Database Acceleration on FPGAs

Fang, Jian

DOI
10.4233/uuid:84dfc577-ca6f-43ea-9b24-4dc160c103f5

Publication date
2019

Document Version
Final published version

Citation (APA)
Fang, J. (2019). *Database Acceleration on FPGAs*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:84dfc577-ca6f-43ea-9b24-4dc160c103f5

Database Acceleration on FPGAs

DATABASE ACCELERATION ON FPGAS

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus prof.dr.ir. T.H.J.J. van der Hagen
chair of the Board for Doctorates
to be defended publicly on
Tuesday 10 December 2019 at 10:00 o'clock

by

Jian FANG

Master of Engineering in Computer Science & Technology,
National University of Defense Technology, Hunan, China,
born in Huizhou, Guangdong, China.

This dissertation has been approved by the promotors:

Prof.dr. H.P. Hofstee
Dr.ir. Z. Al-Ars

Composition of the doctoral committee:

Rector Magnificus,	chairman
Prof.dr. H.P. Hofstee,	Delft University of Technology, promotor
Dr.ir. Z. Al-Ars,	Delft University of Technology, promotor
Independent members:	
Prof.dr. J. Teubner	TU Dortmund
Prof.dr.ir. S. Hamdioui	Delft University of Technology
Prof.dr.ir. K.L.M. Bertels	Delft University of Technology
Prof.dr.ir. D.H.J. Epema	Delft University of Technology
Other member:	
Dr. J. Hidders	vml TU Delft

ISBN 978-94-028-1868-0
This research was financially supported by China Scholarship Council (CSC)
SIKS Dissertation Series No. 2019-37
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Keywords: Database, FPGA, Acceleration, Decompression, Join

Printed by: Ipskamp Printing, the Netherlands

Front & Back: Tiantian Du (TU Delft), resources from: www.flaticon.com

Dedicated to my country and my family

SUMMARY

Though field-programmable gate arrays (FPGAs) have been used to accelerate database systems, they have not been widely adopted for the following reasons. As databases have transitioned to higher bandwidth technology such as in-memory and NVMe, the communication overhead associated with accelerators has become more of a burden. Also, FPGAs are more difficult to program, and GPUs have emerged as an alternative technology with better programming support. However, with the development of new interconnect technology, memory technology, and improved FPGA design tool chains, FPGAs again provide significant opportunities. Therefore, we believe that FPGAs can be attractive again in the database field.

This thesis focuses on FPGAs as a high-performance compute platform, and explores using FPGAs to accelerate database systems. It investigates the current challenges that have held FPGAs back in the database field as well as the opportunities resulting from recent technology developments. The investigation illustrates that FPGAs can provide significant advantages for integration in database systems. However, to make further progress, studies in a number of areas, including new database architectures, new types of accelerators, deep performance analysis, and the development of the tool chains are required. Our contributions focus on accelerators for databases implemented in reconfigurable logic. We provide an overview of prior work and make contributions to two specific types of accelerators: both a compute-intensive (decompression) and a memory-intensive (hash join) accelerator.

For the decompression, we propose a “refine” technique and a “recycle” technique to achieve high single-decompressor throughput by keeping only a single copy of the history data in the internal block RAM (BRAM) memory of the FPGA, and operating on each BRAM independently. We apply these two techniques to Snappy, a widely used decompression algorithm in big data and database applications. The experimental results show that the proposed Snappy decompressor achieves up to 7.2 GB/s throughput per decompressor, which presents a significant speedup compared to the software implementation. One such decompressor can easily keep pace with a non-volatile memory express (NVMe) device (PCIe Gen3 x4) on a small FPGA. We also propose a Parquet-to-Arrow converter on FPGAs to improve the efficiency of reading an Apache Parquet file from the storage into the main memory presented in the Apache Arrow format.

For hash joins, we first analyze the impact factors for hash join algorithms, and point out that the granularity factor can significantly influence the throughput. Then, we build a performance model based on these impact factors that considers both the computation and data transfer. Our model can accurately predict the best performing designs be-

tween no-partitioning hash join and partitioning hash join. Adopting this performance model, we study a no-partitioning hash join and a radix partitioning hash join algorithm, and conclude that no-partitioning hash join should be more competitive than the partitioning hash join when the tuple size is large and the granularity is small. Then we focus on FPGA acceleration of hash joins, where we study the performance effect of adding HBMs to the FPGA. We conclude that FPGAs with HBMs can improve the hash join throughput, but requires resolving the challenge that random accesses to HBMs suffer obvious performance drop, especially for the cases where the requests need to cross different channels. To solve this problem, we present a hash join accelerator that stores the hash table in the HBMs. In the proposed architecture, all the HBM channels can operate independently. A pre-partition method is presented to drive the HBM traffic to the appropriate channels, in order to reduce the traffic contention. The proposed method should efficiently utilize the HBM bandwidth, and connecting the proposed hash join engine to a host memory can process the data with a throughput that is only limited by the host-to-accelerator interface bandwidth.

SAMENVATTING

Hoewel FPGAs (Field-Programmable Gate-Arrays) zijn gebruikt om databasesystemen te versnellen, zijn ze om de volgende redenen niet breed toegepast. Met de overgang naar hogere bandbreedte technologie zoals in-memory en NVMe, is de communicatieoverhead geassocieerd met versnellers zoals FPGAs meer een last geworden. FPGAs zijn ook moeilijker te programmeren en GPUs zijn naar voren gekomen als een alternatieve technologie met betere programmeerondersteuning. Met de ontwikkeling van nieuwe interconnecttechnologie, betere geheugentechnologie en verbeterde FPGA-ontwerptoolketens bieden FPGAs echter opnieuw interessante mogelijkheden. Wij zijn van mening dat FPGAs weer aantrekkelijk kunnen zijn in het databaseveld.

Dit proefschrift richt zich op FPGAs als een krachtig rekenplatform en onderzoekt het gebruik van FPGAs om database systemen te versnellen. Het onderzoekt de huidige uitdagingen die het gebruik van FPGAs op het gebied van databases hebben tegengehouden, evenals de kansen die voortvloeien uit recente technologische ontwikkelingen. Het onderzoek illustreert dat FPGAs voordelen kunnen bieden voor integratie in databasesystemen. Om verdere vooruitgang te boeken, zijn echter studies op een aantal gebieden, waaronder nieuwe database-architecturen, nieuwe ontwerpen van versnellers, een diepgaande analyse van de prestaties en de ontwikkeling van betere toolketens vereist. Onze bijdragen zijn gericht op versnellers voor databases geïmplementeerd in herconfigureerbare logica. We bieden een overzicht van eerder werk en leveren bijdragen aan twee specifieke typen versnellers: zowel een rekenintensieve (decompressie) als een geheugenintensieve (hash join) versneller.

Voor de decompressie stellen we een “verfijnings techniek” en een “recycle techniek” voor om een hoge single-decompressor doorvoer te bereiken door slechts een enkele kopie van de decompressie historie te bewaren in interne FPGA block RAM (BRAM) geheugen en onafhankelijk op iedere BRAM te opereren. We passen deze twee technieken toe op Snappy, een veelgebruikt decompressie-algoritme in big data- en database-applicaties. Experimentele resultaten laten zien dat de voorgestelde Snappy-decompressor tot 7,2 GB/s doorvoer per decompressor behaalt, wat een aanzienlijke versnelling ten opzichte van de software-implementatie oplevert. Eén decompressor kan gemakkelijk gelijke tred houden met een non-volatile memory express (NVMe)-opslagelement (PCIe Gen3 x4) op een kleine FPGA. We stellen ook een Parquet-naar-Arrow converter voor in FPGAs om de efficiëntie van het lezen van een Apache Paquet bestand naar de opslag in het hoofdgeheugen, gepresenteerd in het Apache Arrow-formaat, te verbeteren.

Voor hash joins analyseren we eerst de impactfactoren voor de hash-join-algoritmen en wijzen we erop dat de granulariteitsfactor de doorvoer aanzienlijk kan beïnvloeden. Ver-

volgens bouwen we een prestatiemodel op basis van deze impactfactoren die rekening houden met zowel de berekening als de gegevensovergang. Ons model kan de best presterende ontwerpen tussen "no-partitioning"hash join en "partitioning"hash join nauwkeurig voorspellen. Door gebruik te maken van dit prestatiemodel bestuderen we een no-partitioning hash-join en een radix-partitioning hash-join-algoritme en concluderen we dat no-partitioning-hash-join beter zou moeten presteren dan de partitioning-hash-join wanneer de tuple-grootte groot is en de ganulariteit klein is. Wat betreft de FPGA-versnelling op hash joins bestuderen we het effect van het toevoegen van HBMs (High Bandwidth Memories) aan de FPGA. We concluderen dat FPGAs met HBMs de hash join-doorvoersnelheid kunnen verbeteren, maar het vereist het oplossen van de uitdaging dat de willekeurige toegangen tot HBMs een duidelijke prestatieverlies vertonen, vooral voor de gevallen waarin verschillende toegangs kanalen moeten worden overschreden. Om dit probleem op te lossen presenteren we een hash join-versneller die de hashtabel opslaat in de HBMs. In de voorgestelde architectuur kunnen alle HBM-kanalen onafhankelijk werken. We presenteren een pre-partitiemethode om het HBM-verkeer naar de juiste kanalen te sturen, om de toegangsconflicten te verminderen. De voorgestelde methode moet de HBM-bandbreedte efficiënt gebruiken en het verbinden van de voorgestelde hash-join-engine met een host-geheugen kan de gegevens verwerken met een doorvoer die wordt beperkt door de host-naar-versneller interface bandbreedte.

ACKNOWLEDGMENTS

As I reach the finish line of this long PhD journey, I can spend the time to reflect and recall how it started. My earliest memories go back to September 16th, 2014. That was the first day I arrived in the Netherlands. It was a cool morning. The sun was just rising and was ready to send its first light to welcome me. At that moment, I recognized that a new journey has just started for me, and this journey would be long but full of joys and sorrows. I believe that whenever I recall every little event that happened during these years, I will be much pleased. Honestly speaking, without the help of those around me, sometimes I even doubted that I can reach the destination. Therefore, I would like to take this opportunity to show my great gratitude to all of those who helped me, supported me, and encouraged me. Without you, I wouldn't have been able to continue my journey to the end.

First, I want to express my deepest gratitude to my promotor Prof. **H. Peter Hofstee**. It is my honor to be your first PhD student in Delft. Thank you for all your help during these years. Not only are you my promotor, but you are also my English teacher, my vocational trainer, my psychotherapist, and most importantly, my friend. You taught me how to read a paper, how to do research, how to present our work, how to talk with people, how to be confident, and even how to relax. You have always been patient. I can remember that you spent an hour to explain to me how powerful HLS would be if OpenMP is supported, and then spent another hour rephrasing your explanation to me when I said "I don't understand". You always encourage me. My English has now improved from "You might want to take some English course which can help you a lot" to "Good, I am fine with the content, and I will do some small editing", and then to "Did you use Grammarly for the writing? It looks quite good", and to now "I like this sentence a lot". Even though I know my English has a long way to go, I am not worried to write and talk English any more. You always helped me and supported me, no matter where you were and what time it was. If you can still remember how we met the VLDB Journal deadline, it perfectly illustrates this. On that day, you needed to fly from Austin to LA, and then transferred to Hong Kong, and finally to India. You revised the paper on the flight, and sent me a version whenever you touched the ground, and even talked to me through Skype. When you sent me the last version in India, you told me you'd like to sleep for a while, and we can talk two hours later. I calculate the time in India, and it was 3am in the morning. Your voice was tired, but every word you said was so clear. I can remember the day when I did a presentation in Denver, you flew from Austin to Denver and arrived at the conference room at 10am, joining my talk. But you left right after I finished my talk and flew back to Austin again. I even didn't get a chance to say thank you. You told me that you know many professors support their students like this, but I want to tell you that, you are the only one I know and I hope one day I will also be such a professor. I'd still like to apologize to you that the year of 2018 was the only year that

you had the entire Christmas holiday for yourself since you helped me with my work in all other holidays. When we corrected the master theses for Kangli, Xianwei, and Qiao during the Christmas holiday 2017, it even took up your New year holiday. I remember that you only used one day to send them feedback as soon as they gave you their draft. But I knew, you must do a pass for all three theses in your 10+ hour flight. In the next few days, we worked together from very early in the morning till very late at night. Though we were very tired, we were thrilled to see three excellent master theses. There are so many details that I'd almost forget to thank you for, like your quick email response (I don't have any memory that your email response has exceeded 24 hours). I could write a book to remember every second and to say thank you. My friends always say "You must have saved the galaxy in your previous incarnation to have such a professor supervising you". But I always answer jokingly: "No, Peter must have destroyed the Galaxy in his previous incarnation to have a PhD student like me in this life". Once again, thank you.

I would like to express sincere appreciation to Dr. **Zaid Al-Ars**, who is also my promotor. Thank you for bringing me to the Accelerated Big Data Systems (ABS) group. You spent so much time to discuss with me and help me in my project. You taught me how to manage a project and keep it in the right track. You gave me a chance to set up a small team and teach me how to work as a team leader, how to manage the sub-projects, how to work with people, who to communicate with people, how to get funding, etc. I believe that my research career will benefit from all these instructions you gave me.

Dr. **Jan Hidders**, thank you. Thank you for being my supervisor for the first three years. You helped me a lot in my research, especially in the database field. **Jinho Lee**, thank you for participating in my project. You always gave helpful suggestions. I hope you can have a successful career in Korea.

It is my honor to have Prof. **Jens Teubner**, Prof. **Koen Bertels**, Prof. **Said Hamdioui**, and Prof. **Dick Epema** as my committee members. Thank you for spending time to review my thesis and giving many helpful suggestions.

My supervisors in China, Prof. **Yuhua Tang**, Prof. **Weixia Xu**, Prof. **Yutong Lu**, Prof. **Qiong Li**, Dr. **Zhenlong Song**, and Dr. **Dengping Wei**, thank you for your help and support during my Bachelor and Master studies. Without you, I can hardly start a research life.

Thanks to all ABS group members, **Johan**, **Matthijs**, **Joost**, **Joroen**, **Tanveer**, **Baozhou**, **Edwin**, **Lars L** and **Lars W**. I was very happy to discuss with you and have beer afterwards. I miss the BBQs at the roof of Johan's house and the marinated ribs.

I would like to extend my thanks to all the members in the QCE department. Thanks to **Stephan Wong Qi Guo**, **Sensen Hu**, **Shanshan Ren**, **Ahn**, **Motta Razvan**, **Leon**, **Imran**, **Carmina**, **Mengyu**, **Tom**, **Hamid**, **Nauman**, **Hani**, I really enjoyed working and having fun with you guys. Special thanks to **Lidwina**, **Joyce**, and **Erik**, you gave us a wonderful working environment.

Furthermore, it was my honor to supervise and work with you for your master thesis projects and internships, **Kangli Huang**, **Xianwei Zeng**, **Yang Qiao**, **Bastiaan Feenstra**, **Jianyu Chen**, and to collaborate with **Yvo Mulder** and **Las van Leeuwen** for your master projects. I learned a lot from you about how to work in a team, how to lead a project, and how to be a good teacher. You, your graduations and your bright future are the best gifts for my PhD life.

Thank you, **Jianbin Fang**, **Siqi Shen**, **Yong Guo**, **Jie Shen**, **Changlin Chen**, **Yao Wang**, **Chang Wang**, **Ke Tao**; thanks for guiding me and helping me, especially the first month I was in Delft. **Guangming Li**, **Yazhou Yang**, **Sihang Qiu**, **Hai Zhu**, **Renfei Bu**, **Xiaohui Wang**, **Xinyi Li**, **Xu Huang**, **Xu Xie**, **Laobin Zhang**, **Yunlong Li**, thanks for the events; I would like to have karting, bowling, skating, and BBQ one more time.

Yue Zhao (越越), thanks for being my classmate and roommate for more than ten years, wish you have a happy life with **Xing Li** (星星仔). **Xiang Fu** (院士), I am very lucky to have you in the same group in both master and PhD. You always supported and encouraged me when I felt hopeless, best wishes to you and **Zhaokun Guo** (坤坤). **Yu Xin** (鱼) and **Zhijie Ren** (师兄), wish our friendship can be as solid as a rock, please keep the rock (砖头) we gave you. **Zixuan Zheng** (子轩) and **Lingling Lao** (玲玲), thanks for dinners, games, trips. I miss the wine and the chat in your studio.

Many thanks to my roommates. **Shengzhi Xu** (圣志), it is thankful to live in the same apartment with you for the last half a year. **Jianping Wang** (建平), wish you have a successful career in academia. **Jiapeng Yin** (加鹏) and **Yuan Li** (媛姐), I like to talk with you; you always have good ideas. **Zishun Liu** (子瞬) and **Dan Cheng** (成丹), **Yanze Yang** (晏泽), I had delightful times living with you, especially the New Year Eve.

Big thanks to **Lei Xie** (解磊), **Jintao Yu** (锦涛), **Shi Xu** (徐实), **Yande Jiang** (艳德) and **Na Chen** (娜娜), **He Wang** (王贺), **Lizhou Wu** (立舟), **Baozhou Zhu** (保周), thanks for all the good memories in the office and out of campus. Thank you, **Xin Guo** (郭昕), for all the cakes and snacks you made for us. I really enjoyed baking together with you in our party house. Also, thanks to **Rong Zhang** (张荣) for joining the party house, and the same flight when we were heading to the Netherlands. I want to express my gratefulness to Prof. **Weidong Jiang** (姜老师) and Prof. **Tian Jin** (金老师), I enjoyed the time we had drinks together, went out together, and I still can remember that you taught us how to appreciate antiques, how to taste wine, how to sip tea, and how to enjoy life. Thanks to my friend in Maastricht and also my middle school classmate **Ning An** (安宁), thanks for sharing your store and helping me in my hard times. I wish you can also get your PhD degree soon. I want to say thank you to **Yang Qu** (曲阳), **Tao Lv** (涛哥), **Guangliang Chen** (冠良), **Mingjuan Zhao** (明娟), I had good experience swimming with you. I had the warmest memories of my friends, **Jiefang Ma** (洁芳), **Lin Jia** (贾玲), **Tingyu Zeng** (曾哥), **Xichen Shun** (学姐), **Shengzhi Xu** (圣志), thanks for the days and nights you spent with me, together protecting our home and eating chicken afterwards. The last sentence is reserved to Dalaos (大佬们), **Qiang Liu** (老刘), **Xin Du** (小杜), **Xiang Fu** (院士), **Yue Zhao** (越越). I cannot forget how we encourage each other (商业互吹) in the

BigFish weChat group (大佬群). I enjoyed the time when we discuss research (吹牛皮) and potential business (还是吹牛皮).

