partial matching multiplexers and demultiplexers able to dynamically switch matchings to a memory sink. The GraphMatch query parser can generate query plans for arbitrary subgraph queries on-the-fly and flexibly switch queries in a matter of cycles. With these query plans we select how we chain together and execute the operators.

GraphFlow [MS19] and RapidMatch are two CPU-based subgraph query processing systems that represent the state-of-the-art. GraphMatch shows promising scalability with an average speedup of 2.68× over GraphFlow and 5.16× over RapidMatch with a maximum speedup of over 100×. Overall, we conjecture that FPGAs are well suited to solve the set intersection bottleneck of CPU-based subgraph query processing systems. However, we still see areas of improvements for instance work balancing and for highly degree-skewed graphs.

We start this chapter by introducing the different set intersection approaches for CPUs and FPGAs on a spectrum from a software engineer's perspective to a hardware engineer's perspective in Sect. 7.1. In particular, we introduce LeapFrog variants for FPGAs implemented with OneAPI and VHDL, propose the AllCompare set intersector and compare these approaches comprehensively. Thereafter, we introduce the GraphMatch system in Sect. 7.2 with its important components and optimizations and show how GraphMatch does flexible query processing. In Sect. 7.3, we evaluate GraphMatch scalability and optimizations and compare it to GraphFlow and RapidMatch performance. We discuss related work on subgraph query processing in Sect. 7.4 before we summarize and discuss the results of this chapter with a conclusion (Sect. 7.5).

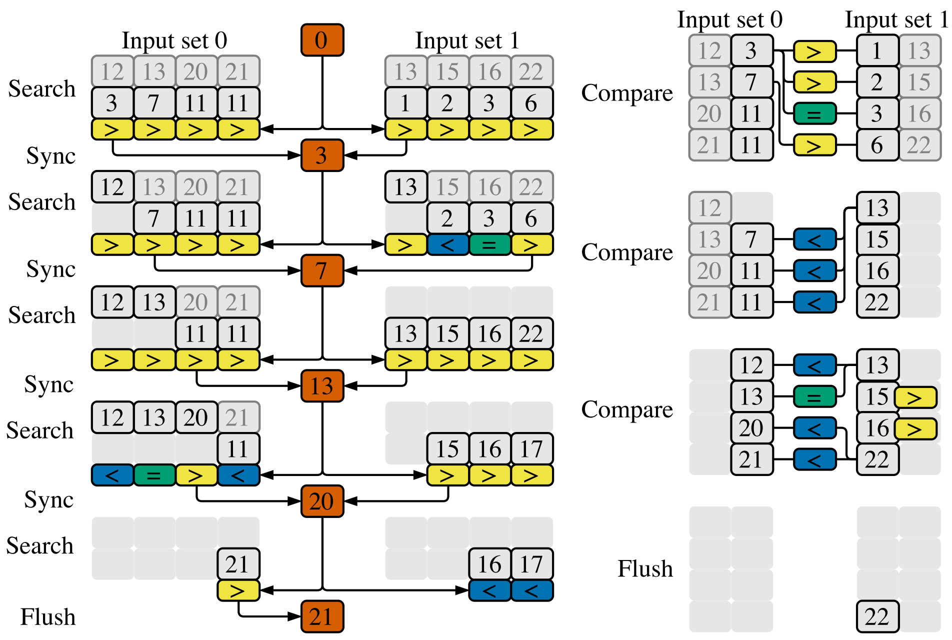
Parts of this chapter have been submitted as a conference paper to SIGMOD 2024.

7.1 Set Intersections on FPGAs

In this section, we focus on designing a parallel, efficient set intersector for FPGAs motivated by the observation that set intersections are the most expensive operation of subgraph query processing systems on CPUs (cf. Fig. 7.1(a)). We describe the different set intersection approaches for CPUs and FPGAs on a spectrum from a software engineer's perspective to a hardware engineer's perspective. Thereafter, we introduce the novel AllCompare set intersection approach that is highly optimized for FPGAs. Lastly, we show how the different set intersection approaches compare to each other and characterize the performance dimensions of the AllCompare set intersector.

7.1.1 Set Intersector Approaches for FPGAs

Figure 7.2 compares LeapFrog as the dominant approach for CPUs to the novel AllCompare set intersection approach specifically designed for FPGAs.



(a) LeapFrog set intersection (b) AllCompare set intersection

Fig. 7.2 LeapFrog and AllCompare set intersection approaches.

LeapFrog Set Intersection LeapFrog processes set intersections in turns of searching for a new search item and syncing the search item (Fig. 7.2(a) shows an example). The execution starts with 0 as the search item (orange box in the middle). In each search step, the current search item is compared against each element in each input set (two in this example). For the sync step, the next biggest element of each input set is communicated and compared to form the next search item and all elements that are smaller than the previous search item are discarded. This process is repeated until the search item is bigger than all remaining elements of one of the input sets. The rest of the set elements are then flushed. In each search and sync loop, LeapFrog has a guaranteed progress of only one element per set while the actual progress may be higher for real world sets. We implement a parallelized version of LeapFrog set intersection in OneAPI that is able to perform all comparisons in the search step in one clock cycle and implement a VHDL version that does the same parallelization and additionally does input set prefetching which is not easily implementable in OneAPI.

AllCompare Set Intersection With the observation in mind that on an FPGA we can implement many comparison operators in parallel, we propose the novel AllCompare set intersection approach for FPGAs. Figure 7.2(b) shows

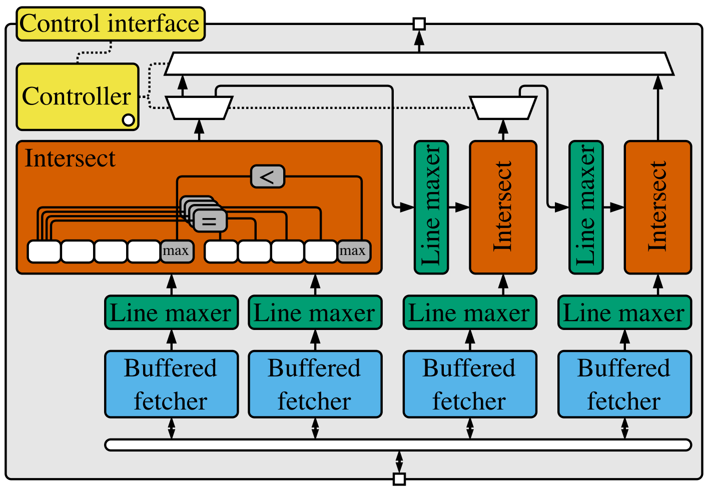


Fig. 7.3 AllCompare set intersector architecture.

how AllCompare is able to massively reduce the number of steps for the same set intersection that is shown for LeapFrog. In each compare step, conceptually AllCompare compares all elements of both input sets against each other. Elements that are smaller than an element in the other set are discarded, elements that have an equal element in the other sets are put out and discarded and all other elements remain. In the end, the last remaining elements are flushed. AllCompare guarantees progress of at least one full line of one of the input sets and is thus able to process the same set intersection in under half of the clock cycles that are required by the LeapFrog set intersection approach in this example.

7.1.2 AllCompare Set Intersector Architecture

Figure 7.3 shows the AllCompare set intersector architecture for four input sets that implements the concept presented in Fig. 7.2. Four buffered fetchers read the input data into the intersector and feed the input lines (4 elements per line in this example; 16 elements per line on the actual FPGA hardware) through line maxers. The line maxers find the maximum of the line which may not be the last element because not all elements of a line have to be valid (e. g., set is smaller than line width). These lines with extracted maximums are fed into the intersect operators. In each intersect operator, in each cycle, all elements are equal compared against all elements from the other input set (connections only shown for one element in this example) to determine which elements should be put out as the result of the set intersection. Additionally, the previously extracted

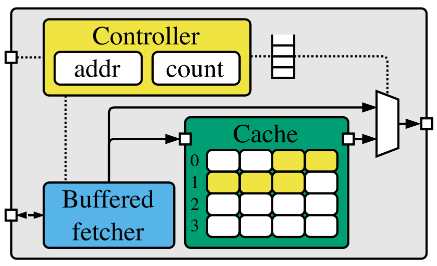


Fig. 7.4 Cached fetcher architecture.

maximums of both lines are compared and the line with the smaller maximum is completely discarded as all elements are smaller than at least one element of the other line. The results are fed into a demultiplexer which either forwards the output to the next line maxer and intersect operator or the output port. As the last component, each AllCompare set intersector contains a controller that is connected to the control interface. The user may define the switches that switch the demultiplexers and the multiplexer before the execution. This allows dynamic reconfiguration of number of input sets during runtime.

Especially during subgraph query processing, oftentimes the same vertex neighborhoods are accessed as input sets of the set intersectors repeatedly. Thus, we propose a cached fetcher architecture that stores the most recently accessed input set in a cache and serves subsequent requests to the same input set from the cache. Figure 7.4 shows the cached fetcher architecture. The controller receives the requests and stores the last address and number of elements accessed in a register. If the current request is equal to the previous one, it is flagged as cached and directly inserted into a FIFO queue which multiplexes the output port. If the current request cannot be served through the cache, it is forwarded to a buffered fetcher which sends the request to memory, flagged as fetched, and inserted into the FIFO queue, too. When the request is eventually served by memory, the data is written into the cache (implemented as BRAM) as whole lines of memory starting from cache address 0. The FIFO queue is continuously observed. If a new request arrives, the cached flag is read out and either the data from the buffered fetcher is directly forwarded or the respective number of memory lines are read from the cache again starting at address 0. The cached fetcher has the same interface as the buffered fetcher and may thus directly replace the buffered fetchers in the AllCompare set intersector architecture.

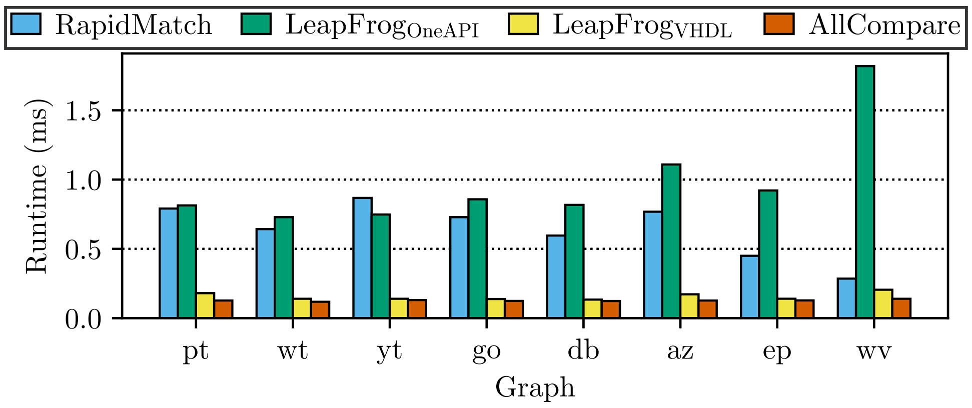


Fig. 7.5 CPU (RapidMatch) vs. FPGA intersection operators.

7.1.3 Set Intersector Performance Characteristics

In this section, we first compare the four different set intersection approaches: CPU-based, vectorized RapidMatch set intersection, LeapFrog implemented in OneAPI, LeapFrog implemented in VHDL, and AllCompare set intersection. Thereafter, we discuss the performance of AllCompare set intersection in detail for characteristics such as input set size, output set size, and number of input sets and degree of caching with the cached fetcher.

The benefits of adapting algorithms to FPGAs Figure 7.5 shows the comparison of the CPU-based RapidMatch intersection function against the different FPGA-based implementations of intersectors that we introduced.

The benchmark environment as well as the graphs used are described in Sect. 7.3.1. For $\text{LeapFrog}_{\text{OneAPI}}$, a new Agilex FPGA is used which, however, was not enabled by Intel for the VHDL-based design flow during the work on this chapter. The benchmark shows the runtime in milliseconds of 5000 set intersections of neighborhoods of random vertices of the respective graphs.

Overall, we observe that $\text{LeapFrog}_{\text{OneAPI}}$ performs similar to the RapidMatch set intersection function for some graphs and worse for others. The RapidMatch set intersection function performance benefits from very small graphs that fit mostly into the cache hierarchy of the CPU for the epinions (ep) and wiki-vote (wv) graphs. The performance of $\text{LeapFrog}_{\text{OneAPI}}$ in turn is heavily influenced by average degree of the graph, thus, the performance for the amazon (az) and wiki-vote (wv) graphs are noticeably worse than for the other graphs. Additionally, the VHDL FPGA implementations $\text{LeapFrog}_{\text{VHDL}}$ and AllCompare (i. e., specifically

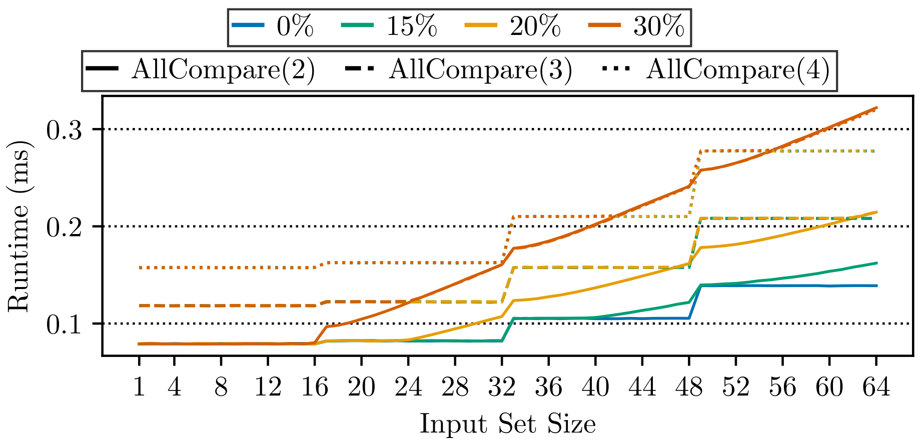


Fig. 7.6 Runtime of set intersection with AllCompare for two to four input sets over input set size and output size as percentage of input size.

tailored to FPGAs) perform significantly better than the CPU and OneAPI implementations. However, AllCompare always outperforms $\text{LeapFrog}_{\text{VHDL}}$. Thus, we can see the progression from adopting a CPU algorithm with $\text{LeapFrog}_{\text{OneAPI}}$ in a software engineer-friendly HLS environment over a VHDL implementation of the CPU algorithm ($\text{LeapFrog}_{\text{VHDL}}$) to a highly optimized FPGA implementation in AllCompare. In the remainder of this paper, we proceed with the tailor-made AllCompare set intersector.