A thousand thanks to my dear friends **Shuai Yuan** (袁帅), **Zhi Hong** (洪智), **Hai Gong** (龚海), **Xun Gong** (龚勋), **Qingqing Ye** (叶青青), **Bo Wang** (小虫), **Zhidong He** (之栋), **Xiangrong Wang** (向荣), **Jiefang Ma** (洁芳), **Mei Liu** (刘美), **Jiani Liu** (刘佳妮), **Lin Jia** (贾玲), **Meng Meng** (孟梦), **Juan Yan** (严娟), **Pengling Wang** (王鹏玲), **Peiyao Luo** (佩瑶), **Jianpeng Zhang** (建朋); thanks for your company and invitation. I want to extend this appreciation to all the friends I met in the Netherlands .. thank you. I cannot list all your names, but without you I cannot imagine how I could have endured the boring evenings and weekends.

I also want to say thank you to my friends in the Center for Quantum Computing in PengCheng Laboratory in Shenzhen, especially to **Yuan Feng**, **Kejia Zhang**, **Yuxin Deng**, **Hua Wu**, **Hanru Jiang**, **Peng Zhou**, **Xi Deng**, **Jinrong Zhang**, **Fucheng Cheng**, **Yonglong Ding**, **Chunchao Hu**, **Pan Gao**, **Xinxin Li**. I am a layman in quantum computing, but all of you are very nice and patient to introduce this field to me. I learned a lot from you. Thank you. In addition, I am glad to discuss with you, chat with you, and have dinners with you.

I would also take this opportunity to thank you, my teacher and friends in middle school and high school, particularly to **Youhua Ye** (花姐), **Haiping Long**, **Su zeng**, **Zhixiang Ke**, **Jianmin Ye** (JM), **Lifeng Xu** (行长), **Huicong Chen** (啊赤), **Dinghao Chen** (豪哥), **Zijian Yu** (精哥), **Kai Xie** (色), **Jiancheng Lu** (成哥), **Haoxuan Cai** (车干), **Jialun Gu** (古董). Thanks for your encouragement at all times and sharing your ideas, ideals and struggling experiences to me.

Tiantian, it is a good time to say thank you. Thanks for your company, encouragement, support, patience, and everything you did for me. Every second with you is a precious memory for me. If you ask me what is the biggest regret with you is, I would say “I haven’t met you earlier".

Last but not least, I owe a big thank to my parents for all the selfless love and all the support you gave me. There are no words that can fully express my gratitude towards you. I never said this before, but today, I want to say **Dad**, **Mom**, I love you (**爸爸，妈妈，我爱你!**).

Jian Fang
27-11-2019
Changsha, China

CONTENTS

1

INTRODUCTION

SUMMARY

Databases have now largely transitioned from hard-disk-drive-based (HDD-based) storage to much higher-bandwidth technologies such as in-memory and NVMe (non-volatile memory express), which causes database-related operations that used to be communication bound to now be computation bound. CPUs do not improve at a fast enough speed to keep pace with the computational requirements of database processing, and this demands new solutions. Recently, field programmable gate arrays (FPGAs) have proven to be successful accelerators in a number of fields such as security, machine learning, and high performance computing. In addition, new developments of interconnect technology, memory technology, and improved FPGA design tool chains set the stage for FPGAs to provide significant performance improvement for database operations. Therefore, we believe that FPGAs have great potential to accelerate in-memory database applications. The work in this thesis aims to identify a number of these applications and to show the advantage of FPGAs to accelerate their computation. This chapter of the thesis presents the motivation for the work, discusses the challenges in this field and lists the contributions in this thesis to address these challenges.

1

1.1. BACKGROUND

This section provides an overview of the field of the research discussed in this thesis. It also briefly introduces the background and related basic knowledge needed to understand the context of the work. More details are provided in Chapter 2.

1.1.1. DATABASE SYSTEMS BACKGROUND

INTRODUCTION TO DATABASE SYSTEM

A database is a collection of data that is organized in a way for easy accessing, managing, and processing. The data can be organized in different forms such as tables, graphs, documents, etc. Database management systems (DBMS) are systems that interact with the applications or users and the databases, performing data management and data analysis. We call it database system for short in the remainder of this thesis. The study in [1] presents a comprehensive introduction to the architecture of the DBMS. In this thesis, we explain it briefly. As shown in Fig. 1.1, a database system typically contains the following four components: the process manager, the query manager, the storage manager, and the shared utilities.

The process manager is responsible for making decisions for the execution of concurrent user requests, as well as mapping the requests to processes or threads in the operating system. Once the request is authorized to execute the query, the query manager takes care of the query execution. The job of the storage manager is to control the data fetching and updates to the storage for disk-based databases or the main memory for in-memory databases. The shared utilities are a set of components that not all requests need to touch, but can provide addition functionality such as memory allocation, catalog management, and replication services, etc. This thesis focuses on the query execution which is conducted in the query manager.

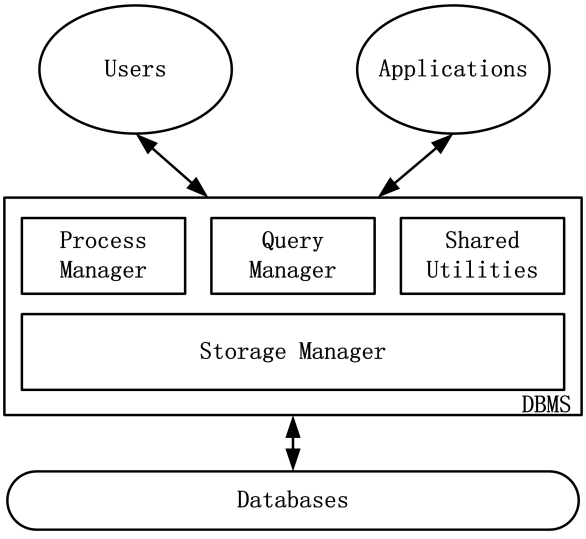


Figure 1.1: Structure of Database Systems

There are four main components in the query manager (Fig. 1.2): the parser, the query rewrite, the optimizer, and the executor. Once a query is received, the parser first checks whether the query is correctly specified, and converts it into an internal format that can be used for the next steps. Then, the query rewrite will further simplify and normalize the query, and will output an internal representation of the query. After that, the optimizer takes this internal representation as input, and generates an efficient query plan that contains a collection of operations for executing the query. The execution of the query is conducted in the executor by fully executing the query plan. For large data sets, executing the database operations significantly impacts the performance of the query execution. Thus, it is important to study how to improve the performance of executing the database operations. There are different types of database operations. The basic ones are selections, projections, arithmetic, aggregation, sort, joins, compression and decompression. Different operations have different features such as different memory access patterns, and different computation and memory access requirements. The study in [2] discusses and covers the most frequently used operations. In this thesis, we study two of the most time-consuming operations, including one compute-intensive operation and one memory-intensive operation. They are the decompression and the hash join, whose performance significantly impacts the performance of the database system. We present more details about these two operations in the rest of this section, and present further study in Chapter 3, 4, and 5

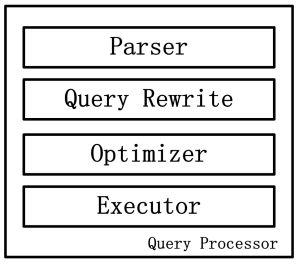


Figure 1.2: Internal Components in the Query Manager

SNAPPY (DE)COMPRESSION

Compression and decompression is one of the frequently used operations in database applications. It reduces the data amount that needs to be transferred through network or between the processor and the storage. The basic idea is to find repeated information and use a smaller piece of data to represent it. A simple but widely used compression algorithm is the run length encoding (RLE). It uses a pair that represents the repeated character and the number of repeats to replace the repeated characters in a sequence. For example, a data sequence of “RRRRRRRLLLLEE" after RLE encoding becomes “7R4L2E". Some other compression algorithms work on the word level instead of the character level. A well-known example of the word level compression is the Lempel Ziv 77 (LZ77) series [3] compression algorithm. In this class of compression algorithms, the repeated byte sequence is converted to a pair of a back reference and a length, where the back reference indicates where the previous sequence occurs and the length stands

1

for how long this sequence is. If a sequence is not found to be replicated, the original data is kept and no reference-length pair is used to replace this sequence. Generally, the reference-length pairs are called copy tokens, while non-repeated sequences are referred to as literal tokens. This thesis focuses on LZ77-based compression algorithms and chooses Snappy [4] as an example for further studies.

Snappy is an LZ77-based, byte-level (de)compression algorithm, which has been used in many big data and database systems, especially in the Hadoop ecosystem. It is supported by many data formats including Apache Parquet [5] and Apache ORC [6]. Similar to LZ77, a compressed Snappy file contains two types of tokens, including the literal tokens and the copy tokens. Both types of tokens have different lengths and formats. Fig. 1.3 and Fig. 1.4 illustrate the formats of the literal token and the copy token, respectively. The first byte of a token is called the tag byte. It contains information of the token type, the token length, and the size of extra bytes. If the last two bits of the tag byte are detected to be “00", it means the token is a literal token, and the first six bits stand for the length of the literal content. If this length is too large, it uses the succeeding one byte or two bytes to represent the length of the literal content. Meanwhile, the first six bits of the tag byte will be set to “111100" or “111101" depending on the length. Thus, a literal token can be 1 to 3 bytes in size (without the literal content). Similarly, the current Snappy implementation supports two different sizes of copy token, indicated by the last two bits of the tag byte. If they are “01", the upcoming one byte is used as extra information for the offset. If they are “10", the upcoming two bytes are used together to represent the offset. The Snappy compression algorithm works on a 64KB block level, which means every 64KB block in the original sequence is compressed independently and combined together afterward.

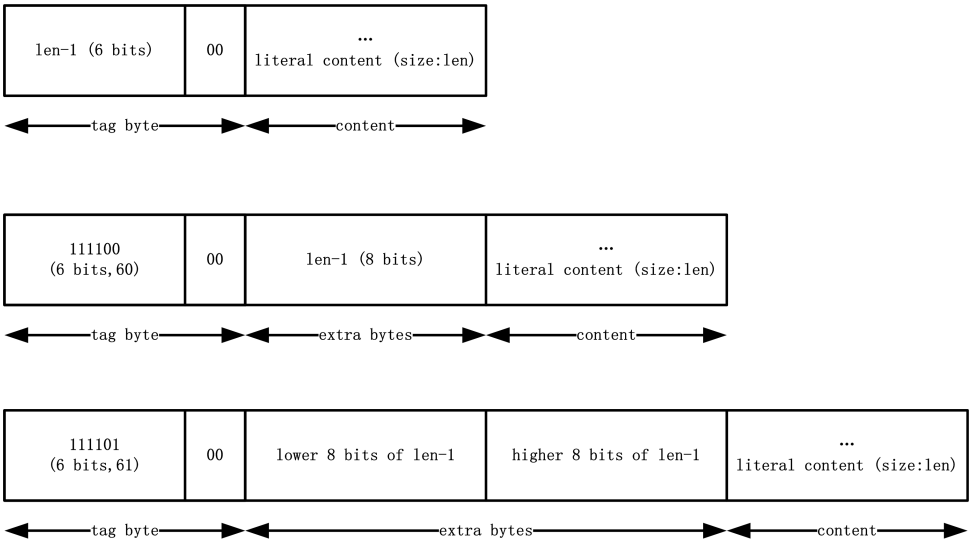


Figure 1.3: Snappy Literal Token Format

Snappy decompression is the reverse process of the compression. It converts a

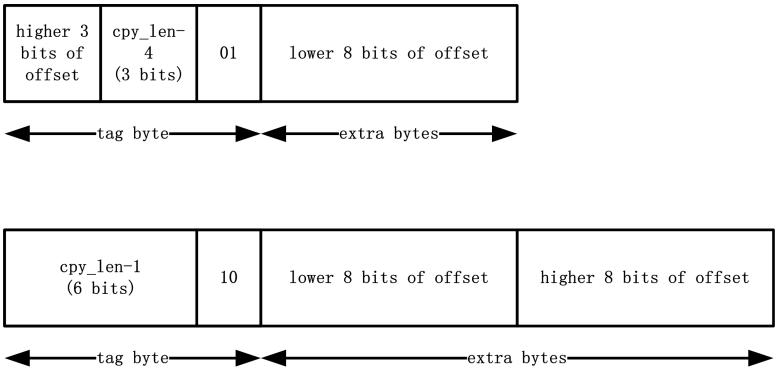


Figure 1.4: Snappy Copy Token Format

stream consisting of different types of tokens into an uncompressed sequence. It maintains a 64KB-block history during the decompression. If a literal token is detected, it copies the literal content to the history directly. Otherwise, it uses the “offset" to locate the repeated sequence, and to copy a size of “length" sequence from the located position to the history. Once a 64KB block is filled, decompression of this block is completed, and a new history is started. Since there are many dependencies during parsing the token, such as locating the token boundary and locating the block boundary, it is difficult to parallelize the Snappy decompression. We discuss a potential solution to parallelize this process and optimize the decompression performance in Chapter 3.

HASH JOINS

The join is a commonly used operation in table-based databases. It combines tuples from different tables that meet specific conditions. In most cases, this means having a common key. If a tuple in one table shares the same key with a tuple in the other table, a match is found. The join finds all these matches and outputs the combination of these matched tuple pairs. There are many different join algorithms including nested-loop joins, sort-merge joins, and hash joins, etc. Among these join algorithms, the hash join is understood to be one of the most efficient join algorithms since it is a linearly scalable algorithm. The simplest hash join algorithm is the classical hash join [7]. As illustrated in Fig. 1.5, the classical hash join builds a single hash table from one table which is used to find matches for the other table. It contains two phases, including the build phase and the probe phase. The build phase reads tuples from table R and generates a hash table. During the probe phase, the hash table is used to find the potential matched tuples in table S that is validated afterward. As we know, the complexity of this algorithm is $O(|R|+|S|)$, where $|R|$ stands for the number of tuples in table R, and $|S|$ stands for the number of tuples in table S. However, this can be improved by utilizing more processors or more processing elements. As demonstrated in Fig. 1.6, by dividing both input tables into portions and assigning them to p different workers, an ideal speedup of p can be achieved, compared with the classical hash join. However, this may still suffer a large number of cache misses, which leads to longer latency and bad throughput performance.

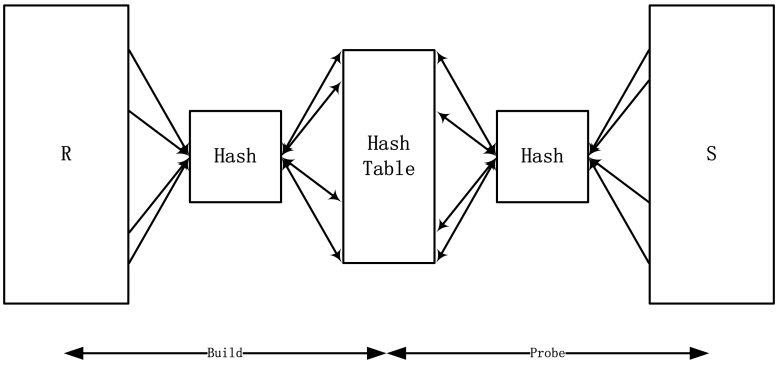


Figure 1.5: Classical Hash Join

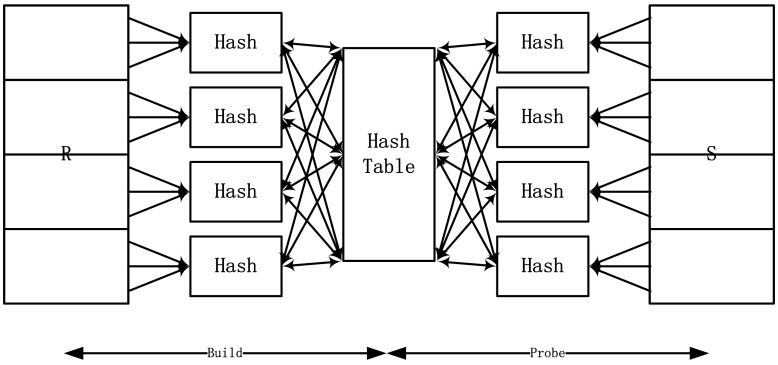


Figure 1.6: Classical Hash Join on Multiple Processors (Non-Partitioning Hash Join)

An efficient way to solve the cache miss problem is partitioning the tables to fit the size of the cache [8]. This method is called the partitioning hash join. In contrast, the classical hash join is called the non-partitioning hash join. Fig. 1.7 gives an overview of this idea. The main idea of this algorithm is to add an extra phase to partition relations into small chunks with each size fitting in the cache by hashing on their key values before the build phase. Consequently, tuples in one bucket of relation R can only match tuples in one bucket with a same bucket number of relation S. Thus, the hash table of one bucket can be stored in the cache, leading to reduce cache misses. An improved algorithm, radix hash join [9], further splits the partition phase into multiple passes to reduce the possibility of TLB (Translation Look-aside Buffer) misses introduced by the partition phase.

1.1.2. DATABASE ACCELERATION

Databases have now moved from hard disk drives to DRAM memory and NVMe which have much higher data access rates. These new technologies allow for two orders of magnitude more bandwidth between the CPU and stored database compared to traditional solutions. As a result, some database operations are transformed from

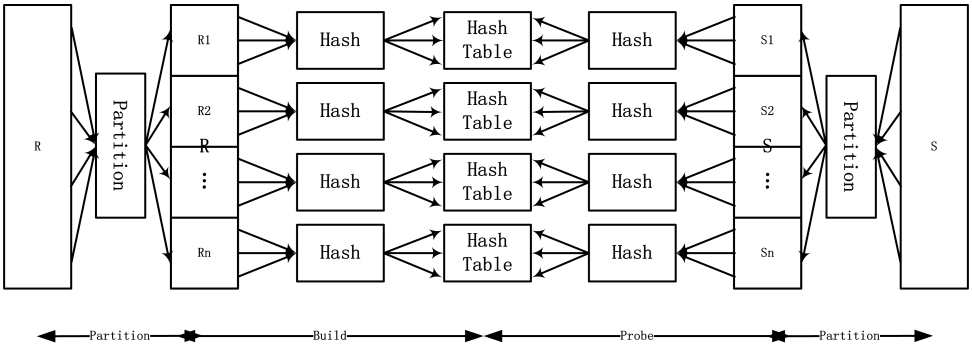


Figure 1.7: Partitioning Hash Join

bandwidth-bound to compute-bound. As a result, new computational solutions are needed to improve the performance of database processing in modern database systems. To solve this problem, the community has shifted their attention to heterogeneous processors such as *graphics processing units* (GPUs) [10–12], *field programmable gate arrays* (FPGAs) [13,14], etc. This thesis focuses on FPGA-based acceleration, and explores using FPGAs to accelerate database systems.

An FPGA is a reconfigurable chip, where the internal functionality can be reprogrammed. It consist of a large number of programmable logic blocks, a configurable interconnect fabric, local memory, as well as small general purpose processing devices. It intrinsically supports a high degree of parallelism, enabling effective utilization of task-level parallelism, data-level parallelism, and pipelining techniques. It also provides internal memory with low latency and high aggregate bandwidth. Recently, new FPGA devices have been introduced that deploy an on-socket high-bandwidth memory (HBM) [15] which provides up to 460GB/s bandwidth accessing a few GB accelerator-side memory.

These features make FPGAs a suitable accelerator for database systems, especially for streaming data processing and computation-heavy applications. Examples can be seen from both industry and academia, an early one of which is the IBM Netezza [13] data analytic appliance. In the Netezza system, an FPGA was placed between the CPU and the storage, performing decompression and aggregation in each node. Thus, it only transferred the pre-processed data to the CPU, relieving the CPU pressure. Another example from academia of database FPGA acceleration is DoppioDB [16], which extends MonetDB [17] with user defined functions in FPGAs. In addition, it provides software APIs using the Centaur framework [18] to bridge the gap between CPUs and FPGAs. Both of these examples present speedup in throughput performance compared to CPU-only solutions. There are also many existing studies focusing on accelerating a number of database operators such as aggregation [19,20], filtering [21,22], sort [23–25], join [26,27], etc. For more details about the background knowledge and related work, please see Chapter 2

1.2. MOTIVATION

Even though prior work shows significant performance gains in database systems by using FPGAs, both industry and academia are not showing large interest in integrating FPGAs into database systems due to the following three reasons. First, while FPGAs can provide high data processing rates, the system performance is bounded by the limited bandwidth from conventional IO technologies. Second, FPGAs are competing with a strong alternative, GPUs, which can also provide high throughput and are much easier to program. Last, programming FPGAs typically requires developers to have full stack skills, from high-level algorithm design to low-level circuit implementation.
Fortunately, these challenges are being addressed by various technology innovations in the field that improve the opportunity to create viable accelerated FPGA solutions for database system in the coming years, evidence of which can be seen in current technology developments. One of them is that data interfacing technologies develop so fast that the interconnection between memory and accelerators is expected to have main-memory scale bandwidth, e.g. the OpenCAPI [28] from IBM and the Compute Express Link [29] from Intel. In addition, FPGAs are incorporating new even higher-bandwidth memory technologies such as HBM, giving FPGAs a chance to bring the highly parallel computation capabilities of the FPGA together with a high-bandwidth large-capacity local memory. Finally, emerging FPGA development tool chains including HLS (high-level synthesis), new programming frameworks, and SQL-to-FPGA compilers, provide developers with better programmability. Therefore, FPGAs are becoming attractive again as database accelerators, making this a a good time to reconsider integrating FPGAs in database systems.

1.3. RESEARCH QUESTIONS

Along with the development of new technologies, some new questions are raised, and new challenges need to be addressed regarding database acceleration in the context of FPGAs. The central hypothesis of this thesis is that **FPGAs can be productively used to accelerate in-memory database operations**. This hypothesis can be divided into multiple research questions. In this thesis, we address four research questions related to the following topics: surveying in-memory database acceleration on FPGAs, acceleration of decompression operations, analysis of hash joins in software, and acceleration of hash joins in hardware. In the following, we present the research questions addressed in this thesis and identify in which chapter they are discussed.

- **Can FPGAs be productively applied to in-memory database acceleration?**

  FPGA have been used to accelerate database system in the past, especially for HDD-based database systems. It is believed that in-memory databases can be accelerated by FPGAs as well, depending on the operations. Thus, investigations should be carried out on identifying the current state of the art in this topic in both academia and industry. The overview presented in Chapter 2 concludes that performance of operations such as sort can be improved by adopting FPGAs. Subsequently, we implement our own hardware accelerator for such operations to make

them more efficient.

- **Can FPGAs accelerate compute-intensive operations given the newly available bandwidth?**

  Since the speed of data interconnect has improved significantly in recent years, previously proposed FPGA designs might not be able to fully consume the newly available bandwidth, especially for compute-intensive applications. Consequently, accelerator architectures of compute-intensive database operations that can leverage such huge data bandwidth should be investigated. This thesis explores new accelerator architectures of compute-intensive operations in databases and uses the Snappy decompressor as an example. The details are shown in Chapter 3.

- **What factors influence memory-intensive operations?**

  FPGAs are known to accelerate compute-intensive operations. In contrast, for memory-intensive operations, it is not very clear to what degree an FPGA can help to improve their performance. Thus, it is important to carry out performance analysis on the software operations to identify the most important factors that influence their performance and potential acceleration on hardware. This thesis uses the hash join operation as an example and presents analysis on how its performance is influenced by different factors, which is explained in Chapter 4.

- **Can FPGAs accelerate memory-intensive operations?**

  Even after the impact factors of memory-intensive operations running on software have been studied, it is not trivial to use this knowledge in designing their corresponding accelerators on FPGAs. Therefore, accelerator design of memory-intensive operations should be studied. This thesis studies the accelerator architecture of hash join as an example of memory-intensive operations that can benefit from the software analysis mentioned above. See Chapter 5 for more details.

1.4. RESEARCH METHODS

To address the research questions discussed above, we use the following research method. We first start with an analysis of the different classes of applications used in the database domain. Next, we select example algorithms in each of these classes of applications. This is followed by an analysis of the algorithms to identify the specific bottlenecks in the application and model its potential acceleration. These algorithms are then implemented in hardware, and their performance is measured and compared with the expected performance in the model. This research method has been applied throughout the thesis in the different chapters as discussed in the list below.

- First, we conduct a comprehensive study and survey on the topic of database acceleration. We start by investigating the current developments for this topic and review state-of-the-art research in the field, and summarize the reasons that held FPGAs back in the context of database acceleration. We identify two classes of operations that require a different acceleration approach: compute-intensive and

memory-intensive operations. Then, we study the new technology trends that bring new opportunities to FPGAs, followed by giving a qualitative analysis of the topic and present preliminary conclusions.

- We choose Snappy decompression as a compute-intensive operation example, and design and implement an accelerator and demonstrate that FPGAs are able to perform compute-intensive operations at a throughput that satisfies the needed performance requirements, thereby eliminating the computational bottleneck of these operations. We first analyze the software algorithm and identify its limitations in terms of parallelization. Then, we present a method to resolve these problems and implement the idea in an ADM-9V3 card integrated with a Xilinx VU3P FPGA. The implementation is validated and measured using a wide range of input files ranging from several MB to several GB in size, from highly compressed files to almost non-compressed files, and from generated synthetic data to practical data. The measurement on the hardware design is then compared with the software implementation running on a Power9 CPU.

- For memory-intensive operations, we use hash joins as an example. We first theoretically analyze the important factors influencing the performance, which we use to build a model to predict the performance. The performance model is evaluated on two different hash join algorithms running on the Intel x86 processor and an IBM Power 8 processor using a variety of data sets that range from several MB to several GB in size. Using this performance model, we can optimize the performance for different processor architectures.

- The knowledge learned from this analysis is then adopted to implement a hash join accelerator in the FPGA to show that FPGAs are also able to improve the performance of some memory-intensive operations. The targeted device is a Xilinx VU37P FPGA integrated with 8GB HBMs on the ADM-9H7 card that can communicate with the host memory using the OpenCAPI interface. We evaluate the HBMs performance by simulation using multiple data sets with different access patterns. This simulated bandwidth performance is used to evaluate the performance of the proposed hash join accelerator using a mathematical performance model to demonstrate that the proposed methods can saturate the interface bandwidth.

1.5. CONTRIBUTIONS

We summarize our contributions in this thesis as follows.

- We present a comprehensive survey on using FPGAs to accelerate database systems. We analyze the pros and cons in current FPGA-accelerated database system architecture alternatives. By studying the memory-related technology trends, we conclude that FPGAs deserve to be reconsidered for integration in database systems. Also, we give an overview of the state-of-the-art studies on database operator acceleration as well as discuss some potential solutions to optimize and improve them.

- We propose a method to increase the decompression parallelism by refining the tokens in a compressed file into independent BRAM commands. We propose a recycle method to reduce the stalls caused by the read-after-write data dependencies during the compression. We apply these two techniques to the Snappy decompression algorithm and present a Snappy decompressor that can process multiple tokens per cycle. We also present a proof-of-concept Parquet-to-Arrow converter that can benefit from the proposed Snappy decompressor to improve the conversion speed.
- We analyze the performance of main memory hash join in the CPU architecture. We discuss factors that impact performance of hash joins and point out the importance of granularity. Based on these factors, we proposed a performance model that considers both computation and memory accesses to estimate the hash join performance. Finally, we study different hash join algorithms and validate the proposed model in different processor architectures.
- We propose an accelerator architecture of the hash join that utilizes the HBMs to store the hash table. The proposed method allows all HBM channels to operate independently. A pre-partition method is presented to drive the HBM traffic to the appropriate channels, in order to reduce the traffic contention, and thus improve the bandwidth efficiency.

1.6. THESIS ORGANIZATION

The remainder of this thesis is organized as follows.

- In **Chapter 2**, we introduce the background and survey the related work of using FPGAs to accelerate database systems.
- In **Chapter 3**, we present the implementation of the FPGA-based Snappy decompressor, as well as the architecture of the Parquet-to-Arrow converter.
- In **Chapter 4**, we discuss the in-memory hash joins and explain the mathematical performance model.
- In **Chapter 5**, we describe the architecture of the FPGA-based hash join accelerator that utilizes the HBMs.
- In **Chapter 6**, we summarize and conclude our work, and recommend possible directions for future work.

2

BACKGROUND AND RELATED WORK

SUMMARY

This chapter surveys using FPGAs to accelerate in-memory database systems targeting designs that can operate at the speed of main memory. We first introduce the background and review the previous FPGA-based database system architectures. After that we discuss the challenges of integrating FPGAs into database systems and study a number of technology trends. We also summarize the state-of-the-art research on FPGA-accelerated database operations. Based on the study and the summaries, we present the major challenges and possible solutions for adopting accelerators for high bandwidth in-memory databases.

The content of this chapter is based on the following paper:

J. Fang, Y.T.B. Mulder, J. Hidders, J. Lee, H.P. Hofstee, *In-Memory Database Acceleration on FPGAs: A Survey*, International Journal on Very Large Data Bases (VLDBJ), 2019, https://doi.org/10.1007/s00778-019-00581-w.

The VLDB Journal
https://doi.org/10.1007/s00778-019-00581-w

SPECIAL ISSUE PAPER

2

In-memory database acceleration on FPGAs: a survey

Jian Fang[1] · Yvo T. B. Mulder[2] · Jan Hidders[3] · Jinho Lee[5] · H. Peter Hofstee[1,4]

Received: 5 December 2018 / Revised: 8 July 2019 / Accepted: 11 October 2019

Abstract
While FPGAs have seen prior use in database systems, in recent years interest in using FPGA to accelerate databases has declined in both industry and academia for the following three reasons. First, specifically for in-memory databases, FPGAs integrated with conventional I/O provide insufficient bandwidth, limiting performance. Second, GPUs, which can also provide high throughput, and are easier to program, have emerged as a strong accelerator alternative. Third, programming FPGAs required developers to have full-stack skills, from high-level algorithm design to low-level circuit implementations. The good news is that these challenges are being addressed. New interface technologies connect FPGAs into the system at main-memory bandwidth and the latest FPGAs provide local memory competitive in capacity and bandwidth with GPUs. Ease of programming is improving through support of shared coherent virtual memory between the host and the accelerator, support for higher-level languages, and domain-specific tools to generate FPGA designs automatically. Therefore, this paper surveys using FPGAs to accelerate in-memory database systems targeting designs that can operate at the speed of main memory.

Keywords Acceleration · In-memory database · Survey · FPGA · High bandwidth

1 Introduction

The computational capacity of the *central processing unit* (CPU) is not improving as fast as in the past or growing fast enough to handle the rapidly growing amount of data. Even though CPU core-count continues to increase, power per core from one technology generation to the next does not decrease at the same rate and thus the "power wall" [7] limits progress. These limits to the rate of improvement bring a demand for new processing methods to speed up database systems, especially in-memory database systems. One candidate is *field-programmable gate arrays* (FPGAs), that have been noted by the database community for their high parallelism, reconfigurability, and low power consumption, and can be attached to the CPU as an IO device to accelerate database analytics. A number of successful systems and research cited throughout this paper have demonstrated the potential of using FPGAs as accelerators in achieving high throughput. A commercial example is IBM Netezza [41], where (conceptually) an FPGA is deployed in the data path between *hard disk drives* (HDDs) and the CPU, performing decompression and pre-processing. This way, the FPGA mitigates the computational pressure in the CPU, indirectly amplifying the HDD-bandwidth that often limited database analytics performance.

While FPGAs have high intrinsic parallelism and very high internal bandwidth to speed up kernel workloads, the low interface bandwidth between the accelerator and the rest of the system has now become a bottleneck in high-bandwidth in-memory databases. Often, the cost of moving data between main memory and the FPGA outweighs the computational benefits of the FPGA. Consequently, it is a challenge for FPGAs to provide obvious system speedup, and only a few computation-intensive applications or those with

Jian Fang
j.fang-1@tudelft.nl

Yvo T. B. Mulder
yvo.mulder@ibm.com

Jan Hidders
jan.hidders@vub.be

Jinho Lee
leejinho@yonsei.ac.kr

H. Peter Hofstee
hofstee@us.ibm.com

1 Delft University of Technology, Delft, The Netherlands
2 IBM Research and Development, Böblingen, Germany
3 Vrije Universiteit Brussel, Brussels, Belgium
4 IBM Research, Austin, TX, USA
5 Yonsei University, Seoul, Korea

Published online: 26 October 2019

data sets that are small enough to fit in the high-bandwidth on-FPGA distributed memories can benefit.

Even with higher accelerator interface bandwidth, the difficulty of designing FPGA-based accelerators presents challenges. Typically, implementing efficient designs and tuning them to have good performance requires developers to have full-stack skills, from high-level algorithm design to low-level circuit implementation, severely limiting the available set of people who can contribute.

While some of these challenges also apply to GPUs, GPUs have become popular in database systems. As is the case for FPGAs, GPUs can benefit from their massive parallelism and provide high throughput performance, but also like FPGAs, GPU to system memory bandwidth typically falls well short of the bandwidth of the CPU to system memory. However, compared to FPGAs GPUs support much larger on-device memory (up to 32 GB) that is accessible at bandwidths (more than 800 GB/s) that exceed those of the CPU to system memory. For these reasons, a GPU-accelerated system can provide benefit in a larger number of cases.

Emerging technologies are making the situation better for FPGAs. First, new interface technologies such as OpenCAPI [123], *Cache Coherent Interconnect for Accelerators* (CCIX) [13], and *Compute Express Link* (CXL) [112] can bring aggregate accelerator bandwidth that can exceed the available main-memory bandwidth. For example, an IBM POWER9 SO processor can support 32 lanes of the OpenCAPI interface, supplying up to 100 GB/s for each direction, while the direct-attach DDR4 memory on the same processor provides up to 170 GB/s (2667MT/s $*$ 8 channels) in total [129]. Another feature brought to FPGAs by the new interfaces is shared memory. Compared to using FPGAs as I/O devices where FPGAs are controlled by the CPU, in the OpenCAPI architecture, the coherency is guaranteed by the hardware. FPGAs are peers to the CPUs and share the same memory space. With such a high-bandwidth interface, the computational capability and the parallelism of the accelerator can now be much more effectively utilized.

Apart from new interface technologies, high-bandwidth on-accelerator memory is another enabler for FPGAs. Some FPGAs now incorporate *high bandwidth memory* (HBM) [138] and have larger local memory capacity as well as much higher (local) memory bandwidth. Similar to the GPUs with HBM, such high-bandwidth memory with large capacity allows FPGAs to store substantial amounts of data locally which can reduce the amount of host memory access, and bring the potential to accelerate some of the data-intensive applications that require memory to be accessed multiple times.

In addition, FPGA development tool chains are improving. These improvements range from *high-level synthesis* (HLS) tools to domain-specific FPGA generation tools such as query-to-hardware compilers. HLS tools such as Vivado HLS [38] and OpenCL [115] allow software developers to program in languages such as C/C++ but generate hardware circuits automatically. Other frameworks such as SNAP [136] further automate the designs of the CPU-FPGA interface for developers. In this case, the hardware designer can focus on the kernel implementation, and the software developers do not have to concern themselves with the underlying technology. Domain-specific compilers such as query-to-hardware compilers (e.g., Glacier [86]) can even compile SQL queries directly into FPGA implementations.

Therefore, with these emerging technologies, we believe that FPGAs can again become attractive as database accelerators, and it is a good time to reexamine integrating FPGAs into database systems. Our work builds on [127] which has presented an introduction and a vision on the potential for FPGA's for database acceleration. Related recent work includes [98] which draws similar conclusions with respect to the improvements in interconnect bandwidth. We focus specifically on databases, we include some more recent work, and we emphasize the possibilities with the new interface technologies.

In this paper, we explore the potential of using FPGAs to accelerate in-memory database systems. Specifically, we make the following contributions.