Characteristics of the tailor-made FPGA intersector Figure 7.6 shows the runtime of AllCompare in milliseconds with two, three, and four input sets on 5000 set intersections with varying input set and output set size. AllCompare on two input sets where 0% of the input set are part of the output set (blue solid line) forms a memory-bound baseline. This baseline is not influenced by fine-granular input set size changes but by the number of memory lines it has to fetch. For this implementation, 16 elements of an input set fit into one line. Thus, the runtime increases each time the last element is part of a new line. For the measurements where 15%, 20%, and 30% of the input sets are in the output sets, the runtime is additionally bounded by the output set size because AllCompare only puts out one element per clock cycle. More is not required by GraphMatch. For three and four input sets, the runtime is increasingly more bound by memory because more data has to be fetched from memory per set intersection. For four input sets, only the measurement where 30% of the input set are part of the output set (orange dotted line) is bound by output size.

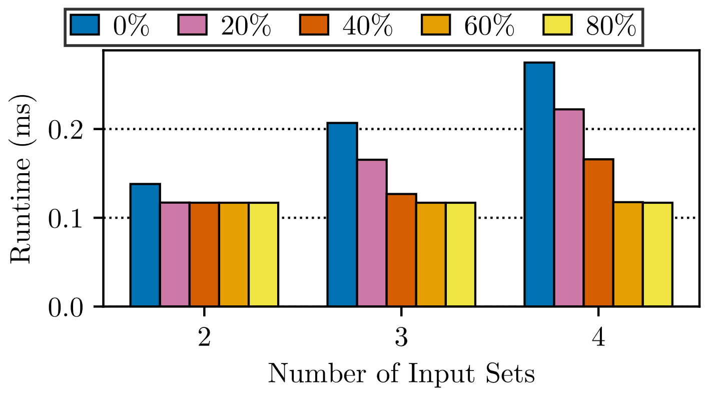


Fig. 7.7 Input set caching for access percentage cached by number of input sets.

Figure 7.7 shows how input set caching (cf. Fig. 7.4) influences set intersection performance for the AllCompare set intersector for different number of input sets. Again, we measure the runtime in milliseconds over 5000 set intersections. Each set intersection intersects sets of size 64 without overlap such that output size does not influence performance. Additionally, we vary cache hit rate from 0% to 80% in steps of 20%. Overall, we observe that at the latest of 80% cache hit rate, the performance reaches the same baseline for each number of input sets denoting a compute-bound baseline set by the cycles the FPGA logic needs to perform the intersection itself. For larger numbers of input sets this baseline is reached later with higher cache hit rate. This is because more data has to be fetched from memory in the first place but the number of comparison steps stays the same. It is important to note that AllCompare has the same runtime irregardless of number of input sets if memory requests do not play a role.

7.1.4 Discussion

This section provided a detailed look at the hardware design process in parts from the perspective of a software engineer to understand where FPGAs can provide benefits and that algorithms and data structures have to be specifically designed for the FPGA to provide good performance. From the benchmarks, we conclude that the novel AllCompare set intersection approach provides the best performance and is thus used for our full subgraph query processing system GraphMatch. The AllCompare set intersector performance is bound by memory which we optimize with a cached fetcher implementation that is able to provide increased performance for set intersections with repeated input sets.

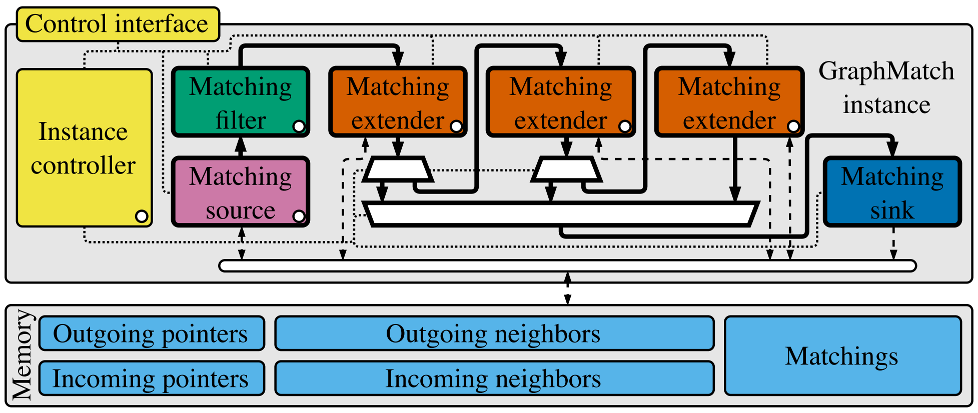


Fig. 7.8 GraphMatch instance architecture.

7.2 GraphMatch

In this section, we first introduce the instance architecture of GraphMatch, our subgraph query processor, and its components. We then describe how subgraph queries can be flexibly switched in GraphMatch during runtime and suitable performance optimizations that we applied to the base system.

7.2.1 GraphMatch Instance Architecture

Figure 7.8 depicts the architecture of a GraphMatch instance for subgraph matchings with up to five levels / vertices. The flow of matchings through the architecture's components is depicted with bold arrows. It starts at the matching source – producing matchings with two query vertices – runs through a matching filter and multiple matching extenders (i. e., a configuration with three extenders results in five levels overall), demultiplexers and multiplexers (trapezoids), and ends at the matching sink. The matching source reads all outgoing pointers and neighbors of each vertex in the graph and combines them as edges, thereby forming the initial partial matchings. Then the matching filter discards initial matchings that do not fit certain criteria like vertices not being distinct in the case of graph isomorphisms. Each matching extender extends input partial matchings by one query vertex in order of the query vertex ordering, which most often means set intersections. After each matching extender, there is a matching demultiplexer, which either forwards the incoming partial matchings to the next matching extender, or through a matching multiplexer to the matching

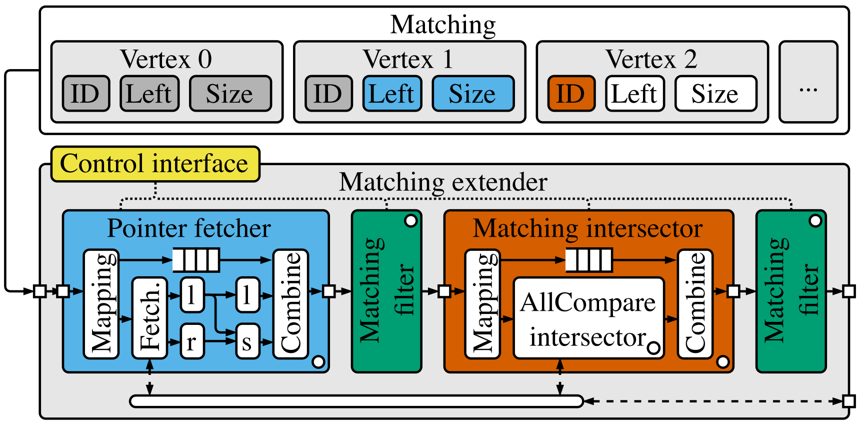


Fig. 7.9 Matching data type and matching extender component.

sink. Finally, the matching sink writes complete matchings to the designated matchings array in the FPGA's on-board memory.

The on-board memory contains in total five data arrays: one CSR data structure consisting of a pointers array and a neighbors array for both incoming and outgoing edges of each vertex, and the matchings array. The matching source and matching extenders only read from memory, whereas the matching sink only writes to memory. These accesses are shown in Fig. 7.8 as dashed lines and are combined into one request stream fed to memory by a request merger (white oval box). The request merger also routes the memory read responses to the corresponding requesters.

The query graph is configured by providing GraphMatch with parameters. All system components with white dots have parameters which are connected to the control interface operated by the CPU (shown by the dotted lines). The matching source is parameterized with the addresses of the arrays it has to read. The matching filter can be turned on and off with parameters and the matching extenders are also parameterized with memory addresses but also, for example, with the number of input sets and which the neighborhoods of which partial matching vertices they should intersect. Finally, the instance controller manages the query's execution. It has a parameter for query size that switches the demultiplexers and multiplexer, is responsbile to trigger the execution when all components are ready, and finally returns relevant statistics to the CPU.

Matching data type Figure 7.9 depicts the component design of a matching extender in the context of the matching data type. Matchings each consist of a configurable number of vertices with a vertex identifier, left bound for pointers, and size that denotes the neighborhood size. The number of vertices in the matching data type is equal to the number of levels in the GraphMatch instance (e. g., five for the instance in Fig. 7.8). The left bound and neighborhood size are kept in the partial matching as metadata between matching extenders. Only when the query, for instance, first requires intersection on the outbound edges of a vertex and then an intersection on the inbound edges or the other way around, we do have to fetch new metadata for this particular vertex from the respective pointers array. The metadata is discarded when writing to memory in the matching sink, since only the vertex identifiers are relevant for the result.

Matching extender The matching extender component at level l takes a matching with l vertices where the vertex at position l is still missing the metadata. It then fetches the required metadata and tries to extend the matching to vertex $l+1$. To do this, the partial matching is acquired through a pointer fetcher, retrieving required metadata, a first matching filter, a matching intersector performing the set intersection required and extending the matching by a vertex, and lastly another matching filter. The pointer fetcher takes a partial matching and a mapping as a parameter and fetches the respective metadata. Dictated by the mapping, a buffered fetcher (fetch.) fetches the lines containing the pointers at positions v and $v+1$, where v is the vertex identifier. These form the left (l) and right (r) bound of the neighborhood. The pointer fetcher subtracts l from r to get the neighborhood size s and, finally, combines the new metadata with the partial matching. The first matching filter filters out empty sets, i. e., sets where any of the neighborhood sizes used in the following intersection are 0. The matching intersector maps the partial matching vertices to intersector spots in the AllCompare intersector, specified in Sect. 7.1, based on a mapping parameter extracted from the query graph. For example, if there is an intersection between vertices 0 and 2, the AllCompare intersector is configured to do a 2 set intersection mapping vertex 0 to spot 0 and vertex 2 to spot 1. During the intersection, the partial matching is stored in a FIFO queue and the combined subcomponent extends the current partial matching until the intersection is finished. It then proceeds with the next partial matching from the FIFO queue. Depending on whether the workload is subgraph isomorphism or homomorphism, the second matching filter sieves out partial matchings for which the newly added vertex is

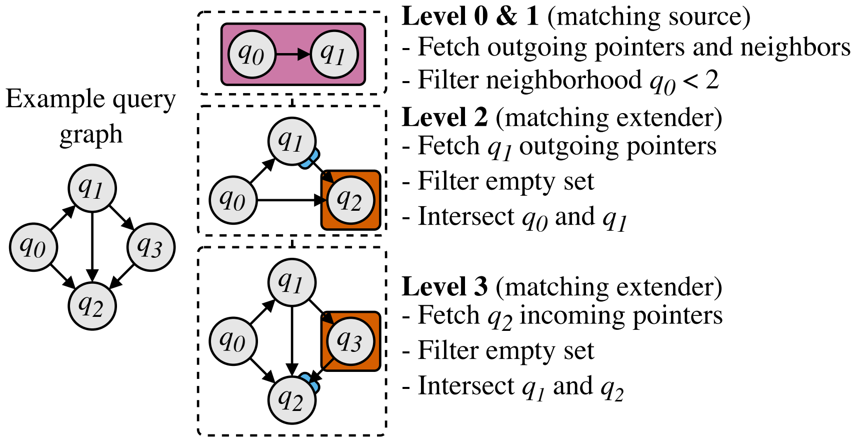

Example query graph
Level 0 & 1 (matching source)
- Fetch outgoing pointers and neighbors
- Filter neighborhood q_0 < 2
Level 2 (matching extender)
- Fetch q_1 outgoing pointers
- Filter empty set
- Intersect q_0 and q_1
Level 3 (matching extender)
- Fetch q_2 incoming pointers
- Filter empty set
- Intersect q_1 and q_2

Fig. 7.10 GraphMatch query parser for query graph Q5.

different from all vertices already part of the partial matching.

7.2.2 Flexible query processing & Optimizations

Based on the GraphMatch architecture, we specify flexible query processing with our query parser and transformations of a query graph into query parameters for GraphMatch. We also explain our four key optimizations for GraphMatch: input set caching, failing set pruning, instance parallelization, and stride mapping.

Flexible query processing Figure 7.10 shows how the example query on the left side is deconstructed by the query parser to get the GraphMatch parameters to map the query graph to the system. The instance controller receives the number of query vertices and address of the matchings array. Each query starts with two vertices connected via an edge. This forms levels 0 and 1 of the GraphMatch instance in the matching source which is parameterized with the addresses of the outgoing pointers and neighbors of q_0. Additionally, the matching filter after the matching source receives the neighborhood size of q_0 in the complete query graph as a parameter. On level 2, the first matching extender is parameterized with a mapping to fetch the metadata for q_1 and the address to outgoing pointers for the pointer fetcher. Additionally, the matching intersector receives a mapping to intersect the neighborhoods of q_0 and q_1 as a two-set intersection on outgoing neighbors. Finally, for level 3, the pointer fetcher should load the incoming pointers for q_2 and we pass a mapping to intersect the neighborhoods of q_1 and q_2 as a two set intersection on outgoing neighbors. If a

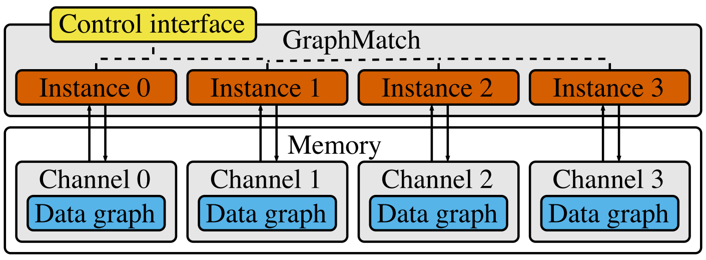


Fig. 7.11 GraphMatch multi-instance scaling.

query is larger than the number of levels of the instance, we can materialize the partial matchings into memory, read them back to the beginning of the matching extender pipeline, and feed them through the levels again. However, details of this are beyond the scope of this work.

Input set caching As a first optimization, we implement input set caching (cf. Sect. 7.1.3). We suspect that locality of accesses exists for the input sets of the AllCompare intersector, as found in each matching intersector, as well as in the accesses of the pointer fetcher. This is especially the case when new metadata for existing vertices has to be loaded for a partial matching. Thus, all instances of the AllCompare intersector and the pointer fetcher employ caching.

Failing set pruning As a second optimization, we introduce failing set pruning [Han+19]. In GraphMatch, we implement this inside the matching filter, after the matching source, and in the first matching filter of each matching extender. In addition to filtering out empty sets, we can also filter partial matchings when the neighborhood of a vertex is not at least as big as the corresponding vertices neighborhood in the query graph. For example, for graph isomorphisms on Q5 (cf. Fig. 7.13), for q_0 the neighborhood size needs to be at least 2. Thus, we parameterize the matching filters for each vertex such that we can customize them for the query vertex neighborhood size.