- We present the FPGA background and analyze FPGA-accelerated database system architecture alternatives and point out the bottlenecks in different system architectures.
- We study the memory-related technology trends including database trends, interconnection trends, FPGA development trends, and conclude that FPGAs deserve to be reconsidered for integration in database systems.
- We summarize the state-of-the-art research on a number of FPGA-accelerated database operators and discuss some potential solutions to achieve high performance.
- Based on this survey, we present the major challenges and possible future research directions.

The remainder of this paper is organized as follows: In Sect. 2, we provide FPGA background information and present the advantages of using FPGAs. Section 3 explains the current database systems accelerated by FPGAs. We discuss the challenges that hold back use of FPGAs for database acceleration in Sect. 4. The database, interconnect and memory-related technology trends are studied in Sect. 5. Section 6 summarizes the state-of-the-art research on using FPGAs to accelerate database operations. Section 7 presents the main challenges of using high-bandwidth interface attached FPGAs to accelerate database systems. Finally, we conclude our work in Sect. 8.

System designers may be interested in Sect. 3 for the system architecture overview, Sect. 4 for the system limita-

tions, and Sect. 5 for the technology trends that address these limitations. FPGA designers might want to concentrate on Sect. 6 that discusses the state of the art for high-bandwidth operators relevant to database queries. For performance analysts, Sect. 4 gives a brief comparison between FPGAs and GPUs, as well as the challenges of FPGA regarding database acceleration. For the performance of each operator, a deeper discussion is presented in Sect. 6. For software developers, Sect. 2 provides an introduction to FPGAs, while FPGA programming is discussed in Sect. 4 and 5. We also present lessons learned and potential future research directions in Sect. 7 addressing different groups of researchers.

2 FPGA background

This section gives an introduction to FPGAs, and provides software researchers and developers with background knowledge of FPGAs including architecture, features, programming, etc.

2.1 Architecture

An FPGA consists of a large number of programmable logic blocks, interconnect fabric and local memory. *Lookup tables* (LUTs) are the main component in programmable logic. Each LUT is an n-input 1-output table,[1] and it can be configured to produce a desired output according to the combination of the n inputs. Multiple LUTs together can be connected by the configurable interconnect fabric, forming a more complex module. Apart from the logic circuits, there are small memory resources (registers or flip-flops) to store states or intermediate results and larger block memory (*Block RAMs* or BRAMs) to act as local memory. Recently, FPGA chips are equipped with more powerful resources such as built-in CPU cores, *Digital Signal Processor* (DSP) blocks, *UltraRAM* (URAM), HBMs, preconfigured I/O blocks, and memory-interface controllers.

2.2 Features

The FPGA is a programmable device that can be configured to a customized circuit to perform specific tasks. It intrinsically supports high degrees of parallelism. Concurrent execution can be supported inside an FPGA by adopting multi-level parallelism techniques such as task-level parallelization, data-level parallelization, and pipelining. In addition, unlike the CPU where the functionality is designed for generic tasks that do not use all the resources efficiently for a specific application, the circuit in an FPGA is highly customizable, with only the required functions implemented. Even though building specific functions out of reconfigurable logic is less efficient than building them out of customized circuits, in many cases, the net effect is that space is saved and more processing engines can be placed in an FPGA chip to run multiple tasks in parallel. Also, the capability of customizing hardware leads to significant power savings compared to CPUs and GPUs, when the required functionality is not already directly available as an instruction. The FPGA can also support data processing at low latency due to the non-instruction architecture and the data flow design. In CPUs, the instructions and data are stored in the memory. Executing a task is defined as running a set of instructions, which requires fetching instructions from memory. However, FPGAs define the function of the circuit at design-time, where the latency is dependent on the signal propagation time. Apart from that, the data flow design in FPGAs allows forwarding the intermediate results directly to the next components, and it is often not necessary to transfer the data back to the memory.

2.3 FPGA-related bandwidth

As we focus on the bandwidth impact on this paper, we give a brief introducing of FPGA-related bandwidth and present the summary in Table 1. Similar to the CPU memory hierarchy, the memory close to the FPGA kernel has the lowest latency and highest bandwidth, but the smallest size. The FPGA internal memory including BRAM and URAM typically can reach TB/s scale bandwidth with a few nanoseconds latency. The on-board DDR device can provide tens GB/s bandwidth, While the HBM that within the same socket with the FPGA have hundreds of GB/s bandwidth, and both of them require tens to hundreds nanoseconds latency to get the data. The bandwidth to access the host memory typically is the lowest one in this hierarchy. However, it provides the largest memory capacity.

Hiding long memory latency is a challenge for FPGA designs. Typically, applications with streaming memory access patterns are less latency-sensitive: because the requests are predictable it is easier to hide the latency. However, applications that require a large amount of random access (e.g., as hash join) or unpredictable streaming access (e.g., sort) could get stalls due to the long latency. In this case, we might need to consider using memory with lower latency or transform the algorithms to leverage streaming. We discuss more details based on different operators in Sect. 6.

2.4 Programming

The user-defined logic in the FPGA is generally specified using a *hardware description language* (HDL), mostly

[1] Multi-output LUTs are available now. See Figure 1-1 in https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf.

Table 1 FPGA-related bandwidth and latency (from data source to FPGA kernels)

Mem source	Mem type	BW (GB/s)	Latency (ns)	Capacity (MB)
Internal	BRAM	$\geq 10^3$	10^0	10^0
	URAM	$\geq 10^3$	10^1	10^1
On-board	HBM	10^2–10^3	10^1–10^2	10^3
	DRAM	10^1–10^2	10^1–10^2	10^4
Host	DRAM	10^1	$\geq 10^2$	$\geq 10^5$

VHDL or Verilog. Unlike software programming languages such as C/C++ handling sequential instructions, HDLs describe and define an arbitrary collection of digital circuits. It requires the developers to have knowledge on digital electronics design, meaning understanding how the system is structured, how components run in parallel, how to meet the timing requirement, and how to trade off between different resources. This is one of the main reasons that make the software community reluctant to use FPGAs.

High-level synthesis (HLS) tools such as Vivado HLS [38] and Altera OpenCL [115] overcome this problem by supporting software programmers with the feasibility of compiling standard languages such as C/C++ and higher-level hardware-oriented languages like systemC into register-transfer level (RTL) designs. In such a design procedure, HLS users write C code and design the interface protocol, and the HLS tools generate the microarchitecture. Apart from generating the circuit itself, programming frameworks such as OpenCL [121] provide frameworks for designing programs that run on heterogeneous platforms (e.g., CPU+FPGA). These frameworks typically specify variants of standard programming languages to program the kernels and define application programming interfaces to control the platforms. The corresponding *software development kits* (SDKs) are now available for both Xilinx FPGAs [137] and Intel FPGAs [56]. There are also some domain-specific compilers that can compile SQL queries into circuits or generate the circuit by setting a couple of parameters. An example is Glacier [86]) which provides a component library and can translate streaming queries into hardware implementations.

3 FPGA-based database systems

How to deploy FPGAs in a system is a very important question for system designers. There are many ways to integrate FPGAs into database systems. The studies in [82,83] categorize the ways FPGAs can be integrated by either placing it between the data source and CPU to act as a filter or by using it as a co-processor to accelerate the workload by offloading tasks. Survey [57] presents another classification that contains three categories including "on-the-side" where the FPGA is connected to the host using interconnect such as PCIe, "in data path" where the FPGA is placed between the storage/network and the CPUs, and "co-processor" where the FPGA is integrated together with the CPU in the same socket. In this section, we specify three possible database architectures with FPGA accelerators in a logical view and explain their shortcomings and advantages.

Fig. 1 FPGA as a bandwidth amplifier

3.1 Bandwidth amplifier

In a storage-based database system, the bottleneck normally comes form the data transmission to/from the storage, especially the HDD. Compared to hundreds of Gbit/s bandwidth supported by DRAM, the data rate of an HDD device remains at the 1 Gbit/s level, which limits the system performance. In these systems, FPGAs can be used to amplify the storage bandwidth.

As shown in Fig. 1, the FPGA is used as a decompress-filter between the data source (disks, network, etc.) and the CPU to improve the effective bandwidth. In this architecture, the compressed data is stored on the disks, and would be transferred to the FPGA, either directly through the interfaces like SCSI, SATA, Fibrechannel, or NVMe or indirectly, for network-attached storage or protocols like NVMe over Infiniband or Ethernet. In the FPGA, the data is decompressed and filtered according to some specific conditions, after which the data is sent to the CPU for further computation. As the compressed data size is smaller than the original data size, less data needs to be transferred from storage, improving the effective storage bandwidth indirectly.

The idea has proven to be successful by commercial products such as Netezza [41], or a few SmartNIC variants [80,92]. In Netezza, an FPGA is placed next to the CPU doing the decompression and aggregation in each node, and only the data for post-processing is transferred to the CPU. In a few SmartNIC products, an FPGA sits as a filter for the network traffic. By applying compression/decompression or deduplication, they greatly enhance the effective bandwidth

of an network-to-storage applications. A similar idea is also studied by prior research such as the ExtraV framework [72], where the FPGA is integrated into the system in an implicit way. The FPGA is inserted in the data path between the storage and the host, performing graph decompression and filtering. Some research [42,128,139] shows that even without doing the decompression by only performing filtering and aggregation on the FPGA, one can significantly reduce the amount of data sent to the CPU, as well as relieve the CPU computational pressure. This is a good solution for latency-sensitive applications with data stream processing, where the FPGA capability for the high throughput and low latency processing is demonstrated.

3.2 IO-attached accelerator

Attaching FPGAs as an IO-attached accelerators is another conventional way to deploy accelerators in the systems, especially for computational-intensive applications in which the CPUs are the bottleneck of these systems. In this case, FPGAs are used as IO devices performing data processing workloads offloaded by CPUs. Figure 2 illustrates the architecture of using the FPGA as an IO-attached accelerator. In this architecture, the FPGA is connected to the CPU through buses such as PCIe, and the FPGA and CPU have their own memory space. When the FPGA receives tasks from the CPU, it first copies the data from the host memory to the device memory. Then the FPGA fetches data from the memory and writes the results back to the device memory after processing it. After that, the results can be copied back to the host memory.

This approach is illustrated by both industry and academic solutions. Kickfire's MySQL Analytic Appliance [63], for example, connects the CPU with a PCIe-attached FPGA card. The offloaded queries can be processed in the FPGA with a large amount of high-bandwidth on-board memory. Xtremedata's dbX [106] offers an in-socket FPGA solution where the FPGA is pin compatible with the CPU socket. Key database operations including *joins*, *sorts*, *groupbys*, and *aggregations* are accelerated with the FPGA. In recent academic research on accelerating database operators such as sort [18] and join [109], the data is placed in the device memory to avoid data copies from/to the host. This architecture is also present in GPU solutions such as Kinetica [66], MapD [81], and the research work in [51]. A drawback of this architecture is that it requires extra copies (from the host memory to the device memory and the other way around) which leads to longer processing latencies. Also the application must be carefully partitioned, as the accelerator is unable to access memory at-large. The separate address spaces also affect debug, and performance tools. Even today, it is often difficult to get an integrated view of the performance of a GPU-accelerated system for example. Placing the whole database in the device memory such as the design from [43] can reduce the impact of the extra copies. However, since the device memory has limited capacity which is much smaller than the host memory, it limits the size of the database and the size of the applications.

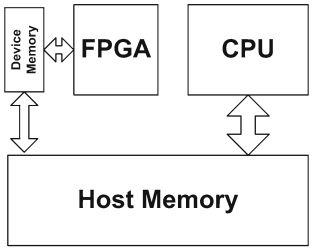


Fig. 2 FPGA as an IO-attached accelerator

3.3 Co-processor

Recent technology allows FPGA to be deployed in a third way where the FPGA acts as a co-processor. As shown in Fig. 3, in this architecture, the FPGA can access the host memory directly, and the communication between the CPU and the FPGA is through shared memory. Unlike the IO-attached deployment, this deployment provides the FPGA full access to system memory, shared with the CPU, that is much larger than the device memory. In addition, accessing the host memory as shared memory can avoid copying the data from/to the device memory. Recently, there have been two physical ways to deploy FPGAs as co-processors. The first way tightly couples the FPGA and the CPU in the same die or socket, such as Intel Xeon+FPGA platform [48] and ZYNQ [27]. The FPGA and CPU are connected through *Intel QuickPath Interconnect* (QPI) for Intel platforms and *Accelerator Coherency Port* (ACP) for ZYNQ, and the coherency is handle by the hardware itself. The second method connects the FPGAs to the host through coherent IO interfaces such as IBM *Coherent Accelerator Processor Interface* (CAPI) or OpenCAPI [123], which can provide high bandwidth access to host memory. The coherency between the CPU and the FPGA is guaranteed by extra hardware proxies. Recently, Intel also announced a similar off-socket interconnect called *Compute Express Link* (CXL) that enables a high-speed shared-memory based interaction between the CPU, platform enhancements and workload accelerators.

DoppioDB [114] is a demonstrated system of this architecture from academia. It extends MonetDB with user-defined functions in FPGAs, along with proposing a Centaur framework [97] that provides software APIs to bridge the gap between CPUs and FPGAs. Other research work studies the acceleration of different operators including compression [104], decompression [35], sort [146] and joins [49], etc.

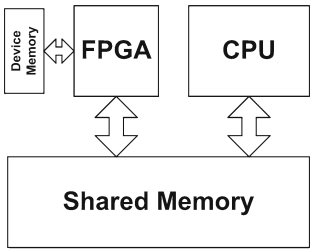


Fig. 3 FPGA as a co-processor

4 What has held FPGAs back?

In this section, we discuss three main challenges that have reduced the interest in using FPGAs to accelerate in-memory database systems in both industry and academia. System designers and performance analysts might be interested in Sects. 4.1 and 4.3 where the system performance limitation and the comparison to GPUs are explained, while software developers can focus on the FPGA programmability in Sect. 4.2. In Sect. 5 we discuss the technology trends that address the challenges discussed here.

4.1 Significant communication overhead

A first obvious challenge for using any accelerator is communication overhead. While many FPGA accelerators discussed in Sect. 6 such as [49,61,125] have demonstrated that FPGAs can achieve high (multi-) kernel throughput, the overall performance is frequenty-limited by the low bandwidth between the FPGA and the host memory (or CPU). Most recent accelerator designs access the host memory data through PCIe Gen3, which provides a few GB/s bandwidth per channel or a few tens of GB/s accumulated bandwidth. This bandwidth is not always large enough compared to that between the CPU and the main memory in the in-memory database systems, and the cost of data transmission from/to the FPGA might introduce significant overhead.

In addition, transferring data from the host to FPGAs that are not in the same socket/die as the CPU increases latency. When the FPGA operates in a different address space, latency is increased even more (a few microseconds is not uncommon). This brings challenges to the accelerator designs, especially for those have unpredictable memory access patterns such as scatter, gather and even random access. To hide the long latency, extra resources are required to buffer the data [88] or to maintain the states of a massive number of tasks or threads [49].

4.2 Weak programmability

Another big challenge is the difficulty of developing FPGA-based accelerators and effectively using them, which has two main reasons.

First, programming an FPGA presents a number of challenges and tradeoffs that software designers do not typically have to contend with. We give a few examples. To produce a highly-tuned piece of software, a software developer occasionally might have to restructure their code to enable the compiler to do sufficient loop unrolling (and interleaving) to hide memory latencies. When doing so there is a tradeoff between the number of available (renamed) registers and the amount of loop unrolling. On the FPGA, the equivalent of loop unrolling is pipelining, but as the pipeline depth of a circuit is increased, its hardware resources change, but its operating frequency can also change, making navigating the design space more complex. Even when we limit our attention to the design of computational kernels, an awareness of the different types of resources in an FPGA may be required to make the right tradeoffs. The implementation in an FPGA is either bound by the number of the computation resources (LUTs, DSP, etc.) or the size of the on-FPGA memory (BRAM, URAM, etc.) or it can be constrained by the available wiring resources. Which of these limits the design informs how the design is best transformed. As can be seen from this example, implementing an optimized FPGA design typically requires developers to have skills across the stack to gain performance While nowadays HLS tools can generate the hardware circuits from software language automatically by adopting techniques such as loop unrolling and array partitioning [26,28], manual intervention to generate efficient circuits that can meet the timing requirements and sensibly use the hardware resources is still required too often. In many cases, the problem is outside the computational kernels themselves. For example, a merge tree that can merge multiple streams into a sorted stream might require prefetching and bufferring of data to hide the long latency. In this case, rewriting the software algorithms to leverage the underlying hardware or manually optimizing the hardware implementation based on the HLS output or even redesigning the circuit is necessary.

Second, generating query plans that can be executed on an FPGA-accelerated system demands a strong query compiler that can understand the underlying hardware, which is still lacking today. The flipside of having highly specialized circuits on an FPGA is that (parts of) the FPGA must be reconfigured when a different set, or a different number of instances of functions or kernels is needed, and FPGA reconfiguration times exceed typical context switch penalties in software by orders of magnitude. In software, the cost of invoking a different function can usually be ignored. Unlike the single-operator accelerators, we survey in Sect. 6, in real-

ity, a query is typically a combination of multiple operators. The query optimizer component in the query compiler optimizes the query plan by reordering and reorganizing these operators based on the hardware. While this field has been well studied in the CPU architecture, it becomes more challenging when taking FPGAs into account. Because of the long reconfiguration times, a query plan for a short running query may look vastly different than an optimized query plan for a long-running query, even if the queries are the same. Thus query compilers need to map the query plan to meet the query execution model in the FPGA. In addition, the FPGA designs may not be optimized and implemented for all required functions or operators. Thus, for some special functions or operators that have not been implemented in the FPGA or where FPGAs do not provide adequate performance gain, the query compiler should drive the query plan back to the pure CPU execution model. Without a shared address space, and a common method to access the data (e.g., the ability to lock memory), a requirement to flexibly move components of a query between the CPU and accelerators is significantly complicated.

4.3 Accelerator alternative: GPU

In recent years, the GPU has become the default accelerator for database systems. There are many GPU-based database systems from both industry and academia such as Kinetica/GPUDB [66,143], MapD [81], Ocelot [53], OmniDB [147], and GPUTx [52]. A comprehensive survey [15] summarizes the key approaches to using GPUs to accelerate database systems and presents the challenges.

Typically, GPUs achieve higher throughput performance while FPGAs gain better power-efficiency [23]. The GPU has a very large number of lightweight cores with fewer control requirements and provides a high degree of data-level parallelism. This is an important feature to accelerate database applications since many database operators are required to perform the same instructions on a large amount of data. Another important feature is that the GPU has large capacity high-bandwidth on-device memory that is typically much larger than the host main memory bandwidth and the CPU-GPU interface bandwidth. Such large local memory allows GPUs to keep a large block of hot data and can reduce the communication overhead with the host, especially for applications that need to touch the memory multiple times such as the partitioning sort [39] that achieves 28 GB/s throughput on a four-GPU POWER9 node.

While FPGAs cannot beat the throughput of GPUs in some database applications, the result might change in power-constrained scenarios. One of the reasons is that the data flow design on FPGAs can avoid moving data between memories. A study from Baidu [96] shows that the Xilinx KU115 FPGA is 4x more power-efficient than the GTX Titan GPU when running the sort benchmark. In addition, the customizable memory controllers and processing engines in FPGAs allow FPGAs to handle complex data types such as variable-length strings and latency-sensitive applications such as network processing that GPUs are not good at (we discuss more detail in Sect. 7.3).

However, when considering programmability and the availability of key libraries, FPGAs still have a long way to go compared to GPUs, as mentioned in Sect. 4.2. Consequently, engineers might prefer GPUs which can also provide high throughput but are much easier to program, debug, and tune for performance. In other words, GPUs have raised the bar for using FPGAs to accelerate database applications. Fortunately, the technology trends in Sect. 5 show that some of these barriers are being addressed, and thus it is worth it to have another look ar FPGAs for in-memory database acceleration.

5 Technology trends

In recent years, various new technologies have changed the landscape of system architecture. In this section, we study multiple technology trends including database system trends, system interconnect trends, FPGA development trends, and FPGA usability trends, and we introduce a system model that combines these technologies. We believe that these technology trends can help system designers to design new database system architectures, and help software developers to start using FPGAs.

5.1 Database system trends

Databases traditionally were located on HDDs, but recently the data is often held in-memory or on NVM storage. This allows for two orders of magnitude more bandwidth between the CPU and stored database compared to the traditional solution. Some of the database operators now become computation-bound in a pure CPU architecture, which demands new and strong processors such as GPUs and FPGAs.

However, the downside is that acceleration, or the offloading of queries, becomes more difficult due to the low FPGA-CPU interconnect bandwidth. When using FPGAs as IO-attached accelerator, typically *PCI Express* (PCIe) Gen 3 is used as an IO interconnect. It provides *limited bandwidth* compared to DRAM over DDRx, and may suffer resource contention when sharing PCIe resources between FPGA and NVM over NVMe, which may impact the performance. While PCIe Gen 4 doubles the bandwidth, it does not solve the communication protocol limitation that using FPGAs as IO-attached accelerators requires to copy the data

between the host memory and the device memory, resulting in extra data transmission overhead and long latency.

These limitations result in a high cost of data movement between the database and the FPGA. This limits the applicability of FPGAs in the data center. In order to accelerate databases with FPGAs (again), the interconnect has to overcome these limitations in order to become a viable alternative.

5.2 System interconnect trends

The bandwidth of system interconnects plateaued for quite a few years, after the introduction of PCIe Gen3. More recently the pace has increased, the PCI Express Consortium released the specification of Gen 4 in 2017 and Gen 5 in 2019, respectively [100], and is expected to release the specification of Gen 6 in 2021 [99]. Because of the long wait for a new generation, other initiatives had started, proposing new interconnect standards to solve the bandwidth bottlenecks mentioned in Sect. 4.

5.2.1 Increase in system interconnect bandwidth

Figure 4 shows collected data regarding DRAM, network and storage bandwidth in 2019 [70] and predicts the future until 2022 (indicated by the diamond-shaped markers). The bandwidth of PCIe was added to act as a proxy for interconnect bandwidth. For each generation of the PCIe standard, the bandwidth of sixteen lanes is plotted, since this is typically the maximum number of lanes per PCIe device. The DRAM data interface is inherently uni-directional and several cycles are required to turn the channel around. A memory controller takes care of this by combining reads and writes to limit the overhead cycles spent in configuring the channel. Therefore, DRAM bandwidth should be interpreted as either a read or write channel with the plotted bandwidth, while attached devices such as network and storage typically have a bi-directional link.

The slope of each of the fitted lines is important here. Clearly, both network and storage bandwidths are increasing at a much faster rate (steeper slope) than DRAM and PCIe. The network and storage slopes are similar and double every 18 months. PCIe doubles every 48 months, while it takes DRAM 84 months to double in bandwidth. A server typically contains one or two network interfaces, but often contains a dozen or more storage devices, lifting the blue line by an order of magnitude. Thus, a shift in balance is expected for future systems where DRAM, interconnect, network and storage bandwidth are about the same.

The fitted straight lines for each of the four data sets shown in Fig. 4 indicate exponential behavior. While it might look like accelerators, such as GPUs and FPGAs, will have to compete for interconnect bandwidth with network and storage, one option is to scale memory and interconnect bandwidth

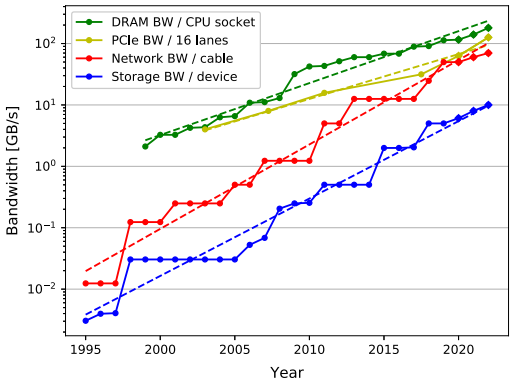


Fig. 4 Bandwidth trends at device-level. Data points were approximated from the referenced figures in order to add the PCI Express standard bandwidth and represent all bandwidths in GB/s [70,88]

accordingly. While scaling is the trend, as becomes apparent from the next paragraph, and works in the short-term, it does not solve the fundamental problem of limited DRAM bandwidth improvements.

The reason that DRAM bandwidth is not increasing at a similar pace is twofold. To increase bandwidth, either the number of channels or the channel frequency must be increased. However, each solution has significant implications. Every additional channel requires a large number of pins (order of 100) on the processor package (assuming an integrated memory controller) that increases chip area cost. Increasing channel frequency requires expensive logic to solve signal integrity problems at the cost of area, and more aggressive channel termination mechanisms at the cost of power consumption [30,47]. If the bandwidth of the interconnect is increased to the same level as DRAM, the same problems that DRAM faces will be faced by attached devices.

5.2.2 Shared memory and coherency

Solely increasing bandwidth will not solve all of our problems, because the traditional IO-attached model will become a bottleneck. Currently, the host processor has a shared memory space across its cores with coherent caches. Attached devices such as FPGAs, GPUs, network and storage controllers are memory-mapped and use a DMA to transfer data between local and system memory across an interconnect such as PCIe. Attached devices can not see the entire system memory, but only a part of it. Communication between the host processor and attached devices requires an inefficient software stack in comparison to the communication scheme between CPU cores using shared memory. Especially when DRAM memory bandwidth becomes a constraint, requir-

ing extra memory-to-memory copies to move data from one address space to another is cumbersome.

This forced the industry to push for coherency and shared memory across CPU cores and attached devices. This way, accelerators act as peers to the processor cores. The Cell Broadband Engine architecture [59] introduced coherent shared system memory access for its Synergistic Processor Element accelerators. A coherent interface between a CPU and GPU has also been adopted by AMD several years ago with their Accelerated Processing Unit (APU) device family [29]. Another example is the *Cache Coherent Interconnect for Accelerators* (CCIX) [13] which builds on top of PCIe and extends the coherency domain of the processor to heterogeneous accelerators such as FPGAs. OpenCAPI is also a new interconnect standard that integrates coherency and shared memory in their specification. This avoids having to copy data in main memory, and coherency improves FPGA programmability.

With shared memory, the system allows FPGAs to read only a small portion of the data from the host memory without copying the whole block of data to the device memory. This can reduce the total amount of data transmission if the application has a large number of small requests. With coherency supported by hardware, programmers can save effort needed to keep the data coherent through software means. In addition, the shared, coherent address space provided by the coherent interface allows programmers to locally transform a piece of code on the CPU to run on the FPGA, without having to understand the full structure of the program and without having to restructure all references to be local to the FPGA. Especially for production code that tends to be full of statements that are very infrequently executed, the ability to focus on performance-critical code without having to restructure everything is essential. The drawback of supporting shared memory and coherency by hardware is that it requires extra hardware resources and can introduce additional latency. Thus, for performance reasons, a developer might need to optimize the memory controller for special memory access patterns.

5.2.3 Concluding remarks

As discussed in this section, both identified bottlenecks will soon belong to the past. This opens the door for FPGA acceleration again. FPGAs connected using a high bandwidth and low latency interconnect, and ease of programming due to the shared memory programming model, make FPGAs attractive again for database acceleration.

5.3 HBM in FPGAs

As shown in Fig. 4, DRAM bandwidth is not increasing at the same rate as attached devices. Even though the latest DDR4 can provide 25 GB/s bandwidth per *Dual In-line Memory Module* (DIMM), for high-bandwidth applications, a dozen or more modules are required. This leads to a high price to pay in *Printed circuit board* (PCB) complexity and power consumption.

The new high-bandwidth memory technologies provide potential solutions, one of which is high-bandwidth memory (HBM). HBM is a specification for 3D-stacked DRAM. HBM has a smaller form factor compared to DDR4 and GDDR5, while providing more bandwidth and lower power consumption [65]. Because HBM is packaged with the FPGA, it circumvents the use of a PCB to connect to DRAM. The resulting package is capable of multi-terabit per second bandwidth, with a raw latency similar to DDR4. This provides system designers with a significant improvement in bandwidth. The latest generation of Xilinx FPGAs supports HBM within the same package [138], providing FPGAs with close to a half TB/s scale bandwidth (an order of magnitude more bandwidth than the bandwidth to typical on-accelerator DRAM). This makes FPGAs also applicable to data intensive workloads.

Due to the area limitation and the higher cost of stacked DRAMs, HBM integrated with the FPGA can not match the capacity of conventional DRAM. Integration with FPGAs results in a competitive advantage for workloads that require, for example, multiple passes over the same data at high bandwidth. Various examples of multi-pass database queries have been studied in this paper. An example is the sort algorithm presented in Sect. 6.3.

5.4 System with accumulated high bandwidth

Today it is possible to have systems with large storage and accelerator bandwidth. Accelerated database systems can leverage these types of heterogeneous systems. A feasible conceptual system model is depicted in Fig. 5, which is based on the IBM AC922 HPC server [55,89]. Note that the numbers shown in Fig. 5 are peak numbers.

This system consists of two nodes that connect to each other via a *Symmetric multiprocessing* (SMP) interface with 64 GB/s bandwidth in each direction. Each node contains one POWER9 CPU and two FPGAs. Each POWER9 CPU has 170 GB/s bandwidth to DDR4 memory in the host side and supports up to two FPGAs. Each FPGA is connected at 100 GB/s rate through two OpenCAPI channels, meaning in total 200 GB/s accelerator bandwidth is provided in each node. The FPGA fabric would be the latest model VU37P [142], where 8 GB HBM is integrated that supports 460 GB/s bandwidth. Each FPGA would have two OpenCAPI interfaces to the I/O that can be used to attach the NVMe storage. In total, eight of these interfaces in a two-node system support 400 GB/s peak storage I/O bandwidth. FPGAs can be connected to each other via 100 GB/s high-end interfaces. This

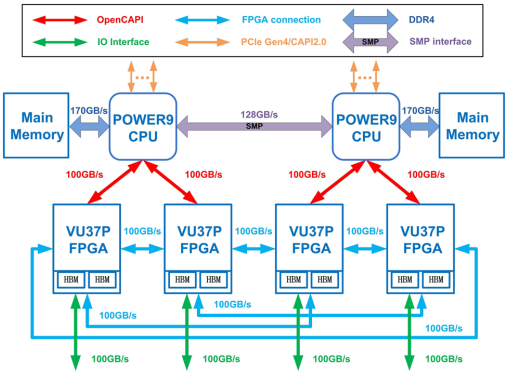


Fig. 5 Proposed FPGA-intensive configuration of POWER9 system (after [89])

example demonstrates that a system with all of the memory bandwidth being available to the accelerators is feasible.

5.5 Programmability trends

Even though the FPGA programmability is a serious and historical challenge that reduces the interest in accelerators for databases from developers, the FPGA community has made great progress and can be expected to keep the current momentum. The number of HLS tools from industry and academia is increasing, and examples include Vivado HLS [38], Altera OpenCL [115], Bluespec System Verilog [93], LegUp [17], DWARV [90], and Bambu [102], etc.

The first step that allowed compiling sequential code in software programming language such as C/C++ into hardware circuits by inserting HLS pragmas and adopting techniques such as loop unrolling, array partitioning, and pipeline mapping [26,28] have proved a milestone contribution. Now these tools are leveraging new techniques to further simplify the programming and enhance the performance. One recent trend is that HLS tools are integrating machine learning techniques to automatically set parameters for performance and controlling resource utilization [75,120,144]. In addition, these techniques can reduce the number of required HLS pragmas which further simplifies FPGA programming in HLS. Another recent change is the support of OpenMP [16,117,134]. OpenMP is one of the most popular languages for parallel programming for shared memory architecture. The support of OpenMP in HLS provides the potential of compiling parallel programs into FPGA accelerators that support shared memory. Because HLS is a big topic study by itself, we can not cover all the aspects. A recent survey [91] comprehensively studies the current HLS techniques and discusses the trends.

The emerging programming framework is another important achievement that contributes to the FPGA programmability, especially for hardware designers. These frameworks help in two different ways. First, such a framework can generate the memory interface for the designers with optimized memory controllers. An example is the SNAP [95] framework for CAPI/OpenCAPI which can take care of the low level communication protocol with the interface and abstract a simple burst mode data request. Another example is the Fletcher [101] framework which can generate interfaces for the Apache Arrow [4] in-memory tabular data format. With these program frameworks, the designer can save time from interface design and focus on the kernel design and performance tuning. Another benefit comes from the support of APIs that can manage the accelerators. These APIs typically wrap up the host accelerator management jobs and communication jobs into a package of software functions. The user only needs to choose and call the right APIs to access the hardware accelerators, which further improves ease of use.

Recently, the above techniques are being applied to the database domain. The prior study in [24,25] gives an overview of how software infrastructure can enable FPGA acceleration in the data center, where the two main enablers are the accelerator generation and the accelerator management. Other work studies SQL-to-hardware compilation. An example is Glacier [86], which can map streaming SQL queries into hardware circuits on FPGAs. Other studies work on the acceleration on database operators such as decompression [73], sort [6], and partitioning [133]. As mentioned before, the frameworks can support FPGA acceleration for databases in two ways, managing the hardware and provide APIs, the Centaur framework [97] is an example of leveraging these ideas for the database domain.

6 Acceleration of query operators

Even though there is not yet a commercial FPGA-accelerated in-memory database, a substantial body of prior work on accelerating database operators or components is pushing progress in this direction. In this section, we summarize the prior work on database operator acceleration including decompression, aggregation, arithmetic, sorts, joins, and others. An overview of the prior work is summarized in Table 2, where FPGA designers and performance analysts can have a quick view on the prior work. We also discuss the potential improvement for operator acceleration, which might be interesting for hardware designers. Most of this work shows that FPGA implementations of the kernels are efficient to the point where performance is limited by the bandwidth from host memory to the FPGAs. In most conventional systems this bandwidth is limited most by the PCIe connection to the FPGA.