Instance parallelization Figure 7.11 shows the GraphMatch system scaled to four instances. Each instance is assigned its own memory channel and otherwise only connected to the control interface. The data graph is copied to each memory channel in whole such that the instances can work truly independent. The system can, thus, either process a query on one data graph in parallel such that the vertex set is split up into four intervals each assigned to one instance or process different combinations of queries and data graphs on the four instances independently.

Stride mapping As a last optimization, we apply stride mapping [Dai+17] to improve load balancing across GraphMatch instances. Load balancing was found to be crucial when processing complex data sets. As the GraphMatch instances cannot communicate partial matchings among each other, each vertex interval should require roughly equal amount of work. Unfortunately, this is not a realistic assumption when working with real world graphs [DRF22] which are often skewed. Here, our optimization for stride mapping comes into play by doing a light-weight vertex reordering. Stride mapping is a technique for semi-random shuffling of the vertex identifiers to create a new vertex ordering with a constant stride. In our case, we use a stride of 100 which results in a new vertex order $v_0, v_{100}, v_{200}, ...$ which virtually results in each vertex being mapped to a different GraphMatch instance.

7.3 Evaluation

In this section, we first introduce the system used for the evaluation, the benchmark setup and key metrics such as resource utilization and clock frequency of the design, the graph data sets, and the graph queries. We then report benchmark results scaling GraphMatch from one up to four instances, compare GraphMatch against the state-of-the-art CPU-based subgraph query processing systems GraphFlow and RapidMatch, and evaluate the effects of different optimizations employed in GraphMatch on overall performance.

7.3.1 Setup

Figure 7.12 shows how GraphMatch is deployed in the system context. In principle, the system features a CPU and an accelerator board – connected via PCIe – which hosts the FPGA, running GraphMatch itself, and memory, used as intermediate data storage for the data graph during subgraph query processing. The CPU loads and prepares the data graph, parses and programs the query graph to GraphMatch and manages the execution on the FPGA. To execute a particular workload with a particular data graph, the GraphMatch framework is

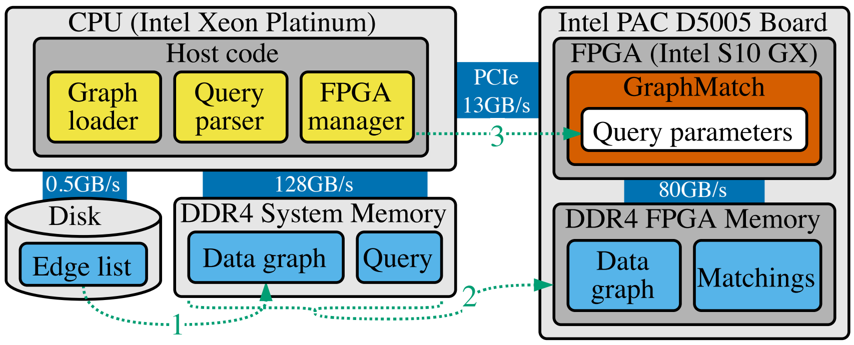


Fig. 7.12 Overall system architecture (incl. host, device, memory).

first synthesized with the query parameter registers, a fixed number of instances and maximum query size (i. e., levels). Afterwards, the synthesized design is programmed to the FPGA.

For the execution of a particular subgraph query on a particular graph data set, the edge list (or any other representation) of the graph is read from disk to the CPU and transformed into two CSR data structures, one for outgoing edges and one for incoming edges of each vertex (cf. step (1)). For undirected graphs, both CSR data structures are the same and one may be omitted. During loading of the data graph, we transform the set of vertex identifiers to be dense (i. e., excluding vertices that have degree 0). The data graph is then replicated to each channel of the FPGA memory (cf. step (2)). Thereafter, the query graph is parsed and the respective parameter registers of GraphMatch are programmed via the control interface such that GraphMatch is configured for the subgraph query matching the query graph. The host code also triggers the execution via the control interface (cf. step (3)). After the execution finished, the resulting matchings can be read back to CPU memory and used for further processing. If desired, the data graph or parameter register values can be used multiple times in a row by loading new query parameters or a new data graph respectively and again triggering the execution.

For our experiments, we work with a server equipped with an Intel FPGA Programmable Accelerator Card (PAC) D5005 attached via PCIe version 3. The system features two Intel Xeon Gold 6142 CPUs at 2.6 GHz and 384 GB of DDR4-2666 memory, while the D5005 board is equipped with four channels of DDR4-2400 memory with a total capacity of 32 GB and a resulting bandwidth of 76.8 GB/s. The design itself is based on the Intel Open Programmable Execution

Table 7.1 Resource utilization and clock frequency by graph problem and number of graph cores.

System	p	l	c	LUTs	Registers	BRAM	DSPs	Clock frequency
GraphMatch	4	6	👎	48%	26%	21%	0%	187 MHz
	4	6	👍	54%	33%	34%	0%	191 MHz

LUTs: Look-up tables, BRAM: Block RAM, DSPs: Digital signal processors, 👍: yes, 👎: no

Table 7.2 Graphs used often by systems in Tab. 7.3 (real-world graphs from [LK14] and [RA15]).

Name	$\|V\|$	$\|E\|$	D_{avg}	ø	SCC
patents (pt)	3.8M	16.5M	4.34	22	1.00
wiki-talk (wt)	2.4M	5.0M	2.10	11	0.05
youtube (yt)	1.2M	3.0M	5.16	20	0.98
google (go)	875.7K	5.1M	5.82	21	0.50
dblp (db)	426.0K	1.0M	4.93	21	0.74
amazon (az)	403.3K	3.4M	8.43	21	0.98
epinions (ep)	75.9K	508.8K	6.70	14	0.43
wiki-vote (wv)	7,115	103.7K	14.56	7	0.18
syn$_{n,d}$	n	$n \cdot d$	d	-	1

SCC: Ratio of vertices in the largest strongly-connected component to $|V|$

Engine (OPAE) platform and is synthesized with Quartus version 19.4.

Table 7.1 shows the two different system configurations used for the benchmarks. We synthesized one system variant without input set caching (c) and one with input set caching for the pointer fetchers and set intersectors. Both variants have $p = 4$ instances of GraphMatch, one for each memory channel, with a maximum query graph size of $l = 6$. Note that, without further resource utilization optimization, up to 8 instances could be placed onto the chip – derived from Tab. 7.1 – however, which could not adequately leverage the available memory channels (cf. discussion on increasing memory bandwidth with HBM in Sect. 7.3.6). Since the instances are not connected, the resource utilization scales linearly excluding the fixed resource utilization of the Intel OPAE wrapper. All types including pointers and vertex identifiers are 32 bit unsigned integers. The resource utilization leaves room to scale to more memory bandwidth or include more functionality, like graph processing (cf. Chapter 5), in the accelerator.

Table 7.2 show the graph data sets used to benchmark GraphMatch. This selection represents the most important graphs considered by the other state-of-the-art subgraph query processing systems (cf. Tab. 7.3). We additionally show graph properties like size ($|V|$ and $|E|$), average degree (D_{avg}), and ratio

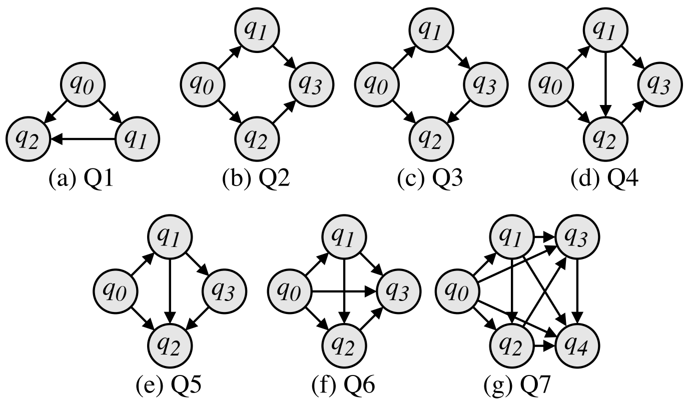


Fig. 7.13 Query graphs (adapted from [MS19]).

of vertices in the largest strongly-connected component (SCC) that are useful to explain different performance effects observed in the benchmarks. For the intersection benchmark, we generated different configurations of a synthetic graph $\text{syn}_{n,d}$ with a parameter for graph size in number of vertices n and a parameter for output size of the resulting intersection between two adjacent vertices d.

Figure 7.13 shows the query graphs we use in our evaluation, taken from [MS19]. These can be classified as cliques (Q1, Q6, and Q7), cycles (Q1, Q2, and Q3), and other graphs (Q4 and Q5).

7.3.2 GraphMatch Scalability

Figure 7.14 shows the scalability of GraphMatch as we increase from a single instance up to four instances. We report the speedup over a baseline using a single instance for all data graphs and queries. When using a single instance, the stride mapping optimization is disabled because it makes no difference for measurements on a single instance. Otherwise, all optimizations discussed in Sect. 7.2.2 are enabled by default. The measurements show that the speedup is mostly dependent on the properties of the data set itself. For example, we observe a linear scalability on the patents and amazon graphs, which is not the case for the other graphs, where scalability is influenced by the number of intermediate (and actual matchings) in the range of vertices assigned to an instance. This effect is particularly pronounced for the highly degree-skewed wiki-talk and youtube graphs. Scalability could be improved in future work, e. g., with work stealing where an idling instance takes over partial matchings from a busy instance to balance out the load. However, this would also introduce data

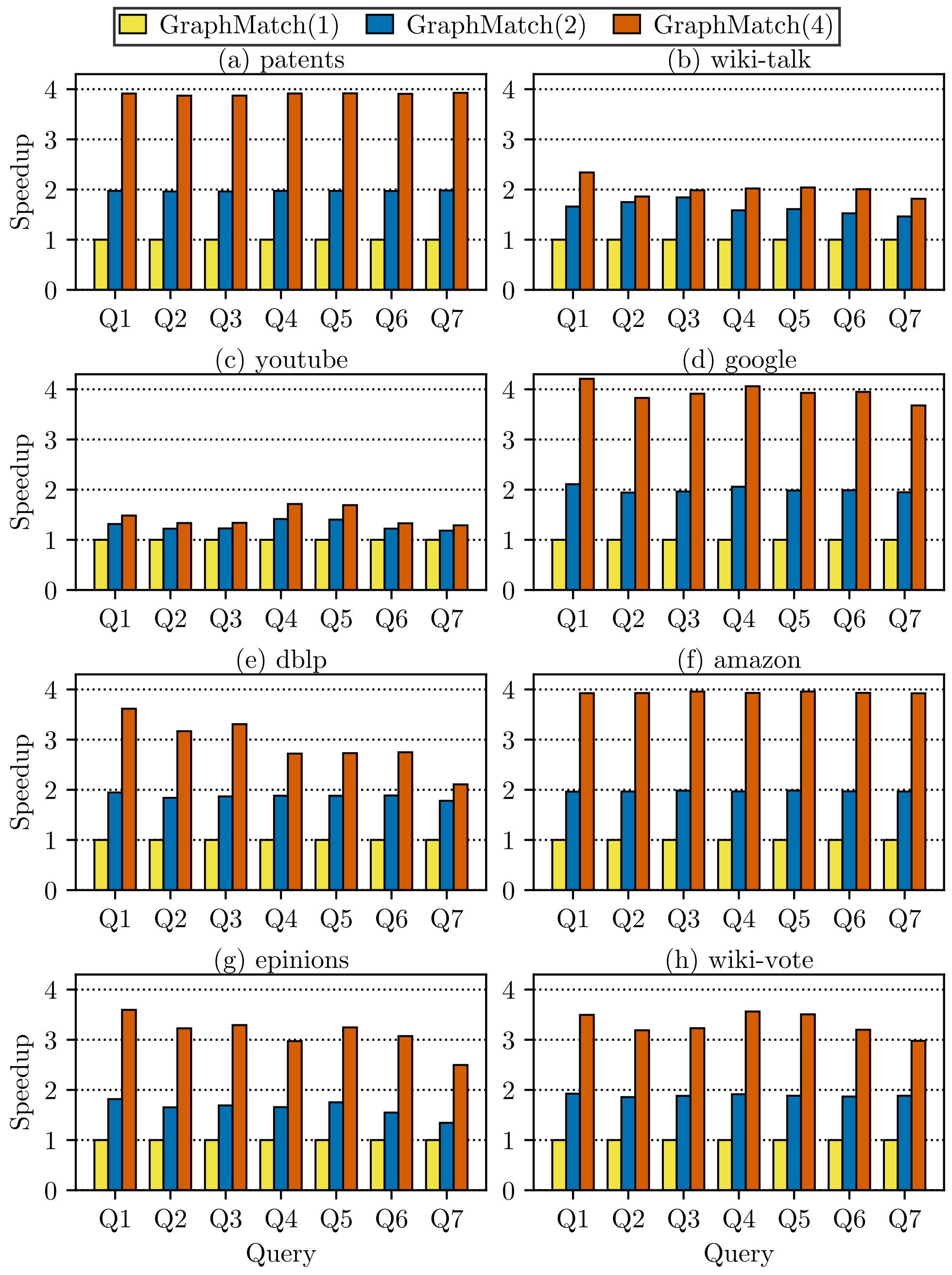


Fig. 7.14 Scalability of GraphMatch from 1 to 4 instances.

dependencies between the instances and could potentially lead to significantly higher design complexity.

7.3.3 Comparison to GraphFlow and RapidMatch

The predominant CPU-based subgraph query processing systems are GraphFlow and RapidMatch which we compare GraphMatch to subsequently.

GraphFlow Figure 7.15 compares the performance of GraphMatch with four instances to GraphFlow [MS19] running on 16 threads (a full CPU on our server) with `numactrl` that restricts all threads to one NUMA node. The plots show the performance in seconds of runtime on a logarithmic scale. For GraphMatch, we tried out different QVOs for each query and data graph combination and show the best one. This makes a huge difference and as future work a query optimizer step could be added to the query parser that does this automatically. GraphFlow finds the best QVO with a preprocessing step. GraphFlow uses directed graphs and only supports subgraph homomorphisms. Thus, to make a fair comparison, we have turned off distinct vertex checking and changed the failing set pruning optimizations to match the subgraph homomorphism workload in GraphMatch. We note that for GraphFlow, we had to execute three scripts that prepare the graphs beforehand for each graph. It was not clear to us how much these scripts optimize the graph layout and compute auxiliary data structures beforehand.

When looking at the performance numbers, overall we observe big performance improvements with GraphMatch. This is especially pronounced for Q1 on all data graphs. GraphMatch also performs exceptionally well for Q4-Q6 on many graphs. Another interesting observation is that Q2 and Q3 pose a challenge for GraphMatch on all graphs. We attribute that to their low intermediate selectivity, which leads to a large number of partial matchings after extending to q_2. Finally, we note that the graph structures of wiki-talk and youtube are more problematic for GraphMatch than for GraphFlow. These graphs exhibit highly exponential (i. e., skewed) degree distributions, which lead to some very large vertex neighborhoods which are used often in partial matchings and poor scalability of GraphMatch (cf. Fig. 7.14). The large vertex neighborhoods cannot be cached with our design which has relatively small caches and are able to be cached by the more sophisticated cache hierarchy of CPUs. In particular,

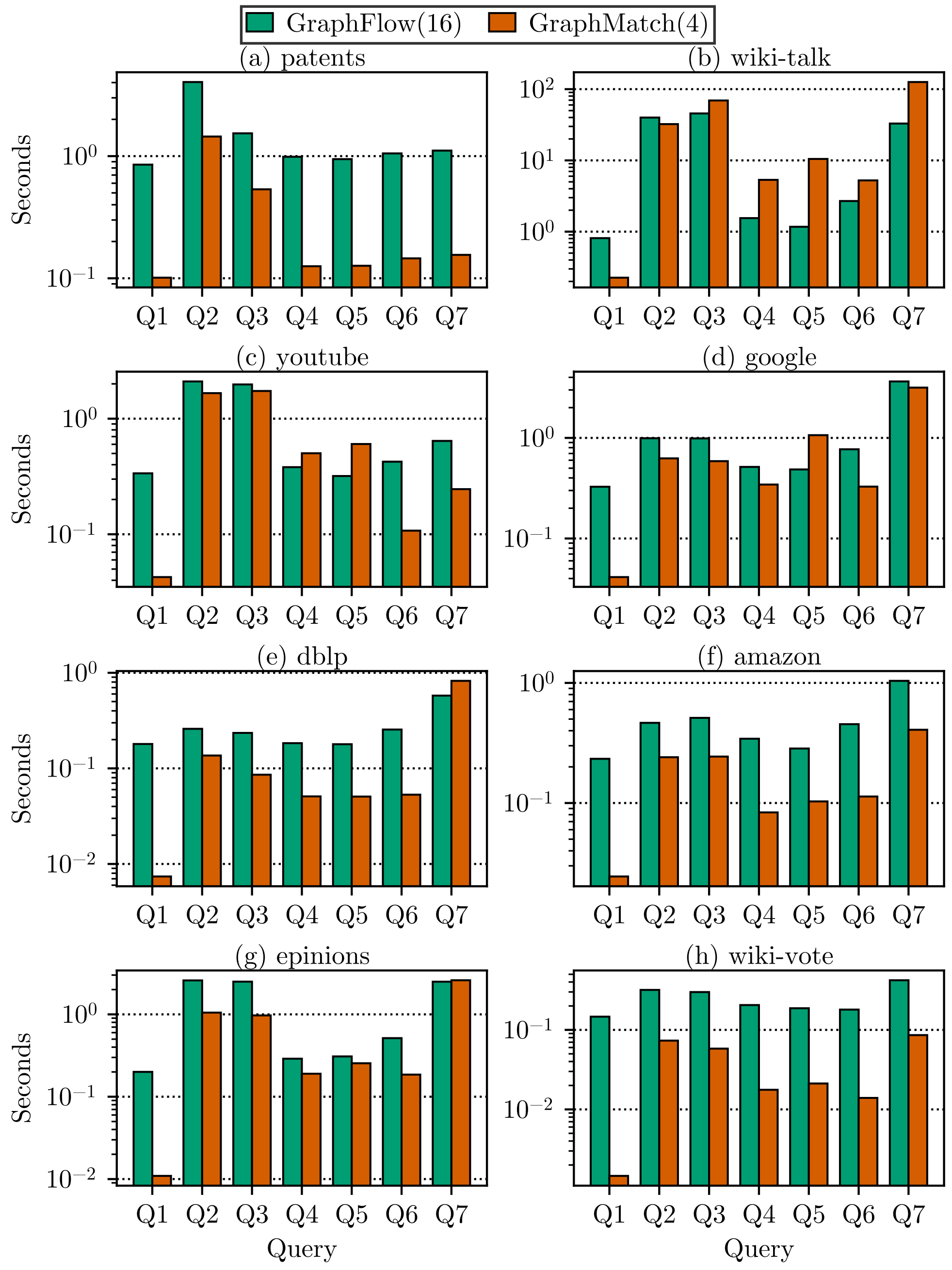


Fig. 7.15 GraphMatch (four instances) vs. GraphFlow (16 threads) on directed graphs computing subgraph homomorphisms.

GraphMatch outperforms GraphFlow by an average of 2.68× for all graphs with a maximum of 100× for query-1 on wiki-vote.

RapidMatch Figure 7.16 compares GraphMatch ran with four instances to RapidMatch [Sun+20a]. We were not able to find any configuration parameter to make RapidMatch run in parallel, so it only uses vectorized instructions for intra-intersection parallelism but no multi-threading. The plots show the runtime of both systems in seconds on a logarithmic scale. RapidMatch only works with undirected graphs, so we also make the graphs undirected for GraphMatch and compute subgraph isomorphisms (even though RapidMatch can also compute homomorphisms). Overall, we observe significantly better performance for GraphMatch compared to RapidMatch, with an average speedup of 5.16×. Similar to the comparison to GraphFlow, GraphMatch performs exceptionally well for Q1. As before, the system performs worse for Q2, Q3, and Q7. We further note that the performance results for Q2 and Q3, as well as for Q4 and Q5 are similar, because in an undirected graph the orientation of the query edges makes no difference. Once again, we note the wiki-talk and youtube graphs are challenging for GraphMatch because of their heavy exponential degree distribution.

7.3.4 Effects of GraphMatch Optimizations

Figure 7.17 shows the effects of the failing set pruning, stride mapping, and input set caching optimizations on GraphMatch performance on the example of the patents and the youtube data graphs. The baseline is GraphMatch with four instances with all optimizations turned off (none). The failing set pruning optimization effectiveness depends on the query graph. For most query graphs it has a small effect but gets more effective with larger query graphs (e. g., Q7). Stride mapping has the largest effect because it balances out the work required between the GraphMatch instances. The effectiveness of input set caching mostly depends on the data graph. It has the biggest effect on the youtube data graph benchmarks. The optimizations work well together to provide a significant performance boost when all of them are turned on (all).

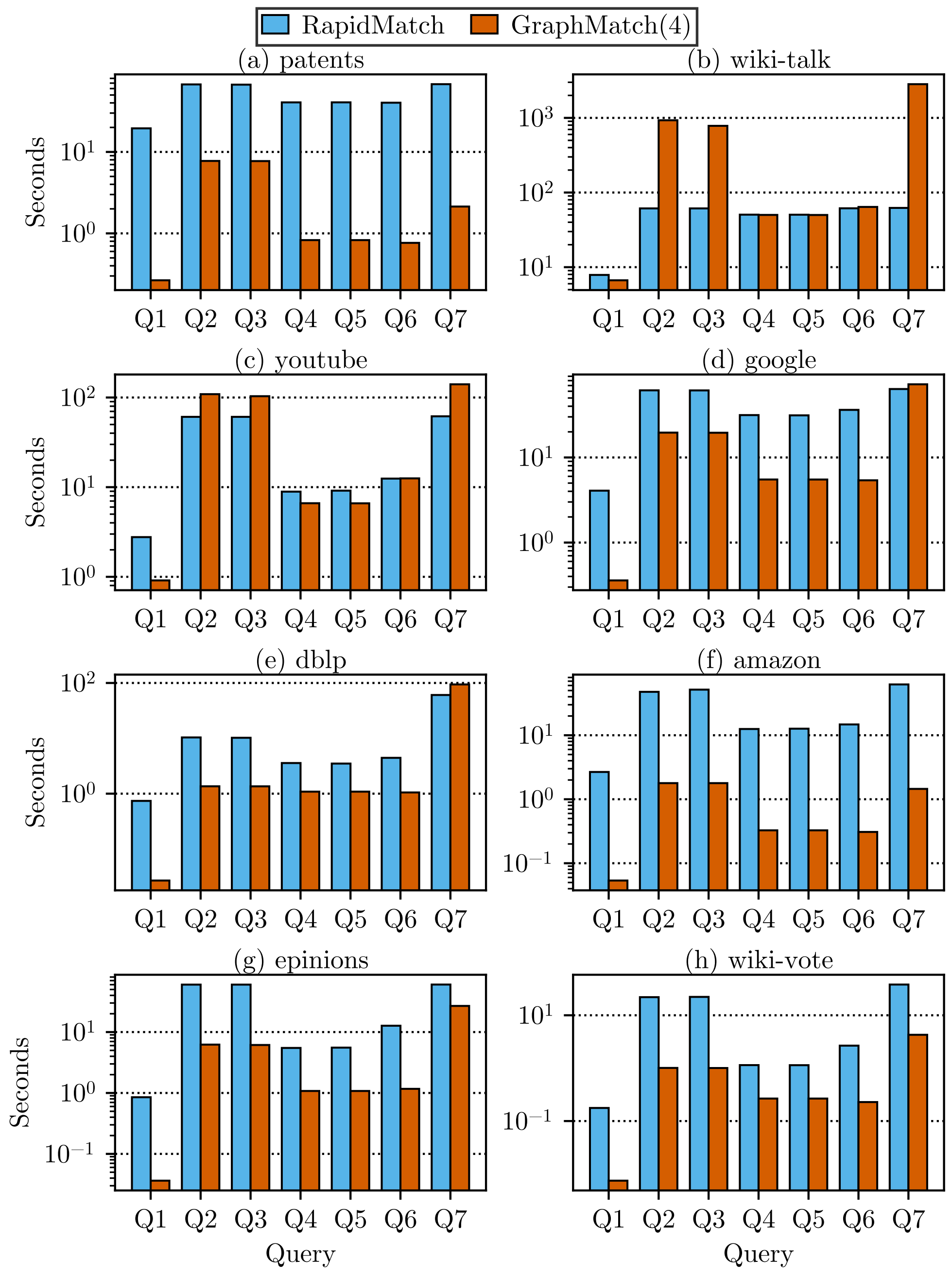


Fig. 7.16 GraphMatch (four instances) vs. RapidMatch on undirected graphs when computing subgraph isomorphisms.

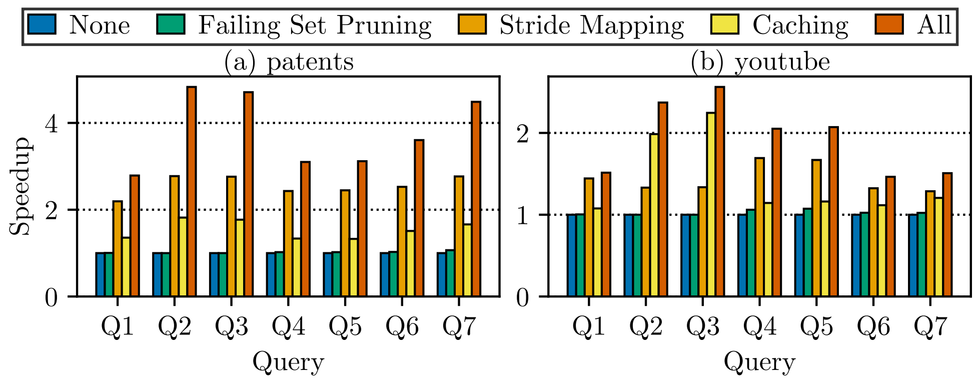


Fig. 7.17 Effects of GraphMatch optimizations for four instances for the patents and youtube data graphs.

7.3.5 Performance Model

The effectiveness of caching showed that memory accesses are the bottleneck of GraphMatch. Thus, we propose a performance model that is based on the number of memory requests with l denoting the number of vertex identifiers or pointers that fit into one line of memory. For instance, for 32 bit values and a 512 bits wide memory interface, l equals 16. To materialize the initial edges in the matching source, we need the following number of memory requests:

$$(|V|+1)/l+|E|/l$$

We need to read $|V|+1$ pointers and $|E|$ edges sequentially. Thus, we can divide these numbers by l. For each extension of the partial matchings we need approximately the following additional amount of memory requests, where f is the number of vertices for which new pointers have to be fetched, m is the number of partial matchings going into this extension, and s is the number of sets being intersected:

$$f\cdot m+s\cdot(m\cdot D_{avg}/\min(l,D_{avg}))$$

We need to fetch $f\cdot m$ pairs of pointers which however mostly are part of the same line of memory such that we only need one request. For each intersector, we need to make $m\cdot D_{avg}/\min(l,D_{avg})$ requests. The minimum term is because if the neighbor hood is smaller than l, we still have to fetch a whole line of memory. However, this may be lowered with caching. For example, $s=1$ for an extension by one edge and $m=|E|$ for the first extension after the matching source.

Table 7.3 Subgraph query processing systems, design principles, and properties.

Name	Platform	Approach	Key data structure	General	Parallel	Dir.	Hom.	Iso.
CFLMatch [Bi+16]	CPU	Backtracking	Compact path index	👍	👎	👎	👎	👍
DAF [Han+19]	CPU	Backtracking	Candidate space	👍	👍	👎	👎	👍
FAST [Jin+21]	FPGA & CPU	Backtracking	Candidate search tree	👍	👍	👎	👎	👍
GraphZero [Maw+21]	CPU	Nested loops	Adjacency lists	👎	👎	👍	👎	👍
EmptyHeaded [Abe+17]	CPU	WCOJ	Trie	👎	👍	👍	👍	👎
GraphFlow [MS19]	CPU	WCOJ & BJ	Adjacency lists	👍	👍	👍	👍	👎
CECI [BLH19]	CPU	Intersections	Compact embedding cluster index	👍	👍	👍	👎	👍
RapidMatch [Sun+20a]	CPU	WCOJ & HJ	Encoded trie	👍	👎	👎	👍	👍
GraphMatch	FPGA	WCOJ	Compressed sparse row	👍	👍	👍	👍	👍

Dir.: Directed graphs; Hom.: Subgraph homomorphisms; Iso.: Subgraph isomorphisms; WCOJ: Worst-case optimal join; BJ: Binary join; HJ: Hash join; 👍: yes; 👎: no

7.3.6 Discussion

This section provided an in-depth analysis of the performance of GraphMatch on various data and query graphs. We observe that the system exhibits linear scalability when the graphs have close to uniform degree distribution; and that the relative performance can be significantly improved with our proposed optimizations (up to 5x on the patents dataset and close to 3x on youtube). A detailed ablation study has demonstrated the effect of each individual optimization strategy and how their effects differ depending on the given query and data graphs. When compared to state-of-the-art systems (GraphFlow and RapidMatch), we have shown that GraphMatch has superior performance, on average outperforming GraphFlow by 2.68×, and RapidMatch by 5.16× across all query and data graphs. When zooming in on the performance for individual queries, we observe that GraphMatch exhibits an excellent performance for all queries when working with graph data sets that have an only slightly skewed degree distribution, and that there is room for improvement when handling cyclic queries like Q2 and Q3 on highly skewed graphs (e.g., youtube and wiki-talk). Additionally, we think because of the moderate resource utilization, almost linear scalability and independence of the instances that GraphMatch is ideal to scale to HBM in the future, possibly delivering another big speedup.