Table 2 Summary of database operator acceleration using FPGAs

Operator category	Methods	References	Interface	Bandwidth	Throughput	Data source	FPGA	Frequency
Streaming operator (selection, projection, aggregation, arithmetic, Boolean, etc.)	Median	[84]	DDR*x*	1.6 GB/s	142 MB/s	Off-chip	Virtex-2	100 MHz
	Combined	[31]	PCIe Gen2	2 GB/s	1.13 GB/s	Host	Virtex-6	125 MHz
	Combined	[85]	Ethernet	1 Gbit/s	–	Network	Virtex-5	100 MHz
	Combined	[107]	Ethernet	1 Gbit/s	–	–	Virtex-5	–
	Combined	[124]	PCIe Gen*x*	–	2.31 GB/s	Host	Stratix V	200 MHz
	Combined	[125]	PCIe Gen2	4 GB/s	2.7 GB/s	Host	Stratix IV	200 MHz
	Combined	[126]	PCIe Gen2	4 GB/s	≈4 GB/s	Host	Stratix V	250 MHz
	Rex	[131]	–	–	2 GB/s	–	Virtex-4	125 MHz
	Rex	[113]	QPI	6.5 GB/s	6.4 GB/s	Host	Stratix V	200 MHz
Decompression	RLE	[33]	ICAP	800 MB/s	≈800 MB/s	Off-chip	Virtex-5	200 MHz
	Snappy	[34]	–	–	3 GB/s	–	KU15P	250 MHz
	Snappy	[105]	–	–	0.82 GB/s	–	KU15P	140 MHz
	LZSS	[68]	–	–	400 MB/s	–	Virtex-2	200 MHz
	GZIP/ZLIB	[141]	–	–	3.96 GB/s	–	KU060	165 MHz
	LZW	[74]	–	–	0.5 GB/s (in ASIC)	–	Spartan-3	68 MHz (FPGA)
	LZW	[148]	–	–	280 MB/s	–	Virtex-7	301 MHz
Sort	SN	[87]	–	–	52.45 GB/s	FPGA	Virtex-5	220 MHz
	FMS	[78]	DDR2	–	–	Off-chip	Virtex-5	166 MHz
	FMS	[69]	–	–	2 GB/s	FPGA	Virtex-5	252 MHz
	Merge tree	[69]	–	–	1 GB/s	FPGA	Virtex-5	273 MHz
	Merge tree	[18]	DDR*x*	38.4 GB/s	8.7 GB/s	Off-chip	Virtex-6	200 MHz

Table 2 continued

Operator category	Methods	References	Interface	Bandwidth	Throughput	Data source	FPGA	Frequency
	Merge tree	[79]	–	–	77.2 GB/s	FPGA	Virtex-7	311 MHz
	Merge tree	[118]	–	–	24.6 GB/s	FPGA	Virtex-7	99 MHz
	Merge tree	[119]	–	–	9.54 GB/s	FPGA	Virtex-7	200 MHz
	Merge tree	[145]	–	–	26.72 GB/s	FPGA	VU3P	208 MHz
	Merge tree	[108]	–	–	126 GB/s	FPGA	Virtex-7	506 MHz
	Merge tree	[20]	DDR3	10 GB/s	7.9 GB/s	Off-chip	Virtex-7	250 MHz
Join	SMJ	[18]	DDRx intra-FPGA	115 GB/s 4 GB/s	6.45 GB/s	Off-chip	Virtex-6	200 MHz
	SMJ	[20]	DDR3	3.2 GB/s	0.69 GB/s	Off-chip	Zynq	100 MHz
	Hash join	[110]	DDR3	10 GB/s	–	Off-chip	Virtex-7	200 MHz
	Hash join	[49]	DDR3	76.8 GB/s	12 GB/s	Off-chip	Virtex-6	150 MHz
	Hash join	[50]	–	3 GB/s	18M rows/s	FPGA	Stratix IV	206 MHz
Partitioning	PHJ and Groupby	[21]	DDR3	–	–	Off-chip	Zynq	100 MHz
	Partitioning	[61]	QPI	6.5 GB/s	3.83 GB/s	Host	Stratix V	200 MHz

Operator: *Combined* a combination of multiple streaming operators, *SN* sorting network, *FMS* FIFO merge sorter, *SMJ* sort–merge join, *PHJ* partitioning hash join.
Data Source: *FPGA* the data is generated by FPGAs or stored in FPGA memory, $off-chip$ the data is stored in the off-chip memory in the accelerator side, *host* the data is stored in the host memory side

6.1 Decompression

Decompression is widely used in database applications to save storage and reduce the bandwidth requirement. The decompressor works as a translator, reading a compressed stream consisting of tokens, translating the tokens into data itself, and outputting a decompressed stream. There are many different (de)compression algorithms. Since in database applications we do not want to lose any data, we consider lossless (de)compression algorithms in this survey paper. The most popular two types of (de)compression in database systems are the *Run-Length Encoding* (RLE) [111] and the *Lempel-Ziv* (LZ) series. This paper focuses on decompression algorithms instead of compression algorithms, even though there are many studies [1,11,40] on compression acceleration. An important reason is that in database systems, the common case is to compress the data once and to decompress it more frequently.

6.1.1 RLE

RLE is a simple form of a compression algorithm that records a token with a single value and a counter indicating how often the value is repeated instead of the values themselves. For example, a data sequence "`AAAAAAAABBBC`" after RLE compression is "`8A3B1C`". In this case, instead of storing 12 bytes of raw data, we store 6 bytes, or 3 tokens with each token in fixed size (1 byte counter and 1 byte value). The RLE decompression works in reverse. The RLE decompressor reads a fixed size token, translates it into a variable-length byte sequence, and attaches this sequence to the decompressed data buffer built from the previous tokens.

The method proposed in [33] shows that their FPGA-based RLE implementation can help reduce the FPGA reconfiguration time and achieves a throughput of 800 MB/s which is limited by the *Internal Configuration Access Port* (ICAP) bandwidth. It is not difficult to parallelize this translation procedure. As the token has a fixed size, the decompressor can explicitly find out where a token starts without the acknowledgement of the previous token, and multiple tokens can be translated in parallel. The write address of each parallel processed token can be provided in the same cycle by adopting prefix-sum on the repeating counter. Thus, we can imagine that a multi-engine version of this implementation can sufficiently consume the latest interface bandwidth.

6.1.2 LZ77-based

Instead of working on the word level, LZ77 [149] compression algorithms leverage repetition on a byte sequence level. A repeated byte sequence is replaced by a back reference that indicates where the previous sequence occurs and how long it is. For those sequences without duplicates, the original

data is stored. Thus, a compressed file consists of a sequence of tokens including copy tokens (output the back reference) and literal tokens (output the data itself). During the compression, a history buffer is required to store the most recent data for finding a matched sequence. Similarly, maintaining this history buffer is a prerequisite for copy tokens to copy context from during the decompression. Typically, the size of history buffer is on the order of tens of KB level and depends on the algorithms and their settings.

Decompression translates these two types of tokens into the original data. For literal tokens, the decompressor selects the original data stored in the tokens and writes it into the history buffer. For copy tokens, the back reference data including the copied position and copied length is extracted, followed by a read from the history buffer and a write to the history buffer. There are many extensions to this algorithm, e.g., LZ4 [22], LZSS [122], Gzip[2] [32] and Snappy [44].

In an FPGA, the history buffers can be implemented using shift registers [68] or BRAMs [54], and the token decoding and the BRAM read/write can be placed in different pipeline stages. While pipeline design can ensure continuous processing of the compressed data, the throughput declines when data dependencies occur. The LZ4 decompression proposed in [77] uses separate hardware paths for sequence processing and repeated byte copying/placement, so that the literal tokens can always be executed since they contain the original data and are independent of the other tokens. Separating the paths ensures these tokens will not be stalled by the copy tokens. A similar two-path method for LZ77-based decompression is shown in [46], where a slow-path routine is proposed to handle large literal tokens and long offset copy tokens, while a fast-path routine is adopted for the remaining cases. This method is further demonstrated at the system level in [45] to hide the latency of slow operations and avoid stalls in the pipeline.

Even though a single-engine FPGA implementation can outperform a CPU core, it is not easy to exceed a throughput of one token per cycle per engine. To saturate the bandwidth from a high-bandwidth connection, we can either implement multiple decompressor engines in an FPGA or implement a strong engine that can process multiple tokens per cycle. A challenge of implementing multiple engines is the requirement of a powerful scheduler that can manage tens of engines, which also drains resources and might limit the frequency. In addition, the implementation of the LZ77-based decompressor in FPGAs takes much more memory resources [141], especially the BRAMs, limiting the number of engines we can place in a single FPGA. Apart from that, the unpredictable block boundaries in a compressed file also bring challenges to decompressing multiple blocks in parallel [58]. As an alternative, researchers also look for intra-block parallelism. However, the demands of processing multiple tokens in parallel pose challenges including handling the various token sizes, resolving the data dependencies and BRAM bank conflicts. A parallel variable length decoding technique is proposed in [2] by exploring all possibilities of bit spill. The correct decoded streams among all the possibilities are selected in a pipelined fashion when all the possible bit spills are calculated and the previous portion is correctly decoded. A solution to the BRAM bank conflict problem is presented in [105] by duplicating the history buffer, where the proposed Snappy decompressor can process two tokens every cycle with throughput of 1.96 GB/s. However, this method can only process up to two tokens per cycle and is not easy to scale up to process more tokens in parallel due to the resource duplication requirement. To reduce the impact of data dependencies during the execution of tokens, Sitaridi et al. [116] proposed a multiround execution method that executes all tokens immediately and recycles those copy tokens that return with invalid data. The method proposed in [34] improves this method to adopt the parallel array structure in FPGAs by refining the tokens into BRAM commands which achieve an output throughput of 5 GB/s.

For LZ-77-based decompression accelerators that need a history buffer (e.g., 64 KB history for Snappy), a light engine that processes one token per cycle would be BRAM limited, while a strong engine that processes multiple token per cycle might be LUT limited. Even the design [34] with the best throughput cited in this paper is not in perfect balance between LUTs and BRAMs for the FPGA it uses, and there is room for improvement.

6.1.3 Dictionary-based

Dictionary-based compression is another commonly used class of compression algorithms in database systems, the popular ones of which are the LZ78 [150] and its extension LZW [135]. This class of compression algorithms maintains a dictionary and encodes a sequence into tokens that consist of a reference to the dictionary and the first non-matched symbol. Building the dictionary lasts for the whole compression. When the longest string matches the current dictionary, the next character in the sequence is appended to this string to construct a new dictionary record. It is not necessary to store the dictionary in the compressed file. Instead, the dictionary is reconstructed during decompression.

A challenge to designing efficient decompression in FPGAs is to handle the variety of string length in the dictionary. When adopting fixed-width dictionaries, while setting a large width for the string wastes a lot of memory space, using a small string width suffers from throughput decrease since multiple small entries must be inspected to find the match. Thus, a good design for these decompression algorithms demands explicit dictionary mechanisms that can efficiently

[2] Gzip is an implementation of DEFLATE [62].

make use of the FPGA's capability of bit-level processing. A two-stage hardware decompressor is proposed in [74] which combines a parallel dictionary LZW with an Adaptive Huffman algorithm in a VLSI, achieving 0.5 GB/s data rate for decompression. The study in [148] presents an efficient LZW by storing the variable-length strings in a pointer table and a character table separately. The implementation of a single instance of this algorithm consumes 13 18 Kb BRAMs and 307 LUTs in an XC7VX485T-2 FPGA, achieving 300 MHz frequency and 280 MB/s throughput.

6.1.4 Discussion

Memory access patterns Table 3 compares the host memory access patterns of operators discussed in this survey, assuming that the source data is initially stored in the host memory. The decompression has a "sequential" memory access pattern since it has streaming input and streaming output. Typically the decompressor outputs more data then the input.

Bandwidth efficiency As we mentioned before, the RLE algorithms can easily achieve high throughput that can meet the interface bandwidth bound, but the LZ77 and LZ78 series are challenging due to the large number of data dependencies. According to our summary, most of the prior work can not reach the latest accelerator interface bandwidths, but some designs [34] can. This depends on many factors including the algorithm itself, hardware design and its trade-offs, including LUT-to-BRAM balance, and FPGA platforms. For a multi-engine implementation, the throughput is defined by the product of throughput per engine and the number of engines. The challenge is that in an FPGA, we can either have strong decompression engines but fewer of them or more less powerful engines. Thus, a good trade-off during design time is indispensable to match the accelerator interface bandwidth.

6.2 Streaming operators

6.2.1 Streaming operators

Streaming operators are database operations where data arrives and can be processed in a continuous flow. These operators might belong to different categories of database operators such as selections [85,107,124,125,139], projections [85,107,124,126], aggregations (sum, max, min, etc.) [31,84,86,97], and regular expression matching [113,131]. We place them together because they typically act as preprocessing or post-processing in most of the queries and have similar memory access patterns.

Projections and selections are filtering operations that only output the fields or records that match the conditions, while the aggregations are performing arithmetic on all the inputs. Due to the pipeline style design in FPGAs, these operations can be performed in a stream processing model in the FPGA implementation, at high bandwidth and with low overall latency. A multi-engine design of these operators, by adopting parallelism at different levels, can easily achieve throughput that may exceed the accelerator interface bandwidth and even get close to the host memory bandwidth. Regular expression matching can be used to find and replace patterns in strings in databases, such as "REGEXP_LIKE" in SQL. The performance of regular expression matching is bounded by the computation in software due to the low processing rate of software deterministic finite automaton. However, it can be mapped to custom state machines in the FPGA and gain performance from a pipelined design.

Even though the implementation of these kernels in FPGAs is trivial, acceleration of the combination of these operators and other operators is non-trivial. Typically, a query is executed according to a query plan that consists of several operators. In software, the operator order is decided and optimized by the query compiler and the query optimizer. Choosing an order of these operators can reduce the amount of data running in the pipeline and avoid unnecessary loads and stores of the intermediate results. Conversely, an irrational query plan can cause extra data accesses and waste the communication resources. Similarly, to achieve high throughput, the consideration of combining different FPGA-implemented operators in a reasonable order is indispensable. There are many methodologies to optimize the query order, a basic one of which is filtering data as early as possible. This idea has been reported in many publications. For instance, the implementation in [124] executes projections and selections before sorting. The compiler proposed in [85] supports combinations of selections, projections, arithmetic computation, and unions. The method from [107] allows joins after the selection and the projection.

6.2.2 Discussion

Memory access pattern The streaming operators have streaming reads and streaming write or can only write a single value (such as sum). They typically produce less output data compared to the input, especially for aggregation operators that perform the arithmetic. In other words, the streaming operators have sequential access patterns to the host memory.

Bandwidth efficiency A single engine of the streaming operators can easily reach GB/s or even tens of GB/s magnitude throughput. For example, the sum operation proposed in [31] shows a throughput of 1.13 GB/s per engine, which is limited by the connection bandwidth bound of their PCIe x4 Gen 2 connected platform. A combination of decompression, selection, and projection presented in [124] reports a kernel processing rate of 19 million rows/s or 7.43 GB/s (amplified by decompression) which exceeds the bandwidth of PCIe used in their reported platform. Since these opera-

Table 3 Summary of memory access in different operators

Operator category	Methods	Host memory access pattern	In-memory intermediate result	Multi-pass host memory access
Streaming operator	Selection	Sequential	No	No
	Projection			
	Aggregation			
	Arithmetic			
	Boolean			
	Rex			
Decompression	RLE	Sequential	No	No
	LZ77-based			
	Dictionary-based			
Partitioning	Partitioning	Scatter	No	No
	Multi-pass partitioning	Scatter	Yes	Yes
Sort	Sorting network	Sequential	No	No
	FIFO merge sorter	Streaming gather	No	No
	Merge tree	Streaming gather	No	No
	Multi-pass merge tree	Streaming gather	Yes	Yes
	Partitioning sort	Scatter sequential	Yes	Yes
Join	Sort–merge join	Patterns of sort streaming gather	Yes	Yes
	Hash Join (Small Dataset)	Sequential	No	No
	In-memory hash join	Scatter (build) random (probe)	Yes	No
	Partitinging hash join	Scatter sequential	Yes	Yes

For some aggregation and arithmetic algorithms, it only requires a single value as the result to write back
Streaming Gather is similar to Gather but from multiple streams, and the next read depends on the current state

tors do not require resource-intensive functions in FPGAs, an instance of multiple engines can be easily implemented in a single FPGA to achieve throughput that is close to the interface bandwidth. Although multiple engines need to read from and write to different streams, this won't increase the data access control since the streams are independent. Thus, a theoretically achievable throughput upper bound of these operators is the accelerator interface bandwidth.

6.3 Sort

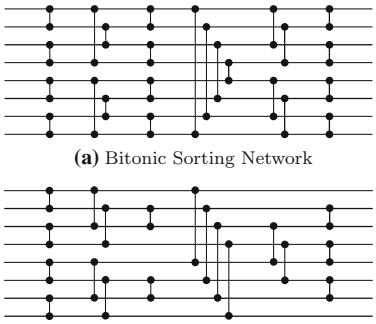
Sorting is a frequently used operation in database systems for ordering records. It is can be used in *ORDER BY* in SQL and in a more complex query to improve the performance. Large scale sort benchmarks are considered key metrics for database performance.

In a CPU-based sort, the throughput is limited by the CPU computation capacity, as well as the communication between computational nodes. For the record holder for a large multi-node sort the per node performance is about 2 GB/s [94]. Single-node sort throughput without the communication overhead may be somewhat larger, but we believe that the single-node performance would be within the same order of magnitude. As network speed increases rapidly, and storage is replaced with NVMe devices where storage bandwidth grows rapidly, sort with CPUs is not going to keep up. Therefore, accelerators are now required for this operation that historically was bandwidth-bound. To reach this goal, many hardware algorithms have been presented using FPGAs to improve performance. In this section, we give an overview of the prior FPGA-based sort algorithms.

6.3.1 Sorting network

A sorting network is a high throughput parallel sort that can sort N inputs at the same time. The compare-and-swap unit is the core element in a sorting network. A compare-and-swap unit compares two inputs and arranges them into a selected order (either ascending or descending), guaranteed by swapping them if they are not in the desired order. Using a set of these compare-and-swap units and arranging them in a specific order, we can sort multiple inputs in a desired order.

A simple way to generate a sorting network is based on the bubble sort or insertion sort algorithm. Thus, these types of sorting networks require $O(N^2)$ compare-and-swap units and $O(N^2)$ compare stages to sort N inputs. More efficient methods to construct the network include the bitonic sorting network and the odd-even sorting network [12]. Figure 6 shows the architecture of the bitonic sorting network (Fig. 6a) and the odd-even sorting network (Fig. 6b) with 8 inputs. We can further pipeline the designs by inserting registers after each stage. Knowing that it takes one cycle for a signal to cross one stage in a pipeline design, both sorting networks take $O(\log^2 N)$ cycles to sort N inputs, while the space complexity is $O(N \log^2 N)$.



(a) Bitonic Sorting Network

(b) Odd-Even Sorting Network

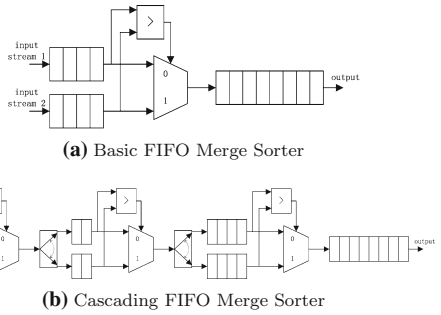
Fig. 6 Architecture of sorting network

A sorting network can sort multiple data sets concurrently by keeping different data sets in different stages. An N-input sorting network is able to process N elements per FPGA cycle. The sorting network proposed in [87] outputs 8 32-bit elements per cycle at 267 MHz, meaning a throughput of 7.9 GB/s. It is not difficult to increase the throughput by scaling up the sorting network for a larger input number. A 64-input sorting network at 220 MHz based on the implementation in [87] can consume data at 52.45 GB/s, approximately equivalent to the bandwidth of two OpenCAPI channels (51.2 GB/s). However, the required reconfigurable resources increase significantly with the increase in the number of inputs. Thus, a sorting network is generally used for the early stages of a larger sort to generate small sorted streams that can be used as input for the FIFO merge sort or merge tree in the later stages.