7.4 Related work

Computing subgraph isomorphisms and homomorphisms is an important task studied in related work. Surveys, like Sun et al. [SL20], explored different methods, approaches, and optimizations of subgraph query processing. Lee at al. [Lee+12] introduced generalized models and techniques for subgraph query processing. The result of the increased attention in research for subgraph query processing is a variety of approaches and systems. Table 7.3 depicts an overview of the current state-of-the-art of subgraph query processing systems, their design principles, and how they relate to GraphMatch.

For backtracking-based systems, CFLMatch [Bi+16] and DAF [Han+19] are instrumental and use their custom candidate data structures to allow subgraph matching of general query graphs. CFLMatch decomposes the query graph into a core, a forest, and leaves that are matched in that order because of their decreasing selectivity. DAF uses a candidate space data structure with dynamic programming, an adaptive query vertex ordering, and failing set pruning. FAST [Jin+21] introduces a subgraph query processing system for hybrid CPU-FPGA hardware platforms. It computes its candidate search tree data structure on the CPU prior to moving it to the FPGA [Jin+21]. After the transfer of the structure to the FPGA's BRAM, the FPGA enumerates all subgraphs of the data graph for the given query [Jin+21]. Additionally, it allows concurrent computation with the host CPU to further speedup the computation [Jin+21]. GraphZero [Maw+21] is a compilation-based approach that tries to eliminate redundant computations in a nested loop structure.

EmptyHeaded [Abe+17] introduced the WCOJ approach to subgraph query processing systems as a compilation-based system and uses its trie data structure to support subgraph queries on directed graphs. Additionally, EmptyHeaded supports graph processing. GraphFlow [MS19] extends EmptyHeaded's approach by combining it with binary joins into a hybrid approach with a query optimizer. This allows query plans which combine multiple partial embeddings with the binary join operator. CECI [BLH19] splits up the data graph into embedding clusters to parallelize execution and employs pruning to reduce the number of intermediate matchings. RapidMatch [Sun+20a] is the most recent subgraph query processing system that proves that the backtracking and WCOJ approaches are complexity-wise equal. Based on this observation, it combines a join-based approach with backtracking-like candidate pruning.

7.5 Conclusion

We proposed GraphMatch, an FPGA-based flexible subgraph query processing accelerator and therewith answer research question RQ3.3 *"How can flexible subgraph query processing benefit from FPGAs?"*. GraphMatch is inspired by the insight that current CPU-based subgraph query processing systems are limited by set intersections which FPGAs are well suited for. We showed the potential of such a design by first introducing a novel set intersector AllCompare specifically developed for FPGAs that is able to outperform state-of-the-art CPU intersectors and FPGA adaptations of CPU set intersection algorithms (partially answering research question RQ3.2 *"How can algorithmic software approaches be useful to hardware design?"*). Thereafter, we introduce the GraphMatch architecture with flexible query reconfiguration and four key performance optimizations. Finally, our experimental performance evaluation showed scalability and superior performance of GraphMatch compared to state-of-the-art CPU-based subgraph query processing systems GraphFlow and RapidMatch with an average speedup of 2.68× and 5.16×, respectively.

In future work, we want to extend GraphMatch to benchmarks with labeled graphs and further improve performance for highly degree skewed graphs with more sophisticated caching. Additionally, we consider connecting the GraphMatch instances with a matching crossbar that enables work-stealing and exploring query plan optimization for GraphMatch that could unlock even higher performance.

Chapter 8

Conclusion

In this chapter, we summarize how we answered the research questions set out in Chapter 1 with the contributions of this work. Additionally, we provide an outlook on promising directions for future work.

8.1 Summary

Databases are an integral part of today's everyday life and non-relational database systems in specific are important for many growing application scenarios. With the ever growing amounts of data, acceleration of non-relational database systems is crucial to meet the performance requirements of future workloads. The main goals of this thesis were to address performance challenges C1–3 of CPU-based non-relational database systems (i. e., irregular memory accesses, limited parallelism, and data movement) with FPGA acceleration and accelerator design challenges C4–6 for FPGA acceleration of such systems (i. e., hardware design, low latency workload switching, and novel memory technologies). We achieve this with our work on the state-of-the-art of non-relational databases on FPGAs (survey), graph processing on FPGAs, and flexible data processing on FPGAs.

Survey The aforementioned challenges are motivated by our survey of related work on FPGA-based query acceleration for non-relational databases in Chapter 3. To provide a structure for related work, we proposed a taxonomy of system aspects for non-relational database systems. In a detailed literature analysis, we showed existing solutions to these system aspects. Additionally, we provided a practitioners guide of patterns observed in the literature analysis to aid the design of FPGA-accelerated non-relational database systems. Lastly, we identified open research gaps that we partially addressed with the subsequent chapters.

In summary, we provided an answer to research question RQ1 *"How can non-relational database systems leverage FPGA acceleration?"*. Most existing non-relational database systems do not yet leverage FPGA acceleration even though the potential has been proven. This is due to the extensive gaps in the existing literature that we identified as open research gaps. However, with our practitioners guide we presented patterns that can already be used today to design FPGA-accelerated non-relational database systems.

Graph processing Based on the observation that all graph processing accelerators are inherently memory-bound, in Part I, Chapter 4 introduces the simulation environment GraphSim and memory access models for four state-of-the-art graph processing accelerators. These were subsequently used to reproduce and compare performance along several dimensions. Regarding reproducibility, we are able to approximately simulate the runtime of the selected graph processing accelerators with a reasonable simulation error of 22.63% with only two major outliers. With GraphSim, we were then able to perform an extensive comparison of the accelerator approaches with deep insights into memory accesses. Among other insights and trade-offs, we found three design decisions that are crucial for good performance of graph processing accelerators: asynchronous graph processing, a compressed graph data structure, and scalability to multiple memory channels.

These insights from the GraphSim analysis were used in Chapter 5 to design GraphScale. GraphScale is the first scalable graph processing framework for FPGAs based on asynchronous graph processing on a compressed graph. We achieved this with our novel two level vertex label crossbar design and novel two-dimensional partitioning scheme. Overall, GraphScale achieves 1.86× average speedup over state-of-the-art graph processing accelerators and is only limited by workload-inherent imbalances in crossbar accesses. Additionally, we showed how GraphScale can be scaled to up to 16 HBM memory channels with an average speedup of 1.53× over GraphScale running on just four channels of DDR4 memory. However, this still only partially utilizes the memory bandwidth available with HBM due to severely limited clock frequency and resource utilization on the

FPGA platform available to us. We think that future FPGA hardware will solve these issues. However, we also discussed another possibility. In this particular case, where the accelerator is not limited by the memory bandwidth anymore, we could remove the highly complex graph compression again in favor of lower resource utilization and thus potentially better scalability.

In summary, in Part I, we addressed research question RQ2 *"How can the FPGA use the available bandwidth for irregular memory accesses most effectively?"*. For graph processing in particular, we identified – among other optimizations – two crucial accelerator design decisions for bandwidth efficiency: asynchronous graph processing and a compressed graph data structure. We showed how these design decisions can be scaled on FPGAs with GraphScale. In our measurements, this lead to up to 2.22× fewer memory requests compare to other state-of-the-art approaches which is a massive improvement in bandwidth efficiency for this workload bottlenecked by irregular memory accesses.

Flexible data processing With Part II, we focused on flexible data ingestion and query processing. We first introduced the PipeJSON FPGA-based JSON parser in Chapter 6. PipeJSON is able to parse arbitrary JSON documents with a flexible, pipelined, data parallel system architecture. With the evaluation, we showed that PipeJSON outperforms the fastest, vectorized, CPU-based JSON parsers by 12.56× when all data is already on the FPGA and 7.95× including data transfers to the FPGA for our drop-in parser configuration, connected via PCIe, only limited by the PCIe version 3 bandwidth of our setup.

Thereafter, in Chapter 7, we proposed GraphMatch as a flexible subgraph query processing accelerator. The performance defining operation of subgraph query processing is set intersection. We showed how set intersection benefits from acceleration with the novel AllCompare set intersector specifically designed for FPGAs. AllCompare provides an average speedup of 4.77× over vectorized, CPU-based set intersection. We then showed how this flexible set intersector can be chained together as a parameterized pipeline in GraphMatch. Our experimental performance evaluation showed linear scalability and an average speedup of 2.68× and 5.16× over the state-of-the-art CPU-based subgraph query processing systems GraphFlow (scaled to 16 threads) and RapidMatch, respectively. All while being able to flexibly switch queries in a matter of clock cycles. GraphMatch performance is currently only limited by memory bandwidth.

In summary, in Part II, we addressed research question RQ3 *“How can flexible data ingestion and query processing be achieved on FPGAs?”* We showed how to design flexible data flow abstractions for FPGAs with PipeJSON and GraphMatch. Both of these accelerators provide a significant speedup over the fastest CPU-based systems currently available.

Overall contribution Altogether, this thesis contributes new insights into the related work in the research area of FPGA-accelerated non-relational database systems (survey) and graph processing on FPGAs in particular (GraphSim). The three FPGA accelerators GraphScale, PipeJSON, and GraphMatch further show the benefits of using FPGAs for non-relational database systems and how to design scalable, flexible accelerators for non-relational database workloads.

8.2 Outlook

Besides the system-specific future work that we discuss in the conclusion of each chapter, we identify six additional overarching directions for future work.

Non-functional system aspects As a broad trend in the literature, the coverage of non-functional non-relational database system aspects like cluster scalability, availability, multi-tenancy, and security is lacking. While, for instance, consistency protocols may transferred from the non-accelerated non-relational database systems literature, it is broadly unclear how to provide production-grade solutions for most non-functional system aspects with an FPGA-accelerated non-relational database system. Additionally, FPGAs as a relatively new processor architecture for data processing are not integrated as deeply into current systems and do, in contrast to CPUs, not have widely used operating systems providing basic functionality [KRA20]. Thus, we see investigation of non-functional system aspects as one possible direction for future research.

Benchmarks We want to reiterate that benchmarking in the FPGA accelerator related work is very much lacking. There are no generally accepted complete benchmarks and almost no generally accepted data sets and workloads that enable comprehensive comparison of accelerator approaches. Additionally, performance-critical input parameters are often not specified in detail. Future work should establish accepted benchmarks to stop cherry-picking of performance results.

Collaborative memory usage Data movement overhead dominates accelerator performance for certain workloads and narrows their potential for performance improvements. The emergence of cache-coherent attachments (e. g., CXL) of FPGAs to the system main memory might alleviate this. FPGA-directed data movement and orchestration could take pressure off the CPU and also make more fine-grained acceleration possible.

Dynamic data structures One remaining issue with FPGA accelerators is query answering latency and data freshness due to necessary data movement between main memory and FPGA memory if the data cannot be permanently kept on the FPGA. This can be solved with dynamic data structures that tolerate updates. With such a data structure, for instance, a dynamic graph data structure, updates (e. g., new edges) can be communicated to the FPGA as small packages – massively reducing data movement – but have to be integrated into the existing data structure which increases the design complexity.

Accelerator integration We see potential in integrating accelerators in general and the accelerators presented in this thesis in particular. For instance, for graph database systems integrating GraphScale for graph processing and GraphMatch for subgraph query processing could potentially cover most of the performance defining parts of the workload. The challenge for this lies in finding common abstractions to be able to integrate the functionality for both accelerators on the same resource-constrained FPGA. Additionally, PipeJSON could be used for cross data model processing of document graphs with GraphScale and GraphMatch.

Heterogeneous computing With the increasing availability of GPUs and FPGAs in the data center and the emergence of ever more specialized hardware, we see potential for heterogeneous computing to satisfy the performance requirements of future database management systems [Kou+21]. This poses unique challenges especially with data movement and control flow inside the computer architecture between accelerators and the CPU. FPGAs are in a unique position because of their flexibility in deployments and roles they can take in the computer architecture (cf. Sect. 3.3).

References

[Aba12] Daniel Abadi. “Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story”. In: *IEEE Computer* 45.2 (2012), pp. 37–42.

[Aba+19] Daniel Abadi et al. “The Seattle Report on Database Research”. In: *SIGMOD Rec.* 48.4 (2019), pp. 44–53.

[Abe+17] Christopher R. Aberger et al. “EmptyHeaded: A Relational Engine for Graph Processing”. In: *ACM Trans. Database Syst.* 42.4 (2017), 20:1–20:44.

[ABK14] Abdessalem Abidi, Belgacem Bouallegue, and Fatma Kahri. “Implementation of elliptic curve digital signature algorithm (ECDSA)”. In: *GSCIT*. 2014, pp. 1–6.

[Abi+18] Serge Abiteboul et al. “Research Directions for Principles of Data Management (Dagstuhl Perspectives Workshop 16151)”. In: *Dagstuhl Manifestos* 7.1 (2018), pp. 1–29.

[AOA20] Matthew Agostini, Francis O’Brien, and Tarek S. Abdelrahman. “Balancing Graph Processing Workloads Using Work Stealing on Heterogeneous CPU-FPGA Systems”. In: *ICPP*. 2020, 50:1–50:12.

[Ahn+15] Junwhan Ahn et al. “A Scalable Processing-in-memory Accelerator for Parallel Graph Processing”. In: *ISCA*. 2015, pp. 105–117.

[Alo+20] Gustavo Alonso et al. “Tackling Hardware/Software Co-design from a Database Perspective”. In: *CIDR, Online Proceedings*. 2020.

[AM10] Vo Ngoc Anh and Alistair Moffat. “Index Compression using 64-bit Words”. In: *Softw. Pract. Exp.* 40.2 (2010), pp. 131–147.

[App16] Austin Appleby. *SMHasher*. https://github.com/aappleby/smhasher. visited 10/2023. 2016.

[AI21] Mikhail Asiatici and Paolo Ienne. “Large-Scale Graph Processing on FPGAs with Caches for Thousands of Simultaneous Misses”. In: *ISCA*. 2021, pp. 609–622.

[Att+14] Osama G. Attia et al. “CyGraph: A Reconfigurable Architecture for Parallel Breadth-First Search”. In: *IPDPS*. 2014, pp. 228–235.

[Att+15] Osama G. Attia et al. “Accelerating All-pairs Shortest Path using a Message-passing Reconfigurable Architecture”. In: *ReConFig*. 2015, pp. 1–6.

[Aus+12] Chad Austin et al. *sajson.* https://github.com/chadaustin/sajson. visited 10/2023. 2012.

[Ayu+18] Andrey Ayupov et al. "A Template-Based Design Methodology for Graph-Parallel Hardware Accelerators". In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 37.2 (2018), pp. 420–430.

[BFA96] Jonathan Babb, Matthew I. Frank, and Anant Agarwal. "Solving Graph Problems with Dynamic Computation Structures". In: *Other Conferences.* 1996.

[BRS13] David F. Bacon, Rodric M. Rabbah, and Sunil Shukla. "FPGA Programming for the Masses". In: *ACM Queue* 11.2 (2013), p. 40.

[BL18] Vignesh Balaji and Brandon Lucia. "When is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs". In: *IISWC.* 2018, pp. 203–214.

[Bec+18] Andreas Becher et al. "Integration of FPGAs in Database Management Systems: Challenges and Opportunities". In: *Datenbank-Spektrum* 18.3 (2018), pp. 145–156.

[Bec+15] Andreas Becher et al. "A Co-design Approach for Accelerated SQL Query Processing via FPGA-based Data Filtering". In: *FPT.* 2015, pp. 192–195.

[Bes+19a] Maciej Besta et al. "Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries". In: *CoRR* abs/1910.09017 (2019).

[Bes+19b] Maciej Besta et al. "Graph Processing on FPGAs: Taxonomy, Survey, Challenges". In: *CoRR* abs/1903.06697 (2019).

[Bes+19c] Maciej Besta et al. "Substream-Centric Maximum Matchings on FPGA". In: *FPGA.* 2019, pp. 152–161.

[Bet+11] Brahim Betkaoui et al. "A Framework for FPGA Acceleration of Large Graph Problems: Graphlet Counting Case Study". In: *FPT.* 2011, pp. 1–8.

[Bet+12a] Brahim Betkaoui et al. "A Reconfigurable Computing Approach for Efficient and Scalable Parallel Graph Exploration". In: *ASAP.* 2012, pp. 8–15.

[Bet+12b] Brahim Betkaoui et al. "Parallel FPGA-based All-pairs Shortest Paths for Sparse Networks: a Human Brain Connectome Case Study". In: *FPL.* 2012, pp. 99–104.

[BLH19] Bibek Bhattarai, Hang Liu, and H. Howie Huang. "CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching". In: *SIGMOD.* 2019, pp. 1447–1462.

[Bi+16] Fei Bi et al. "Efficient Subgraph Matching by Postponing Cartesian Products". In: *SIGMOD.* 2016, pp. 1199–1214.

[Blo+13] Michaela Blott et al. “Achieving 10Gbps Line-rate Key-value Stores with FPGAs”. In: *HotCloud*. 2013.

[Blo+15] Michaela Blott et al. “Scaling Out to a Single-Node 80Gbps Memcached Server with 40 Terabytes of Memory”. In: *HotStorage*. 2015.

[Bob+22] Christophe Bobda et al. “The Future of FPGA Acceleration in Datacenters and the Cloud”. In: *ACM Trans. Reconfigurable Technol. Syst.* 15.3 (2022), 34:1–34:42.

[Bon+06a] Uday Bondhugula et al. “Hardware/Software Integration for FPGA-based All-Pairs Shortest-Paths”. In: *FCCM*. 2006, pp. 152–164.

[Bon+06b] Uday Bondhugula et al. “Parallel FPGA-based All-pairs Shortest-paths in a Directed Graph”. In: *IPDPS*. 2006.

[Bor+22] Dhruba Borthakur et al. *RocksDB Overview*. https://github.com/facebook/rocksdb/wiki/RocksDB-Overview. visited 10/2023. 2022.

[Bra+14] Tim Bray et al. “The JavaScript Object Notation (JSON) Data Interchange Format”. In: (2014).

[Bra+00] Tim Bray et al. *Extensible markup language (XML) 1.0*. 2000.

[Bre+14] Sebastian Breß et al. “GPU-Accelerated Database Systems: Survey and Open Challenges”. In: *Trans. Large-Scale Data- and Knowl.-Cent. Systems* 15 (2014), pp. 1–35.

[Bre12] Eric Brewer. “CAP Twelve Years Later: How the "Rules" Have Changed”. In: *IEEE Computer* 45.2 (2012), pp. 23–29.

[CHL08] Robert D. Cameron, Kenneth S. Herdy, and Dan Lin. “High Performance XML Parsing Using Parallel Bit Stream Technology”. In: *CASCON*. 2008, p. 17.

[Cat10] Rick Cattell. “Scalable SQL and NoSQL Data Stores”. In: *SIGMOD Rec.* 39.4 (2010), pp. 12–27.

[Cha+12a] Sai Rahul Chalamalasetti et al. “Evaluating FPGA-acceleration for Real-time Unstructured Search”. In: *IEEE ISPASS*. 2012, pp. 200–209.

[Cha+12b] Niladrish Chatterjee et al. “USIMM: the Utah SImulated Memory Module”. In: *University of Utah, Tech. Rep.* (2012), pp. 1–24.

[Che+19] Xinyu Chen et al. “On-The-Fly Parallel Data Shuffling for Graph Processing on OpenCL-Based FPGAs”. In: *FPL*. 2019, pp. 67–73.

[Che+21a] Xinyu Chen et al. “Skew-Oblivious Data Routing for Data Intensive Applications on FPGAs with HLS”. In: *DAC*. 2021, pp. 937–942.

[Che+21b] Xinyu Chen et al. “ThunderGP: HLS-based Graph Processing Framework on FPGAs”. In: *FPGA*. 2021, pp. 69–80.

[Che+22] Xinyu Chen et al. “ThunderGP: Resource-Efficient Graph Processing Framework on FPGAs with HLS”. In: *ACM Trans. Reconfigurable Technol. Syst.* 15.4 (2022), 44:1–44:31.

[Chi+15] Avery Ching et al. “One Trillion Edges: Graph Processing at Facebook-Scale”. In: *PVLDB* 8.12 (2015), pp. 1804–1815.

[CC14] Jae Min Cho and Kiyoung Choi. “An FPGA Implementation of High-throughput Key-value Store using Bloom Filter”. In: *International Symposium on VLSI Design, Automation and Test.* 2014, pp. 1–4.

[CG03] Pawel Chodowiec and Kris Gaj. “Very Compact FPGA Implementation of the AES Algorithm”. In: *CHES*. Vol. 2779. 2003, pp. 319–333.

[Cod70] E. F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Commun. ACM* 13.6 (1970), pp. 377–387.

[Coo+10] Brian F. Cooper et al. “Benchmarking Cloud Serving Systems with YCSB”. In: *ACM Symposium on Cloud Computing.* 2010, pp. 143–154.

[Dai+16] Guohao Dai et al. “FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search”. In: *FPGA*. 2016, pp. 105–110.

[Dai+17] Guohao Dai et al. “ForeGraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture”. In: *FPGA*. 2017, pp. 217–226.

[DNZ10] Zefu Dai, Nick Ni, and Jianwen Zhu. “A 1 Cycle-per-Byte XML Parsing Accelerator”. In: *FPGA*. 2010, pp. 199–208.

[DMP99] Andreas Dandalis, Alessandro Mei, and Viktor K. Prasanna. “Domain Specific Mapping for Solving Graph Problems on Reconfigurable Devices”. In: *IPPS*. 1999, pp. 652–660.

[DRF21a] Jonas Dann, Daniel Ritter, and Holger Fröning. “Demystifying Memory Access Patterns of FPGA-based Graph Processing Accelerators”. In: *GRADES-NDA*. 2021, 3:1–3:10.

[DRF21b] Jonas Dann, Daniel Ritter, and Holger Fröning. “Exploring Memory Access Patterns for Graph Processing Accelerators”. In: *BTW*. 2021, pp. 101–122.

[DRF22] Jonas Dann, Daniel Ritter, and Holger Fröning. “GraphScale: Scalable Bandwidth-Efficient Graph Processing on FPGAs”. In: *FPL*. 2022, pp. 24–32.

[DRF23a] Jonas Dann, Daniel Ritter, and Holger Fröning. “GraphScale: Scalable Processing on FPGAs for HBM and Large Graphs”. In: *ACM Trans. Reconfigurable Technol. Syst.* Just Accepted (2023), pp. 1–24.

[DRF23b] Jonas Dann, Daniel Ritter, and Holger Fröning. “Non-relational Databases on FPGAs: Survey, Design Decisions, Challenges”. In: *ACM Comput. Surv.* 55.11 (2023), 225:1–225:37.

[Dan+22] Jonas Dann et al. "PipeJSON: Parsing JSON at Line Speed on FPGAs". In: *DaMoN*. 2022, 3:1–3:7.

[Dan+24] Jonas Dann et al. "GraphMatch: Subgraph Query Processing on FPGAs". In: *SIGMOD*. Submitted (2024), pp. 1–12.

[Dav95] Andrew Davison. "Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers". In: (1995), pp. 38–42.

[DCL18] Ali Davoudian, Liu Chen, and Mengchi Liu. "A Survey on NoSQL Stores". In: *ACM Comput. Surv.* 51.2 (2018), 40:1–40:43.

[DeC+07] Giuseppe DeCandia et al. "Dynamo: Amazon's Highly Available Key-value Store". In: *SOSP*. 2007, pp. 205–220.

[DeL+06] Michael DeLorimier et al. "GraphStep: a System Architecture for Sparse-Graph Algorithms". In: *FCCM*. 2006, pp. 143–151.

[Den+74] Robert H Dennard et al. "Design of Ion-implanted MOSFET's with Very Small Physical Dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268.

[Dre07] Ulrich Drepper. "What Every Programmer Should Know About Memory". In: *Red Hat, Inc* 11 (2007).

[DLN21] Dominik Durner, Viktor Leis, and Thomas Neumann. "JSON Tiles: Fast Analytics on Semi-Structured Data". In: *SIGMOD*. 2021, pp. 445–458.

[EI10] Fadi El-Hassan and Dan Ionescu. "A Hardware Architecture of an XML/XPath Broker for Content-Based Publish/Subscribe Systems". In: *ReConFig*. 2010, pp. 138–143.

[EHS18] Nina Engelhardt, C.-H. Dominic Hung, and Hayden Kwok-Hay So. "Performance-Driven System Generation for Distributed Vertex-Centric Graph Processing on Multi-FPGA Systems". In: *FPL*. 2018, pp. 215–218.

[ES16] Nina Engelhardt and Hayden Kwok-Hay So. "GraVF: a Vertex-centric Distributed Graph Processing Framework on FPGAs". In: *FPL*. 2016, pp. 1–4.

[Eve11] Shimon Even. *Graph Algorithms*. Cambridge University Press, 2011.

[Fan+20] Jian Fang et al. "In-memory Database Acceleration on FPGAs: a Survey". In: *VLDB J.* 29.1 (2020), pp. 33–59.

[Fin+19] Eric Finnerty et al. "Dr. BFS: Data Centric Breadth-First Search on FPGAs". In: *Annual Design Automation Conference*. 2019, p. 208.

[Fou22] Apache Software Foundation. *Apache CouchDB Documentation*. https://docs.couchdb.org/en/stable. visited 10/2023. 2022.

[Fow+14] Jeremy Fowers et al. "A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication". In: *FCCM*. 2014, pp. 36–43.

[Fra+18] Nadime Francis et al. “Cypher: An Evolving Query Language for Property Graphs”. In: *SIGMOD*. 2018, pp. 1433–1445.

[Fra+11] Phil Francisco et al. *The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics.* 2011.

[Gaj12] Santhosh Kumar Gajendran. “A Survey on NoSQL Databases”. In: *University of Illinois* (2012).

[Ges+17] Felix Gessert et al. “NoSQL Database Systems: a Survey and Decision Guidance”. In: *Comput. Sci. Res. Dev.* 32.3-4 (2017), pp. 353–365.

[Gho+19a] Saugata Ghose et al. “Demystifying Complex Workload-DRAM Interactions: An Experimental Study”. In: *Proc. ACM Meas. Anal. Comput. Syst.* 3.3 (2019), 60:1–60:50.

[Gho+19b] Saugata Ghose et al. “Understanding the Interactions of Workloads and DRAM Types: A Comprehensive Experimental Study”. In: *CoRR* abs/1902.07609 (2019).

[GSP16] Heiner Giefers, Peter W. J. Staar, and Raphael Polig. “Energy-efficient Stochastic Matrix Function Estimator for Graph Analytics on FPGA”. In: *FPL*. 2016, pp. 1–9.

[Gra93] Jim Gray. “The Benchmark Handbook for Database and Transasction Systems”. In: *Morgan Kaufmann* (1993).

[Gui+19] Chuangyi Gui et al. “A Survey on Graph Processing Accelerators: Challenges and Opportunities”. In: *J. Comput. Sci. Technol.* 34.2 (2019), pp. 339–371.

[Ham+16] Tae Jun Ham et al. “Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics”. In: *MICRO*. 2016, 56:1–56:13.

[HPM12] Volker Hampel, Thilo Pionteck, and Erik Maehle. “An Approach for Performance Estimation of Hybrid Systems with FPGAs and GPUs as Coprocessors”. In: *Architecture of Computing Systems*. 2012, pp. 160–171.

[Han+19] Myoungji Han et al. “Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together”. In: *SIGMOD*. 2019, pp. 1429–1446.

[HZY18] Shuo Han, Lei Zou, and Jeffrey Xu Yu. “Speeding Up Set Intersections in Graph Algorithms using SIMD Instructions”. In: *SIGMOD*. 2018, pp. 1587–1602.

[He15] Changlin He. “Survey on NoSQL Database Technology”. In: *J. of Applied Sci. and Eng. Innovation* 2.2 (2015).

[HP19] John L. Hennessy and David A. Patterson. “A New Golden Age for Computer Architecture”. In: *Commun. ACM* 62.2 (2019), pp. 48–60.

[HB15] Torsten Hoefler and Roberto Belli. “Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results”. In: *SC*. 2015, 73:1–73:12.

[Hol+21] Philipp Holzinger et al. “Fast HBM Access with FPGAs: Analysis, Architectures, and Applications”. In: *IPDPS*. 2021, pp. 152–159.

[Hu+21] Yuwei Hu et al. “GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs”. In: *ICCAD*. 2021, pp. 1–9.

[Hua+19] Gui Huang et al. “X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing”. In: *SIGMOD*. 2019, pp. 651–665.

[Hua+14] Linan Huang et al. “A Slide-Window-Based Hardware XML Parsing Accelerator”. In: *CCF National Conference on Computer Engineering and Technology*. Springer. 2014, pp. 108–117.

[Hue00] Lorenz Huelsbergen. “A Representation for Dynamic Graphs in Reconfigurable Hardware and its Application to Fundamental Graph Algorithms”. In: *FPGA*. 2000, pp. 105–115.

[Huf52] David A Huffman. “A Method for the Construction of Minimum-redundancy Codes”. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[Huf+08] Ted Huffmire et al. “Managing Security in FPGA-Based Embedded Systems”. In: *IEEE Des. Test Comput.* 25.6 (2008), pp. 590–598.

[inc22a] MongoDB inc. *The MongoDB Manual*. https://docs.mongodb.com/manual. visited 10/2023. 2022.

[inc21] Neo4j inc. *Neo4j Documentation*. https://neo4j.com/docs/. visited 10/2023. 2021.