6.3.2 FIFO merge sorter

The *first-in first-out* FIFO merge sorter is a sorter that can merge two pre-sorted streams into a large one. The key element is the select-value unit. It selects and outputs the smaller (or larger) value of two input streams. The basic FIFO merge sorter is illustrated in Fig. 7a. Both inputs are read from two separate FIFOs that store the pre-sorted streams, and a larger FIFO is connected to the output of the select-value unit. In [78], an unbalanced FIFO merge sorter is proposed which shares the output FIFO with one of the input FIFOs to save FIFO-resources. The proposed architecture is able to sort 32K 32-bit elements at 166 MHz, consuming 30 36 Kb blocks (30 out of 132 from a Virtex-5 FPGA).

A drawback of the FIFO merge sorter is that it takes many passes to merge from small streams to the final sorted stream since it only reduces the number of streams into half each pass, especially when handling large data sets that have to



(a) Basic FIFO Merge Sorter

(b) Cascading FIFO Merge Sorter

Fig. 7 Architecture of FIFO merge sorter

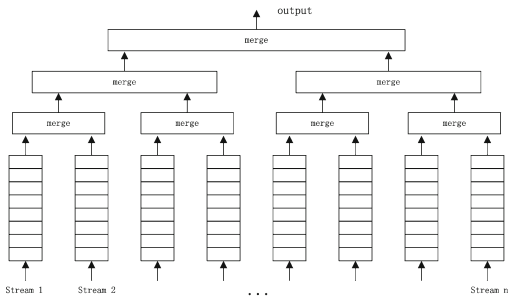


Fig. 8 Architecture of merge tree

store into host memory. Sorting a data set with 1024 small pre-sorted streams requires 10 passes, which means the data needs to travel between the memory and the FPGA multiple times. In this case, the overhead of the data transmission dominates, and the overall performance is limited by the interface bandwidth. This problem can be solved by cascading multiple FIFO merge sorters. As shown in Fig. 7b, FIFO merge sorters with smaller FIFOs are placed in the earlier stages, while those with larger FIFOs are inserted at later stages. An improved method is presented in [69] where the proposed cascading FIFO merge sort can reduce the pipeline filling time and emptying time by starting to process the merge as long as the first element of the second FIFO has arrived. This implementation can sort 344 KB of data at a throughput of 2 GB/s with a clock frequency of 252 MHz. However, in the cascading FIFO merge sorter, the problem size is limited by the FPGA internal memory size, since it needs to store all the intermediate results of each merge sort stage.

6.3.3 Merge tree

For data sets that do not fit in the internal FPGA memory, we need merge trees. The merge tree can merge several sorted streams into one larger stream in one pass. As shown in Fig. 8, a merge tree is constructed by a set of select-value units arranged in multiple levels. In each level, the smaller elements between two streams are selected which will be sent to the next level to select the smallest among these four streams. This process is iterated until the largest element among all the input streams is selected. FIFOs are inserted between the leaves of the tree and the external memory to hide the memory access latency. To pipeline the whole merge and reduce the back pressure complexity from the root, small FIFOs are placed between the adjacent levels. An M-input merge tree merges N streams into $\frac{N}{M}$ streams each pass. Compared to a FIFO merge sorter, it reduces the number of host memory accesses from $\log_2 N$ to $\log_M N$. The merge tree implementation in [69] provides a throughput of 1 GB/s to sort 4.39 million 64-bit elements. Partial reconfiguration was used to configure a three-run sorter (one run of the FIFO merge sorter and two runs of the tree merge sorter) that can handle 3.58 GB of data and achieve an overall throughput of 667 MB/s.

For a multi-pass merge sort, even though we can place multiple merge trees to increase the system throughput, the final pass has to guarantee all the remaining streams are combined into a single sorted stream. This demands a stronger merge tree that can output multiple elements per cycle. The study in [18] presents a multi-element merge unit. This unit compares N elements from the selected stream with N feedback elements produced in the previous cycle and selects the smallest N ones as output, while the remaining N elements will be fed back for the next cycle comparison. The proposed merge tree achieves a throughput of 8.7 GB/s for a two-pass merging which reaches 89% utilization of the interface bandwidth.

A. Srivastava et al. [119] proposed a multi-output merge tree architecture by adopting increasing-in-size bitonic merge units (BMUs). In this architecture, the BMU in the leaf level compares the first element from both input streams and outputs one element per cycle. The next level BMU compares the first two elements from both streams and outputs two elements per cycle. The size of the BMU doubles from one level to the next level until it reaches the root. As a small BMU is much more resource efficient than a large one, as it saves hardware resources in the FPGA, bringing high scalability. A drawback of this method is that the throughput for skewed data drops. A similar architecture [118] where the merge tree is able to output 32 64-bit elements in one cycle reports a throughput of 24.6 GB/s. However, this paper uses a wrapper to produce the input data instead of reading from the external memory, where the control of feeding 32 input streams would consume internal memory of the FPGA as well as potentially reducing the clock frequency. A clock frequency optimization method is illustrated in [79] by deeply pipelining the multi-element merge unit. An instance of this method with 32 outputs per cycle operates at 311 MHz. The

method proposed in [108] further raises the frequency of the merge tree to 500 MHz by breaking the feedback loop.

Based on the observation that it might consume all the BRAM resources in an FPGA to hide the interconnect latency in a wide merge tree, an alternative is presented [145] of an "odd-even" merge sorter combined with a multi-stream interface [88] that can deal with the data skew and supply a stable throughput for any data distribution. The multi-stream interface is constructed as a two-level buffer using both the BRAM and URAM resources. The evaluation shows that the "even-odd" merge sorter kernel can merge 32 streams, providing a throughput of 26.72 GB/s. However, even with the two-level buffer solution, the buffer and the associate logic still require a majority of the FPGA resources.

Even though speedups are gained from the FPGA acceleration, we have to note that the maximum throughput is not going to exceed the interface bandwidth divided by the passes of memory accesses. Thus, a good way to design a sorter needs to trade off between the number of sorting passes and the throughput of each pass.

6.3.4 Discussion

Memory access pattern Different sort algorithms have different memory access patterns. An N-input sorting network typically is used in an early stage of sorting to convert an input stream into an output stream that is a sequence of N-element sorted substreams. The FIFO merge sort has two sorted streams as inputs that will be merged into a larger sorted stream. However, the two inputs are not independent from each other. This is because the next data to be read relies on the comparison result of the current inputs. Since this access pattern is similar with the pattern of "gather" (indexed reads and sequential writes), but the next reads depend on the current inputs, we refer to this as a streaming gather pattern. Thus, the FIFO merge sort has dependent multi-streaming read and streaming write memory access patterns. Similarly, the merge tree has the same access pattern as FIFO merge sort, with the next input might come from multiple streams instead of two.

Bandwidth efficiency All three classes of sort methods mentioned above can sufficiently utilize the interface bandwidth by making stronger engines or deploying more engines. However, a large sort may require multiple passes that each requires the host memory to be accessed. In this manner, the overall throughput of an entire sort depends on both the number of passes and the throughput of each pass, or the overall throughput does not exceed the bandwidth divided by the number of passes. The number of passes can be reduced by a wider merger that merges more streams into one larger stream. However, because each input stream requires buffering, building up a wider merge tree requires a lot of BRAMs, which makes the design BRAM limited.

Buffer challenge Reducing the number of memory access passes is a key point to improve the sort throughput. One way to do this is to sort as much data as possible in one pass. For example, use a wider merge tree to merge more streams. However, in an FPGA, hiding the host memory latency for multiple streams is a challenging problem, let alone for the dependent multi-streaming accesses. In a merge tree, when merging multiple streams, which stream is chosen next cannot be predicted. Even though buffers can be placed at the inputs of each stream, naively deploying buffers for each input to hide the latency would consume all the BRAMs in an FPGA or even more. In the case of a 64-to-1 merge tree that can output 8 16B elements each cycle, to hide the interconnect latency (assume 512 FPGA cycles in a 250 MHz design), 4 MB of FPGA internal memory resources are demanded (which is all the BRAM resources for a Xilinx KU15P).

HBM benefit HBMs on the FPGA card bring the potential to reduce the number of host memory accesses for sort. It provides accelerators with larger bandwidth and delivers comparable latency compared to the host memory. Thus, instead of writing the intermediate results back to the host memory, we can store them in the HBMs. For a data set that fits in an HBM, the analysis from [145] illustrates that it only demands one pass read from and write to the host memory with the help of HBM (the read in the first pass and the write in the final pass), while without HBM it requires five passes access to the host memory. For data sets larger than the HBM capacity, using HBMs for the first several passes of sorting can reduce a remarkable number of host memory accesses.

Partitioning methods The number of host memory accesses goes up fast once the data set size is larger than the HBM capacity. For example, using a 32-to-1 merge tree to sort 256 GB data in an FPGA equipped with 8 GB HBM demands two passes, including one pass to generate 8 GB sorted streams and one pass to merge these 8 GB streams into the final sorted stream. However, with every 32 times increase in the data set size, an extra pass is required.

The partitioning method is a solution to reduce the host memory accesses and to improve the memory bandwidth efficiency. It was previously used to obtain more concurrency by dividing a task into multiple sub-tasks. Another benefit is that it can enhance the bandwidth efficiency.

For sorting large data sets in GPUs, a common method is the partitioning sort [39] where a partitioning phase is executed to partition the data set into small partitions and a follow-up sort phase to sort each of them. This method only demands two passes of host memory accesses and can achieve a throughput of up to one-fourth of the host memory bandwidth (two reads and two writes). Experimental results in [39] illustrate a throughput of 11 GB/s in a single GPU node which is 65% of the interface bandwidth.

This method is feasible in FPGAs if the HBM is integrated. In this way, to sort the whole data set, two passes

accessing the host memory is sufficient. In the first pass, the data is partitioned into small partitions that fit in HBM, and write the partitions back to the host memory. The second pass then reads and sorts each partition and writes the sorted streams back. As this method needs fewer host memory accesses compared to the merge sort, it is interesting to study the partitioning sort in FPGAs and compare it with the multi-pass merge tree method. The partitioning methods are also applicable to hash joins to enhance the memory bandwidth efficiency. This is described in detail in Sect. 6.4.

6.4 Join

Joins are a frequently used operation in database systems, that combine two or more tables into one compound table under specific constraints. The most common one is the equi-join that combines two tables by a common field. In the rest of this paper, we refer to equi-joins as joins. There are many different join algorithms including *nested loop join*, *hash join* (HJ), and *sort–merge join* (SMJ). In this paper, we focus on the HJ and SMJ since they are more interesting to the database community due to their low algorithmic complexity.

6.4.1 Hash join

Hash join is a linear complexity join algorithm. It builds a hash table from one of the two join tables and uses the other table for probing to find matches. The probing time for each element remains in constant time if a strong hash function is chosen. In CPUs, it is difficult for a hash function to have both strong robustness and high speed. In FPGAs, this trade-off is broken because FPGAs allow a complex algebra function implemented in a circuit that calculates faster than the CPU does. Kaan et al. [60] shows a murmur hash FPGA implementation that achieves high performance as well as strong robustness. Research from [67] points out that the indexing takes up most of the time for index-based functions, especially the walk time (traversal of the node list) which accounts for 70%. To solve this problem, the *Widx* ASIC, an index traversal accelerator tightly connected to the CPU cores, is proposed to process the offloaded index-based walker workloads. Widx decouples the hash unit and the walker (the traversal unit) and shares the hash unit among multiple walkers to reduce the latency and to save the hardware resources.

One of the key points to design an FPGA-based hash join algorithm is to have an efficient hash table structure. On one hand, the hash table structure influences the performance of the hash join engine. An inappropriate hash table design might introduce stalls, reducing the throughput. On the other hand, as there is a limited number of BRAMs inside an FPGA, to ensure a multi-engine instance is feasible in one FPGA, a hash table should not consume too many BRAMs. The method in [130] makes use of most of the BRAMs in an FPGA to construct a maximum size hash table. In this method, the BRAMs are divided into groups. The hash table connects the BRAM groups into a chain, and different hash functions are adopted for different BRAM groups. In the case of hash collisions, conflicting elements will be assigned to the next group until the hash table overflows. Even so, due to the skewed data distribution, some of the memory may be wasted. The hash join in [50] uses two separate tables to construct the hash table, including one *Bit Vector* table storing the hash entries and one *Address Table* table maintaining the linked lists. The proposed architecture allows probing without stalls in the pipeline.

For data sets that are too large to store the hash table in the FPGA's internal memory, in-memory hash joins are needed. As the BRAMs can only store part of the hash table, probing tuples demands loading the hash table multiple times from the main memory. However, the latency of accessing main memory is much larger than that of accessing BRAMs. One way to deal with this problem is to use the BRAMs as a cache [110]. However, to achieve high throughput, an efficient caching mechanism is required. If the size of the hash table is much larger than the BRAMs, the cache miss ratio might remain too large to benefit from the cache system. Another way to hide the memory latency is to use multi-tasking. As FPGAs have a large amount of hardware resource for thread states, we can keep hundreds of tasks in an FPGA. An example is shown in [49] where the hash join runs against a Convey-MX system and achieves a throughput of up to 12 GB/s or 1.6 billion tuple/s. However, when applying this method to other architectures, we need to avoid suffering from the granularity effect [36]. Because each access to the main memory must obey the access granularity that is defined by the cache line, every read/write request always acquires the whole cache line(s) of data. If a request does not ask data that covers the whole cache line(s), part of the response data is useless, meaning a waste of bandwidth. Thus, the hash table needs to be properly constructed to reduce or avoid the occurrence of this situation.

Another way to process in-memory hash joins is to partition the data before performing the joins. Both the input tables are divided into non-intersecting partitions using a same partition function. One partition from a table only needs to join with the corresponding partition in the other table. If the hash table of a partition is small enough to fit in the internal FPGA memory, the hash table is required to be loaded only once. The main challenge is how to design a high throughput partitioner. The study in [140] presents a hardware-accelerated range partitioner. An input element is compared with multiple values to find a matched partition, after which this element is sent to the corresponding buffer. In this work, deep pipelining is used to hide the latency of multiple value comparisons. Kaan et al. [61] proposed a hash partitioner that can contin-

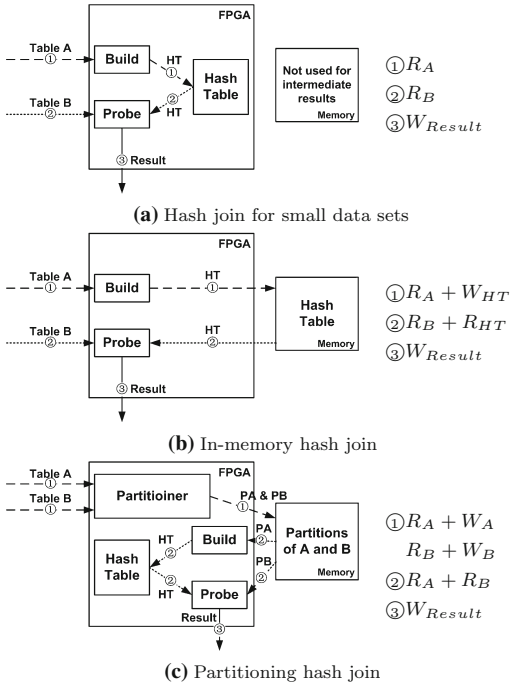


Fig. 9 Data flow regarding main memory accesses in different hash join algorithms. R_x stands for reading x, while W_x stands for writing x. Note that the number of writes to the hash table (W_{HT}) is based on the size of Table A, and the number of reads from the hash table (R_{HT}) is based on the size of Table B

uously output a 64B cache line in a 200 Mhz FPGA design. Write combiners are used to construct a full cache line output to avoid granularity effects [36]. To keep up with the QPI bandwidth, the authors implement multiple engines in an FPGA. Their end-to-end example shows around 3.83 GB/s for partitioning the data in an FPGA and 3 GB/s for a hybrid partitioning hash join (partition in the FPGA and join in the CPU). If the data set is too large and the partition size does not fit in the FPGA's internal memory, a multi-pass partitioning can be adopted. An example is demonstrated in [21] where *LINQits* is proposed to accelerate database algorithms including the hash join and the group-by operator.

6.4.2 Sort–merge join

The sort–merge join is comparable to the hash join algorithm. It first sorts both tables and does a merge step afterward. The most challenging part of a hardware sort–merge join is the sort itself which we have discussed in Sect. 6.3. To further improve the sort–merge join performance, the merge step can be started as long as the first sorted element from the second table is output. The sort–merge join proposed in [18] adopts a similar idea and achieves a throughput of 6.45 GB/s for the join. Ren et al. [20] proposed a sort join in a heterogeneous CPU-FPGA platform. It performs the first few sorting stages in FPGAs and streams the partial results to the CPUs for a later merge sort step, and the merge join afterward.

Others study the comparison between the hash join and the sort–merge join. This topic has been well studied for the CPU architecture [3,8,64], but not too much for FPGA-based accelerators. The work in [130] studies the FPGA-based hash join and the FPGA-based sort–merge join, and claims that the sort–merge join outperforms the hash join when the data sets become larger. However, the in-memory hash join is not included in this comparison which is more suitable for larger data sets. A detailed analysis and comparison is explained in [8] on a multi-core CPU platform. According to the experimental results, the optimized hash join is superior to the optimized sort–merge join, while for large data sets the sort–merge join becomes more comparable. Even though this analysis is based on the CPU architecture, we believe that the principles of the analysis are similar and it would be a good guideline for further analysis based on the FPGA architecture.

6.4.3 Discussion

Memory access pattern Typically hash joins have a streaming read pattern for reading both tables and a streaming write pattern for writing the results back, but accessing the hash table, including establishing the hash table and probing it, is random. If the data set is small enough that the hash table can be stored in the accelerator memory, accessing the hash table becomes internal memory accesses. Thus, from the host memory access aspect, hash joins for small data sets only have streaming read and streaming write patterns. Partitioning a large data set into small partitions before doing the joins allows the hash table to be stored in the accelerator memory, which can avoid the random access to the host memory. The extra memory access passes introduced by the partitioning phase hold scatter patterns (read from sequential addresses and write to indexed/random addresses).

For sort–merge join, the memory access pattern of sorting the two tables is studied in Sect. 6.3.4. Similar with the FIFO merge sorter, the join phase reads both sorted tables in dependent streaming ways and writes the results back as a single stream. Accordingly, the sort–merge join has the same memory access pattern as the sort, plus the streaming gather memory access pattern.