[inc22b] Redis inc. *Redis Documentation*. https://docs.redislabs.com/latest/index.html. visited 10/2023. 2022.

[Int19] Intel. *Intel FPGA Programmable Acceleration Card D5005 Data Sheet*. https://cdrdv2-public.intel.com/691516/ds-pac-d5005-683568-691516.pdf. visited 10/2023. 2019.

[Int22] Intel. *Intel Stratix 10 Logic Array Blocks and Adaptive Logic Modules User Guide*. https://www.intel.com/content/www/us/en/docs/programmable/683699/current/alm.html. visited 10/2023. 2022.

[Ist20] Zsolt István. “Let’s Add Transactions to FPGA-based Key-value Stores!” In: *DaMoN*. 2020, 13:1–13:3.

[IAS18] Zsolt István, Gustavo Alonso, and Ankit Singla. “Providing Multi-tenant Services with FPGAs: Case Study on a Key-Value Store”. In: *FPL*. 2018, pp. 119–124.

[ISA16] Zsolt István, David Sidler, and Gustavo Alonso. “Runtime Parameterizable Regular Expression Operators for Databases”. In: *FCCM*. 2016, pp. 204–211.

[ISA17] Zsolt István, David Sidler, and Gustavo Alonso. “Caribou: Intelligent Distributed Storage”. In: *PVLDB* 10.11 (2017), pp. 1202–1213.

[Ist+13] Zsolt István et al. “A Flexible Hash Table Design for 10GBps Key-value Stores on FPGAs”. In: *FPL*. 2013, pp. 1–8.

[JSL11] George Rosario Jagadeesh, Thambipillai Srikanthan, and C. M. Lim. “Field-programmable Gate Array-based Acceleration of Shortest-path Computation”. In: *IET Computers & Digital Techniques* 5.4 (2011), pp. 231–237.

[JL19] Akanksha Jain and Calvin Lin. “Cache Replacement Policies”. In: *Synthesis Lectures on Computer Architecture* 14.1 (2019), pp. 1–87.

[JQZ20] Lin Jiang, Junqiao Qiu, and Zhijia Zhao. “Scalable Structural Index Construction for JSON Analytics”. In: *PVLDB* 14.4 (2020), pp. 694–707.

[Jin+17] Hai Jin et al. “Towards Dataflow-Based Graph Accelerator”. In: *ICDCS*. 2017, pp. 1981–1992.

[Jin+21] Xin Jin et al. “FAST: FPGA-based Subgraph Matching on Massive Graphs”. In: *ICDE*. 2021, pp. 1452–1463.

[Jin+11] Jing Han et al. “Survey on NoSQL Database”. In: *International Conference on Pervasive Computing and Applications*. 2011, pp. 363–366.

[Kap15] Nachiket Kapre. “Custom FPGA-based Soft-processors for Sparse Graph acceleration”. In: *ASAP*. 2015, pp. 9–16.

[Kho+18] Soroosh Khoram et al. “Accelerating Graph Analytics by Co-Optimizing Storage and Access on an FPGA-HMC Platform”. In: *FPGA*. 2018, pp. 239–248.

[KYM16] Yoongu Kim, Weikun Yang, and Onur Mutlu. “Ramulator: A Fast and Extensible DRAM Simulator”. In: *IEEE Comput. Archit. Lett.* 15.1 (2016), pp. 45–49.

[Kit04] Barbara Kitchenham. “Procedures for Performing Systematic Reviews”. In: *Keele, UK, Keele University* 33.2004 (2004), pp. 1–26.

[KRA20] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. “Do OS Abstractions Make Sense on FPGAs?” In: *OSDI*. 2020, pp. 991–1010.

[Kou+21] Dimitrios Koutsoukos et al. “Modularis: Modular Relational Analytics over Heterogeneous Distributed Platforms”. In: *Proc. VLDB Endow.* 14.13 (2021), pp. 3308–3321.

[LM10] Avinash Lakshman and Prashant Malik. “Cassandra: a Decentralized Structured Storage System”. In: *Operating Systems Review* 44.2 (2010), pp. 35–40.

[LL19] Geoff Langdale and Daniel Lemire. “Parsing Gigabytes of JSON per Second”. In: *VLDBJ* 28.6 (2019), pp. 941–960.

[LM00] N. Jesper Larsson and Alistair Moffat. “Off-line Dictionary-based Compression”. In: *Proc. IEEE* 88.11 (2000), pp. 1722–1732.

[Las+22] Robert Lasch et al. “Bandwidth-optimal Relational Joins on FPGAs”. In: *EDBT*. 2022, 1:27–1:39.

[LAC14] Maysam Lavasani, Hari Angepat, and Derek Chiou. “An FPGA-based In-Line Accelerator for Memcached”. In: *IEEE Comput. Archit. Lett.* 13.2 (2014), pp. 57–60.

[Lee+17] Jinho Lee et al. “ExtraV: Boosting Graph Processing Near Storage with a Coherent Accelerator”. In: *PVLDB* 10.12 (2017), pp. 1706–1717.

[Lee+12] Jinsoo Lee et al. “An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases”. In: *Proc. VLDB Endow.* 6.2 (2012), pp. 133–144.

[Lei+16] Guo-Qing Lei et al. “An FPGA Implementation for Solving the Large Single-Source-Shortest-Path Problem”. In: *IEEE Trans. on Circuits and Systems* 63-II.5 (2016), pp. 473–477.

[LRG15] Guoqing Lei, Li Rong-chun, and Song Guo. “TorusBFS: a Novel Message-passing Parallel Breadth-first Search Architecture on FPGAs”. In: *IRACST—ESTIJ*. Vol. 5. 2015.

[LK14] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection.* http://snap.stanford.edu/data. visited 10/2023. 2014.

[Li+17a] Bojie Li et al. “KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC”. In: *SOSP*. 2017, pp. 137–152.

[Li+17b] Yinan Li et al. “Mison: A Fast JSON Parser for Data Analytics”. In: *PVLDB* 10.10 (2017), pp. 1118–1129.

[Lia+16] Wei Liang et al. “Memory Efficient and High Performance Key-value Store on FPGA using Cuckoo Hashing”. In: *FPL*. 2016, pp. 1–4.

[Lin+08] Erik Lindholm et al. “NVIDIA Tesla: A Unified Graphics and Computing Architecture”. In: *IEEE Micro* 28.2 (2008), pp. 39–55.

[Liu+21] Chenhao Liu et al. “ScalaBFS: A Scalable BFS Accelerator on FPGA-HBM Platform”. In: *FPGA*. 2021, p. 147.

[Liu+19] Ke Liu et al. “Dynamically Reconfigurable Architecture for High-Throughput Hash Function in Key-Value Store”. In: *IEEE HPCC*. 2019, pp. 1964–1970.

[Loc20] John Lockwood. *Achieving a Half-Billion IOPs in a 1U Redis Server with FPGA Acceleration.* https://bit.ly/3GRQOWI. visited 10/2023. 2020.

[LM15] John W. Lockwood and Madhu Monga. “Implementing Ultra Low Latency Data Center Services with Programmable Logic”. In: *HOTI*. 2015, pp. 68–77.

[LH19] Jiaheng Lu and Irena Holubová. “Multi-model Databases: A New Journey to Handle the Variety of Data”. In: *ACM Comput. Surv.* 52.3 (2019), 55:1–55:38.

[Lum+07] Andrew Lumsdaine et al. “Challenges in Parallel Graph Processing”. In: *Parallel Processing Letters* 17.01 (2007), pp. 5–20.

[Lun+04] Jan Van Lunteren et al. “XML Accelerator Engine”. In: *International Workshop on High Performance XML Processing.* 2004.

[MZC17] Xiaoyu Ma, Dan Zhang, and Derek Chiou. “FPGA-Accelerated Transactional Execution of Graph Workloads”. In: *FPGA*. 2017, pp. 227–236.

[MBK02] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. “Generic Database Cost Models for Hierarchical Memory Systems”. In: *PVLDB*. 2002, pp. 191–202.

[Maw+21] Daniel Mawhirter et al. “GraphZero: A High-Performance Subgraph Matching System”. In: *ACM SIGOPS Oper. Syst. Rev.* 55.1 (2021), pp. 21–37.

[MS08] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox.* Springer, 2008.

[Mei+20] Guoqiang Mei et al. “A FPGA-based Intra-parallel Architecture for PageRank Graph Processing”. In: *EDGE*. 2020, pp. 31–38.

[MHH02] Oskar Mencer, Zhining Huang, and Lorenz Huelsbergen. “HAGAR: Efficient Multi-context Graph Processors”. In: *FPL*. 2002, pp. 915–924.

[MS19] Amine Mhedhbi and Semih Salihoglu. “Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins”. In: *PVLDB* 12.11 (2019), pp. 1692–1704.

[Mil+07] Emina I. Milovanovic et al. “Computing All-pairs Shortest Paths on a Linear Systolic Array and Hardware Realization on a Reprogrammable FPGA Platform”. In: *The Journal of Supercomputing* 40.1 (2007), pp. 49–66.

[Mit+09] Abhishek Mitra et al. “Boosting XML Filtering with a Scalable FPGA-based Architecture”. In: *CoRR* abs/0909.1781 (2009).

[MS00] Alistair Moffat and Lang Stuiver. “Binary Interpolative Coding for Effective Index Compression”. In: *Inf. Retr.* 3.1 (2000), pp. 25–47.

[Mog23] Mehdi Moghaddamfar. “Database System Acceleration on FPGAs”. PhD thesis. TU Dresden, 2023.

[Mog+23] Mehdi Moghaddamfar et al. “KeRRaS: Sort-Based Database Query Processing on Wide Tables Using FPGAs”. In: *DaMoN*. 2023, pp. 1–9.

[Mou+10] Roger Moussalli et al. “Accelerating XML Query Matching through Custom Stack Generation on FPGAs”. In: *HiPEAC*. 2010, pp. 141–155.

[Mou+11] Roger Moussalli et al. “Massively Parallel XML Twig Filtering using Dynamic Programming on FPGAs”. In: *ICDE*. 2011, pp. 948–959.

[MT09] René Müller and Jens Teubner. “FPGA: What’s in it for a Database?” In: *SIGMOD*. 2009, pp. 999–1004.

[NRR13] Hung Q. Ngo, Christopher Ré, and Atri Rudra. “Skew Strikes Back: New Developments in the Theory of Join Algorithms”. In: *SIGMOD Rec.* 42.4 (2013), pp. 5–16.

[Ngo+18] Hung Q. Ngo et al. “Worst-case Optimal Join Algorithms”. In: *J. ACM* 65.3 (2018), 16:1–16:40.

[Ni+14] Shi-Ce Ni et al. “Parallel Graph Traversal for FPGA”. In: *IEICE Electronic Express* 11.7 (2014), p. 20130987.

[NJ03] Matthias Nicola and Jasmi John. “XML Parsing: A Threat to Database Performance”. In: *CIKM*. 2003, pp. 175–178.

[Nur+14] Eriko Nurvitadhi et al. “GraphGen: An FPGA Framework for Vertex-Centric Graph Computation”. In: *FCCM*. 2014, pp. 25–28.

[OAA21] Francis O’Brien, Matthew Agostini, and Tarek S. Abdelrahman. “A Streaming Accelerator for Heterogeneous CPU-FPGA Processing of Graph Applications”. In: *IPDPS*. 2021, pp. 26–35.

[ONe+96] Patrick E. O’Neil et al. “The Log-Structured Merge-Tree (LSM-Tree)”. In: *Acta Informatica* 33.4 (1996), pp. 351–385.

[OO16] Tayo Oguntebi and Kunle Olukotun. “GraphOps: a Dataflow Library for Graph Analytics Acceleration”. In: *FPGA*. 2016, pp. 111–117.

[OBS99] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. “Berkeley DB”. In: *FREENIX*. 1999, pp. 183–191.

[Owa+17] Muhsen Owaida et al. “Centaur: a Framework for Hybrid CPU-FPGA Databases”. In: *FCCM*. 2017, pp. 211–218.

[Ozd+16] Muhammet Mustafa Ozdal et al. “Energy Efficient Architecture for Graph Analytics Accelerators”. In: *ISCA*. 2016, pp. 166–177.

[Pal+18] Shoumik Palkar et al. “Filter Before You Parse: Faster Analytics on Raw Data with Sparser”. In: *PVLDB* 11.11 (2018), pp. 1576–1589.

[PDH11] Kyprianos Papadimitriou, Apostolos Dollas, and Scott Hauck. “Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model”. In: *ACM Trans. Reconfigurable Technol. Syst.* 4.4 (2011), 36:1–36:24.

[PL18] Philippos Papaphilippou and Wayne Luk. “Accelerating Database Systems Using FPGAs: a Survey”. In: *FPL*. 2018, pp. 125–130.

[Pel+21] Johan Peltenburg et al. “Tens of Gigabytes per Second JSON-to-Arrow Conversion with FPGA Accelerators”. In: *(IC)FPT*. 2021, pp. 1–9.

[Pot19] Mohan Potheri. *Accelerating Virtualized Cassandra Databases with FPGA*. https://bit.ly/3gKwtIz. visited 10/2023. 2019.

[PCJ06] Natasa Przulj, Derek G. Corneil, and Igor Jurisica. “Efficient Estimation of Graphlet Frequency Distributions in Protein-protein Interaction Networks”. In: *Bioinform.* 22.8 (2006), pp. 974–980.

[Put+14] Andrew Putnam et al. “A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services”. In: *ISCA*. 2014, pp. 13–24.

[Qiu+20] Yunhui Qiu et al. “FULL-KV: Flexible and Ultra-Low-Latency In-Memory Key-Value Store System Design on CPU-FPGA”. In: *IEEE Transactions on Parallel and Distributed Systems* 31.8 (2020), pp. 1828–1444.

[Qiu+18] Yunhui Qiu et al. “Ultra-Low-Latency and Flexible In-memory Key-Value Store System Design on CPU-FPGA”. In: *FPT*. 2018, pp. 142–149.

[Rei+21] James Reinders et al. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL*. Springer Nature, 2021.

[Ren+19] Yuchen Ren et al. “A Low-Latency Multi-Version Key-Value Store Using B-Tree on an FPGA-CPU Platform”. In: *FPL*. 2019, pp. 321–325.

[Ren+17] Vincent Reniers et al. “On the State of NoSQL Benchmarks”. In: *International Conference on Performance Engineering*. 2017, pp. 107–112.

[RP71] Robert Rice and James Plaunt. “Adaptive Variable-length Coding for Efficient Compression of Spacecraft Television Data”. In: *IEEE Transactions on Communication Technology* 19.6 (1971), pp. 889–897.

[Rit17] Daniel Ritter. “Hardware-accelerated Application Integration: Challenges and Opportunities”. In: *Active Workshop at ACM Middleware*. 2017, p. 15.

[Rit+17] Daniel Ritter et al. “Hardware Accelerated Application Integration Processing: Industry Paper”. In: *DEBS*. 2017, pp. 215–226.

[RL17] Mehdi Roozmeh and Luciano Lavagno. “Implementation of a Performance Optimized Database Join Operation on FPGA-GPU Platforms Using OpenCL”. In: *IEEE NORCHIP*. 2017, pp. 1–6.

[RCJ11] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. “DRAMSim2: A Cycle Accurate Memory System Simulator”. In: *IEEE Comput. Archit. Lett.* 10.1 (2011), pp. 16–19.

[RA15] Ryan A. Rossi and Nesreen K. Ahmed. “The Network Data Repository with Interactive Graph Analytics and Visualization”. In: *AAAI*. visited 10/2023. 2015.

[Sah+20] Siddhartha Sahu et al. “The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey”. In: *VLDB J.* 29.2-3 (2020), pp. 595–618.

[SFB16] Juri Schmidt, Holger Fröning, and Ulrich Brüning. “Exploring Time and Energy for Complex Accesses to a Hybrid Memory Cube”. In: *MEMSYS*. 2016, pp. 142–150.

[SE22] SAP SE. *OrientDB Manual*. https://orientdb.com/docs/last/index.html. visited 10/2023. 2022.

[Sha+20] Zhiyuan Shao et al. “Processing Grid-format Real-world Graphs on DRAM-based FPGA Accelerators with Application-specific Caching Mechanisms”. In: *ACM Trans. Reconfigurable Technol. Syst.* 13.3 (2020), 11:1–11:33.

[Sha+19] Zhiyuan Shao et al. “Improving Performance of Graph Processing on FPGA-DRAM Platform by Two-level Vertex Caching”. In: *FPGA*. 2019, pp. 320–329.

[Shi+22] Runbin Shi et al. “Exploiting HBM on FPGAs for Data Processing”. In: *ACM Trans. Reconfigurable Technol. Syst.* 15.4 (2022), 36:1–36:27.

[Shi+18] Xuanhua Shi et al. “Graph Processing on GPUs: a Survey”. In: *ACM Comput. Surv.* 50.6 (2018), 81:1–81:35.

[Sid13] Reetinder Sidhu. “High Throughput, Tree Automata based XML Processing Using FPGAs”. In: *FPT*. 2013, pp. 74–81.

[Sie+19] Roberto Sierra et al. “High-Performance Decoding of Variable-Length Memory Data Packets for FPGA Stream Processing”. In: *FPL*. 2019, pp. 307–313.

[SSP06] Valery Sklyarov, Iouliia Skliarova, and Bruno Pimentel. “Modeling and FPGA-based Implementation of Graph Coloring Algorithms”. In: *ICARA*. 2006, pp. 443–448.

[Sni+06] Tom AB Snijders et al. “New Specifications for Exponential Random Graph Models”. In: *Sociological methodology* 36.1 (2006), pp. 99–153.

[Sto10] Michael Stonebraker. “SQL Databases v. NoSQL Databases”. In: *Commun. ACM* 53.4 (2010), pp. 10–11.

[SL20] Shixuan Sun and Qiong Luo. “In-Memory Subgraph Matching: An In-depth Study”. In: *SIGMOD*. 2020, pp. 1083–1098.

[Sun+20a] Shixuan Sun et al. “RapidMatch: A Holistic Approach to Subgraph Query Processing”. In: *PVLDB* 14.2 (2020), pp. 176–188.

[Sun+20b] Xuan Sun et al. “FPGA-based Compaction Engine for Accelerating LSM-tree Key-Value Stores”. In: *ICDE*. 2020, pp. 1261–1272.

[THK14] Yasuhiro Takei, Masanori Hariyama, and Michitaka Kameyama. “An SIMD Architecture for Shortest-Path Search and Its FPGA Implementation”. In: *PDPTA*. 2014.

[THK15] Yasuhiro Takei, Masanori Hariyama, and Michitaka Kameyama. “Evaluation of an FPGA-based Shortest-path-search Accelerator”. In: *PDPTA*. 2015, p. 613.

[TS18] Muhammad Usman Tariq and Fahad Saeed. “Parallel Sampling-Pipeline for Indefinite Stream of Heterogeneous Graphs using OpenCL for FPGAs”. In: *Big Data*. 2018, pp. 4752–4761.

[Tea22] Apache HBase Team. *Apache HBase Reference Guide*. https://hbase.apache.org/book.html. visited 10/2023. 2022.

[Teu17] Jens Teubner. “FPGAs for Data Processing: Current State”. In: *it Inf. Technol.* 59.3 (2017), p. 125.

[TW13] Jens Teubner and Louis Woods. *Data Processing on FPGAs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013.

[TWN12] Jens Teubner, Louis Woods, and Chongling Nie. “Skeleton Automata for FPGAs: Reconfiguring Without Reconstructing”. In: *SIGMOD*. 2012, pp. 229–240.

[TWN13] Jens Teubner, Louis Woods, and Chongling Nie. “*XLynx* - An FPGA-based XML Filter for Hybrid XQuery Processing”. In: *ACM Trans. Database Syst.* 38.4 (2013), 23:1–23:39.

[THZ20] James J. Thomas, Pat Hanrahan, and Matei Zaharia. “Fleet: A Framework for Massively Parallel Streaming on FPGAs”. In: *ASPLOS*. 2020, pp. 639–651.

[TS21] Neil C. Thompson and Svenja Spanuth. “The Decline of Computers as a General Purpose Technology”. In: *Commun. ACM* 64.3 (2021), pp. 64–72.

[TZP15] Da Tong, Shijie Zhou, and Viktor K. Prasanna. “High-Throughput Online Hash Table on FPGA”. In: *IPDPS*. 2015, pp. 105–112.

[Ull76] Julian R. Ullmann. “An Algorithm for Subgraph Isomorphism”. In: *J. ACM* 23.1 (1976), pp. 31–42.

[UMJ15] Yaman Umuroglu, Donn Morrison, and Magnus Jahre. “Hybrid Breadth-first Search on a Single-chip FPGA-CPU Heterogeneous Platform”. In: *FPL*. 2015, pp. 1–8.

[Umu+17] Yaman Umuroglu et al. “FINN: a Framework for Fast, Scalable Binarized Neural Network Inference”. In: *FPGA*. 2017, pp. 65–74.

[Vai+18] Anuj Vaishnav et al. “Resource Elastic Virtualization for FPGAs Using OpenCL”. In: *FPL*. 2018, pp. 111–118.

[VAM09] Wim Vanderbauwhede, Leif Azzopardi, and Mahmoud Moadeli. “FPGA-accelerated Information Retrieval: High-efficiency Document Filtering”. In: *FPL*. 2009, pp. 417–422.

[Van+13] Wim Vanderbauwhede et al. “A Hybrid CPU-FPGA System for High Throughput (10Gb/s) Streaming Document Classification”. In: *SIGARCH Computer Architecture News* 41.5 (2013), pp. 53–58.

[Vel12] Todd L. Veldhuizen. “Leapfrog Triejoin: a worst-case optimal join algorithm”. In: *CoRR* abs/1210.0481 (2012).

[Vog+23] Lukas Vogel et al. “Data Pipes: Declarative Control over Data Movement”. In: *CIDR*. 2023.

[Wan+10] Qingbo Wang et al. “A Message-passing Multi-softcore Architecture on FPGA for Breadth-first Search”. In: *FPT*. 2010, pp. 70–77.

[Wan+20a] Qinggang Wang et al. “A Conflict-free Scheduler for High-performance Graph Processing on Multi-pipeline FPGAs”. In: *ACM Trans. Archit. Code Optim.* 17.2 (2020), 14:1–14:26.

[Wan+21] Qinggang Wang et al. “GraSU: A Fast Graph Update Library for FPGA-based Dynamic Graph Processing”. In: *FPGA*. 2021, pp. 149–159.

[WZH16] Xu Wang, Yongxin Zhu, and Linan Huang. “A Comprehensive Reconfigurable Computing Approach to Memory Wall Problem of Large Graph Computation”. In: *J. Syst. Archit.* 70 (2016), pp. 59–69.

[Wan+15] Xu Wang et al. “Addressing Memory Wall Problem of Graph Computation in Reconfigurable System”. In: *HPCC*. 2015, pp. 302–307.

[WHN19] Yu Wang, James C. Hoe, and Eriko Nurvitadhi. “Processor Assisted Worklist Scheduling for FPGA Accelerated Graph Processing on a Shared-Memory Platform”. In: *FCCM*. 2019, pp. 136–144.

[Wan+20b] Zeke Wang et al. “Shuhai: Benchmarking High Bandwidth Memory On FPGAS”. In: *FCCM*. 2020, pp. 111–119.

[WNH13] Gabriel Weisz, Eriko Nurvitadhi, and J Hoe. “GraphGen for CoRAM: Graph Computation on FPGAs”. In: *Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*. 2013.

[Wer+17] Stefan Werner et al. “Semi-static Operator Graphs for Accelerated Query Execution on FPGAs”. In: *Microprocessors and Microsystems - Embedded Hardware Design* 53 (2017), pp. 178–189.

[Wha+15] Joyce Jiyoung Whang et al. “Scalable Data-Driven PageRank: Algorithms, System Issues, and Lessons Learned”. In: *Euro-Par*. Vol. 9233. LNCS. 2015, pp. 438–450.

[Wil17] Bruce Wile. *Accelerating Neo4j with CAPI SNAP from IBM Power Systems*. https://bit.ly/3gNyCmK. visited 10/2023. 2017.

[WIA14] Louis Woods, Zsolt István, and Gustavo Alonso. “Ibex - An Intelligent Storage Engine with Support for Advanced SQL Off-loading”. In: *PVLDB* 7.11 (2014), pp. 963–974.

[Xie+19] Jinyu Xie et al. “High-Throughput and Low-Latency Distributed Management Proxy for Key-Value Store Over 100Gbps Ethernet on FPGA”. In: *FPT*. 2019, pp. 224–230.

[Xu+17] Chongchong Xu et al. “Evaluation and Trade-offs of Graph Processing for Cloud Services”. In: *IEEE ICWS*. 2017, pp. 420–427.

[Xu+18] Chongchong Xu et al. “Domino: Graph Processing Services on Energy-Efficient Hardware Accelerator”. In: *ICWS*. 2018, pp. 274–281.

[Xu+16] Shuotao Xu et al. “BlueCache: a Scalable Distributed Flash-based Key-value Store”. In: *PVLDB* 10.4 (2016), pp. 301–312.

[Yan+19] Mingyu Yan et al. “Balancing Memory Accesses for Energy-Efficient Graph Analytics Accelerators”. In: *ISLPED*. 2019, pp. 1–6.

[Yan+17] Chen Yang et al. “OpenCL for HPC with FPGAs: Case Study in Molecular Electrostatics”. In: *HPEC*. 2017, pp. 1–8.

[Yan+20a] Chengbo Yang et al. “Efficient FPGA-based Graph Processing with Hybrid Pull-push Computational Model”. In: *Frontiers Comput. Sci.* 14.4 (2020), p. 144102.

[Yan+21] Haichang Yang et al. “HeteroKV: A Scalable Line-rate Key-Value Store on Heterogeneous CPU-FPGA Platforms”. In: *DATE*. 2021, pp. 834–837.

[YKP20] Yang Yang, Sanmukh R. Kuppannagari, and Viktor K. Prasanna. “A High Throughput Parallel Hash Table Accelerator on HBM-enabled FPGAs”. In: *FPT*. 2020, pp. 148–153.

[Yan+20b] Yang Yang et al. “FASTHash: FPGA-Based High Throughput Parallel Hash Table”. In: *Supercomputer*. Vol. 12151. Lecture Notes in Computer Science. 2020, pp. 3–22.

[Yao+18] Pengcheng Yao et al. "An Efficient Graph Accelerator with Parallel Data Conflict Management". In: *PACT*. 2018, 8:1–8:12.

[Yip+15] Milo Yip et al. *RapidJSON*. https://rapidjson.org. visited 10/2023. 2015.

[Zha+18] Chao Zhang et al. "UniBench: A Benchmark for Multi-model Database Management Systems". In: *TPC Technology Conference*. 2018, pp. 7–23.

[Zha+20a] Fan Zhang et al. "An Effective 2-Dimension Graph Partitioning for Work Stealing Assisted Graph Processing on Multi-FPGAs". In: *IEEE Transactions on Big Data* (2020).

[ZKL17] Jialiang Zhang, Soroosh Khoram, and Jing Li. "Boosting the Performance of FPGA-based Graph Processor using Hybrid Memory Cube: a Case for Breadth First Search". In: *FPGA*. 2017, pp. 207–216.

[ZL18] Jialiang Zhang and Jing Li. "Degree-aware Hybrid Graph Traversal on FPGA-HMC Platform". In: *FPGA*. 2018, pp. 229–238.

[Zha+20b] Ruizhi Zhang et al. "A High Throughput Parallel Hash Table on FPGA using XOR-based Memory". In: *HPEC*. 2020, pp. 1–7.

[Zha+20c] Teng Zhang et al. "FPGA-Accelerated Compactions for LSM-based Key-Value Store". In: *FAST*. 2020, pp. 225–237.

[ZCP15a] Shijie Zhou, Charalampos, and Viktor K. Prasanna. "Accelerating Large-Scale Single-Source Shortest Path on FPGA". In: *IPDPS*. 2015, pp. 129–136.

[ZCP15b] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. "Optimizing Memory Performance for FPGA Implementation of PageRank". In: *ReConfig*. 2015, pp. 1–6.

[ZCP16] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. "High-Throughput and Energy-Efficient Graph Processing on FPGA". In: *FCCM*. 2016, pp. 103–110.

[ZP17] Shijie Zhou and Viktor K. Prasanna. "Accelerating Graph Analytics on CPU-FPGA Heterogeneous Platform". In: *29th International Symposium on Computer Architecture and High Performance Computing*. 2017, pp. 137–144.

[Zho+18] Shijie Zhou et al. "An FPGA Framework for Edge-centric Graph Processing". In: *Proceedings of the 15th ACM International Conference on Computing Frontiers*. 2018, pp. 69–77.

[Zho+19] Shijie Zhou et al. "HitGraph: High-throughput Graph Processing Framework on FPGA". In: *IEEE Trans. Parallel Distrib. Syst.* 30.10 (2019), pp. 2249–2264.

[ZHC15] Xiaowei Zhu, Wentao Han, and Wenguang Chen. “GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning”. In: *USENIX ATC*. 2015, pp. 375–386.

[Zie+16] Daniel Ziener et al. “FPGA-Based Dynamically Reconfigurable SQL Query Processing”. In: *TRETS* 9.4 (2016), 25:1–25:24.

[Zuk+06] Marcin Zukowski et al. “Super-Scalar RAM-CPU Cache Compression”. In: *ICDE*. 2006, p. 59.