Bandwidth efficiency The data flows of different types of hash joins are illustrated in Fig. 9. The memory shown in the figure is the host memory. However, the data flows can also

be applied in the off-chip memory on the accelerator side. Hash joins on small data sets are streaming-like operations. Both tables are read from the host memory, while the hash table is stored in the FPGA internal memory. Thus, a hash join requires a pass reading the data and a pass writing the results back. As hash joins are not highly computational, this streaming read and streaming write can saturate the accelerator interface bandwidth.

For large data set cases, the hash table needs to be stored in host memory. During the build phase, each tuple in the build table needs to write the hash table. Similarly, each tuple in the probe table during the probe phase generates at least one read to the hash table. Consequently, hash joins on large data sets equivalently demand two passes of host memory accesses for the original tables and the hash table, and one pass writing the results back. However, accessing the hash table results in random access patterns, which suffer performance decrease due to cache misses [14], TLB misses [9,10], the *Non-uniform memory access* (NUMA) effect [71], and the granularity effect [36], etc. The study in [36] shows that only 25% of the memory bandwidth is effective during the access to the hash table in a 64B cache line machine if the tuple size is 16B.

The partitioning hash join can avoid random access by splitting the data into small partitions such that the hash table for each can be stored in the FPGA memory. One drawback of this method is that the partitioning phase introduces extra memory accesses. The cost of partitioning even becomes dominating if multi-pass partitioning is inevitable. However, the partition can be implemented in a streaming processing way, which has streaming read and multi-streaming write patterns. For a one-pass partitioning hash join, two passes of streaming read and streaming write to the host are required. Thus, the throughput can reach up to one-fourth of the bandwidth.

HBM benefit The HBM technology can be adopted in the hash joins as well. For non-partitioning hash joins, HBM can be used as caches or buffers, leveraging the locality and hiding the latency of host memory accesses. It also allows smaller granularity access than DRAM, and therefore can reduce the granularity effect.

For partitioning hash joins, HBMs can help in two ways. One way is to use HBMs to store the hash table of each partition, which allows larger size partitions and can reduce the required number of partitions. The other way is to use HBMs as buffers to buffer the partitions during the partitioning phase, which provides the capability of dividing the data into more partitions in a pass. Both methods can reduce the number of partitioning passes, leading to fewer host memory accesses.

7 Future research directions

In this section we discuss future research directions as well as their challenges for different groups of researchers including database architects, FPGA designers, performance analysts, and software developers.

7.1 Database architecture

We believe that in-memory databases should be designed to support multiple types of computational elements (CPU, GPU, FPGA). This impacts many aspects of database design:

- It makes it desirable to use a standard data layout in memory (e.g., Apache Arrow) to avoid serialization/deserialization penalties and make it possible to invest in libraries and accelerators that can be widely used.
- It puts a premium on a shared coherent memory architecture with accelerators as peers to the host that can each interact with the in-memory data in the same way, thus ensuring that accelerators are easily exchanged.
- It implies we need to develop a detailed understanding of which tasks are best executed on what computational element. This must include aspects of task size: which computational element is best is likely to depend on both task type and task size.
- As noted earlier in the paper, new query compilers will be required that are based on this understanding that can combine this understanding with the (dynamically varying) available resources.

This may seem like a momentous task, but if we assume we are in a situation where all elements have equal access to system memory, it may be possible to build some key operations like data transformations in FPGAs and derive an early benefit.

In addition to shared-memory-based acceleration, there are additional opportunities for near-network and near-(stored-)data acceleration for which the FPGAs are a good match. So in building architectures for in-memory databases, it is also important to keep this in mind.

7.2 Accelerator design

As new interfaces and other new hardware provide new features that break the accelerator-based systems balance, more opportunities arise for FPGAs to accelerate database systems.

As noted in this paper, for several key operators FPGA-based implementation exist that can operate at the speed of main memory. However, less work has been done on designs that leverage both the CPU and FPGA or even a combination of CPUs, FPGAs, and GPUs.

Another opportunity brought to FPGAs is the emerging machine learning workloads in databases. Database systems are extending the support of machine learning operators [76]. In addition, the databases are integrating machine learning methods for query optimization [132]. In both cases, the heavy computation required poses computational challenges to the database systems, and FPGAs can likely help. Thus, new accelerators for these emerging workloads are worth studying.

Lastly, from a system perspective, a single FPGA can only provide limited performance enhancement, while a multi-FPGA architecture should bring remarkable speedup. On the one hand, different functions can be implemented in different FPGAs to enable a wider functionality and more complicated workloads. One the other hand, multiple FPGAs can work together for a single job to enhance the performance. Using FPGAs as a cloud resource can further increase the FPGA utilization rate in both clouds and in distributed database systems [19,103]. Important challenges related to this topic include how to connect the FPGA with CPUs and other FPGAs, and how to distribute the workloads.

7.3 Comparison with GPUs

GPUs have now become a serious accelerator for database systems. This is because the GPU has thousands to ten thousands of threads, which is a good choice for throughput-optimized applications. Besides the high parallelism, being equipped with HBM allowing large capacity internal memory and fast speed to access it is another enabler. While FPGA now is also integrating HBM that makes it more powerful, the parallelism in FPGAs does not typically provide the same order of magnitude of threads in GPUs. Thus, a question we would like to ask ourselves is what kinds of database applications can work better on FPGAs than GPUs.

The first possible answer is the latency-sensitive streaming processing applications such as network processing. The request of both high throughput and low latency in the streaming processing presents challenges to GPUs. Since GPUs need to process data in a batch mode with well-formatted data for throughput gains, this formatting might introduce additional latency. However, this requirement can perfectly meet the features of the FPGA with data flow designs where format conversion is often free. The ability to do efficient format conversion might also apply to near storage processing such as Netezza [41] to relieve the pressure for CPUs and networks. Further improvement can include new features to meet the emerging workloads such as supporting format transformation between the standardized storage formats (e.g., Apache Parquet [5]) and the standardized in-memory formats (e.g, Apache Arrow [4]).

In addition, FPGAs may even beat GPUs in terms of throughput in specific domains that require special functions. For example, if one wants to build databases that operate under an encryption layer, it might need to combine encryption with other operators. FPGAs may be efficient at implementing these special functions in hardware and FPGAs often suffer only suffer negligible throughput impact by combining two function components in a pipeline. These special functions also include provisions for database reliability, special data types like geo-location data processing, or data types requiring regular expression matching.

We can also think about exploring more heterogeneity of building a database system with a CPU-GPU-FPGA combination. Different processors have different features and different strong points. CPUs are good at complex control and task scheduling, GPUs work well in a batch mode with massive data-level parallelism, and FPGAs can provide high throughput and low latency using data flow and pipeline design. Combining these processors together in one system and assigning the workloads to the processors that fit best may well deliver an advantage. However, some challenges need to be addressed to build this system including how to divide tasks into software and hardware, how to manage the computational resources in the system, and how to support data coherency between different types of processors.

7.4 Compilation for FPGAs

The FPGA programmability is always one of the biggest challenges to deploying FPGAs in database systems. To reduce the development effort and the complexity for FPGA-based database users, a hardware compiler that can map queries into FPGA circuits is necessary. Even though there is prior work [85,86,139] and proposed proof-of-concept solutions, there is still a long way to go. The following three directions may improve the hardware compiler, and thus increase FPGA programmability.

One important feature of the recent interfaces such as QPI, CXL, and OpenCAPI is that they support shared memory and data coherency. This development allows leveraging FPGAs more easily for a subset of a query or a portion of an operator in collaboration with CPUs. This enablement is significantly important for FPGA usability because not all the functionality may fit in the FPGA, as the FPGA size may depend on problem size, data types, or the function itself. Thus, a future hardware compiler might take the shared memory feature into account. A first step to start exploring this direction is to support OpenMP for hardware compilation [117,134]. OpenMP has been a popular parallel computing programming language for shared memory. It provides a few pragmas and library functions to enable multi-thread processing in CPUs, which eases multi-thread programming and is very similar to the HLS languages. Some recent enhancements, like task support, are an especially good fit for accelerators. Starting from OpenMP in the hardware compiler allows leveraging

the lessons learned from the study of OpenMP and might be the easier path to prove the benefits of supporting shared memory.

The second potential improvement comes from the support of standards for in-memory representation of data, such as Apache Arrow for columnar-oriented in-memory data. Taking Apache Arrow as an example, this standardized in-memory format allows data exchanges between different platform with zero-copy. Since it eliminates the serialization and deserialization overhead, it significantly enhances performance for data analytics and big data applications. Supporting such standards allows FPGA to be leveraged across languages and frameworks, which also reduces total amount of effort required.

Lastly, for FPGA designers that want to exercise more control over the design of the FPGA kernels for database queries or operators, the approaches that separate interface generation from computational kernel design can further increase productivity. Different FPGA kernels have different memory access patterns [37]. Some of them require gathering multiple streams (e.g., the merge tree), while some others have a random access pattern (e.g., the hash join). Thus simply using a DMA engine that is typically optimized for sequential data cannot meet all requirements, and customized data feeding logic is needed. While the ability to customized memory controllers is a strength of using FPGAs, optimizing this interface logic requires significant work. Thus, such interface generation frameworks can generate the interface automatically and designers can focus on the design and the implementation of kernels themselves. An example of such frameworks is shown in an initial work, the Fletcher framework [101].

8 Summary and conclusions

FPGAs have been recognized by the database community for their ability to accelerate CPU-intensive workloads. However, both the industry and academia have shown less interest in integrating FPGAs into database systems due to the following three reasons. First, while FPGAs can provide high data processing rates, the system performance is bounded by the limited bandwidth from conventional IO technologies. Second, FPGAs are competing with a strong alternative, GPUs, which can also provide high throughput and are much easier to program. Last, programming FPGAs typically requires a developer to have full-stack skills, from high-level algorithm design to low-level circuit implementation.

The good news is that these challenges are being addressed as can be seen through the technology trends and even the latest technologies. Interface technologies develop so fast that the interconnection between memory and accelerators can be expected to deliver main-memory scale bandwidth. In addition, FPGAs are incorporating new higher-bandwidth memory technologies such as the high-bandwidth memory, giving FPGAs a chance to combine a high degree parallel computation with high-bandwidth large-capacity local memory. Finally, emerging FPGA tool chains including HLS, new programming frameworks, and SQL-to-FPGA compilers, provide developers with better ease of use. Therefore FPGAs can become attractive again as a database accelerator.

In this paper, we explore the potential of using FPGAs to accelerate in-memory database systems. We reviewed the architecture of FPGA-accelerated database systems, discussed the challenges of integrating FPGAs into database systems, studied technology trends that address the challenges, and summarized the state-of-the-art research on database operator acceleration. We observe that FPGAs are capable of accelerating some of the database operators such as streaming operators, sort, and regular expression matching. In addition, emerging hardware compile tools further increase the usability of FPGAs in databases. We anticipate that the new technologies provide FPGAs with the opportunity to again deliver system-level speedup for database applications. However, there is still a long way to go, and future studies including new database architectures, new types of accelerators, deep performance analysis, and the development of the tool chains can push this progress a step forward.

References

1. Abdelfattah, M.S., Hagiescu, A., Singh, D.: Gzip on a chip: high performance lossless data compression on fpgas using opencl. In: Proceedings of the International Workshop on OpenCL 2013 & 2014, p. 4. ACM (2014)
2. Agarwal, K.B., Hofstee, H.P., Jamsek, D.A., Martin, A.K.: High bandwidth decompression of variable length encoded data streams. US Patent 8,824,569 (2014)
3. Albutiu, M.C., Kemper, A., Neumann, T.: Massively parallel sort-merge joins in main memory multi-core database systems. Proc. VLDB Endow. **5**(10), 1064–1075 (2012)
4. Apache: Apache Arrow. https://arrow.apache.org/. Accessed 01 Mar 2019
5. Apache: Apache Parquet. http://parquet.apache.org/. Accessed 01 Dec 2018
6. Arcas-Abella, O., Ndu, G., Sonmez, N., Ghasempour, M., Armejach, A., Navaridas, J., Song, W., Mawer, J., Cristal, A., Luján, M.: An empirical evaluation of high-level synthesis languages and tools for database acceleration. In: 2014 24th International Conference on Field Programmable Logic and Applications (FPL), pp. 1–8. IEEE (2014)

7. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., et al.: A view of the parallel computing landscape. Commun. ACM **52**(10), 56–67 (2009)
8. Balkesen, C., Alonso, G., Teubner, J., Özsu, M.T.: Multi-core, main-memory joins: sort vs. hash revisited. Proc. VLDB Endow. **7**(1), 85–96 (2013)
9. Balkesen, C., Teubner, J., Alonso, G., Özsu, M.T.: Main-memory hash joins on multi-core CPUs: tuning to the underlying hardware. In: IEEE 29th International Conference on Data Engineering (ICDE), 2013, pp. 362–373. IEEE (2013)
10. Balkesen, Ç., Teubner, J., Alonso, G., Özsu, M.T.: Main-memory hash joins on modern processor architectures. IEEE Trans. Knowl. Data Eng. **27**(7), 1754–1766 (2015)
11. Bartík, M., Ubik, S., Kubalik, P.: LZ4 compression algorithm on FPGA. In: IEEE International Conference on Electronics, Circuits, and Systems (ICECS), 2015, pp. 179–182. IEEE (2015)
12. Batcher, K.E.: Sorting networks and their applications. In: Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, pp. 307–314. ACM (1968)
13. Benton, B.: CCIX, Gen-Z, OpenCAPI: overview and comparison. https://www.openfabrics.org/images/eventpresos/2017presentations/213_CCIXGen-Z_BBenton.pdf (2017). Accessed 3 June 2018
14. Blanas, S., Li, Y., Patel, J.M.: Design and evaluation of main memory hash join algorithms for multi-core CPUs. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, pp. 37–48. ACM (2011)
15. Breß, S., Heimel, M., Siegmund, N., Bellatreche, L., Saake, G.: GPU-accelerated database systems: survey and open challenges. In: Transactions on Large-Scale Data-and Knowledge-Centered Systems XV, pp. 1–35. Springer (2014)
16. Cabrera, D., Martorell, X., Gaydadjiev, G., Ayguade, E., Jiménez-González, D.: OpenMP extensions for FPGA accelerators. In: 2009 International Symposium on Systems, Architectures, Modeling, and Simulation, pp. 17–24. IEEE (2009)
17. Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J.H., Brown, S., Czajkowski, T.: LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In: Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 33–36. ACM (2011)
18. Casper, J., Olukotun, K.: Hardware acceleration of database operations. In: Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 151–160. ACM (2014)
19. Caulfield, A.M., Chung, E.S., Putnam, A., Angepat, H., Fowers, J., Haselman, M., Heil, S., Humphrey, M., Kaur, P., Kim, J.Y., et al.: A cloud-scale acceleration architecture. In: The 49th Annual IEEE/ACM International Symposium on Microarchitecture, p. 7. IEEE Press (2016)
20. Chen, R., Prasanna, V.K.: Accelerating equi-join on a CPU-FPGA heterogeneous platform. In: 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE, pp. 212–219. IEEE (2016)
21. Chung, E.S., Davis, J.D., Lee, J.: Linqits: big data on little clients. In: ACM SIGARCH Computer Architecture News, vol. 41, pp. 261–272. ACM (2013)
22. Collet, Y., et al.: Lz4: extremely fast compression algorithm. https://code.google.com (2013). Accessed 3 June 2018
23. Cong, J., Fang, Z., Lo, M., Wang, H., Xu, J., Zhang, S.: Understanding performance differences of FPGAs and GPUs. In: 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 93–96. IEEE (2018)
24. Cong, J., Huang, M., Pan, P., Wu, D., Zhang, P.: Software infrastructure for enabling FPGA-based accelerations in data centers. In: Proceedings of the 2016 International Symposium on Low Power Electronics and Design, pp. 154–155. ACM (2016)
25. Cong, J., Huang, M., Wu, D., Yu, C.H.: heterogeneous datacenters: options and opportunities. In: Proceedings of the 53rd Annual Design Automation Conference, p. 16. ACM (2016)
26. Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., Zhang, Z.: High-level synthesis for FPGAs: from prototyping to deployment. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **30**(4), 473–491 (2011)
27. Crockett, L.H., Elliot, R.A., Enderwitz, M.A., Stewart, R.W.: The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc. Strathclyde Academic Media, Glasgow (2014)
28. Czajkowski, T.S., Aydonat, U., Denisenko, D., Freeman, J., Kinsner, M., Neto, D., Wong, J., Yiannacouras, P., Singh, D.P.: From OpenCL to high-performance hardware on FPGAs. In: 22nd International Conference on Field Programmable Logic and Applications (FPL), pp. 531–534. IEEE (2012)
29. Dan Bouvier Jim Gibney, A.B., Arora, S.: Delivering a New Level of Visual Performance in an SoC. https://www.slideshare.net/AMD/delivering-a-new-level-of-visual-performance-in-an-soc-amd-raven-rdige-apu (2018). Accessed 15 Oct 2018
30. David, H., Fallin, C., Gorbatov, E., Hanebutte, U.R., Mutlu, O.: Memory power management via dynamic voltage/frequency scaling. In: Proceedings of the 8th ACM international conference on Autonomic computing, pp. 31–40. ACM (2011)
31. Dennl, C., Ziener, D., Teich, J.: Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration. In: IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2013, pp. 25–28. IEEE (2013)
32. Deutsch, P.: GZIP file format specification version 4.3. Technical Report, RFC Editor (1996)
33. Duhem, F., Muller, F., Lorenzini, P.: Farm: fast reconfiguration manager for reducing reconfiguration time overhead on fpga. In: International Symposium on Applied Reconfigurable Computing, pp. 253–260. Springer (2011)
34. Fang, J., Chen, J., Al-Ars, Z., Hofstee, P., Hidders, J.: A high-bandwidth Snappy decompressor in reconfigurable logic: work-in-progress. In: Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis, pp. 16:1–16:2. IEEE Press (2018)
35. Fang, J., Chen, J., Lee, J., Al-Ars, Z., Hofstee, H.P.: A fine-grained parallel snappy decompressor for FPGAs using a relaxed execution model. In: 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 335–335. IEEE (2019)
36. Fang, J., Lee, J., Hofstee, H.P., Hidders, J.: Analyzing in-memory hash joins: granularity matters. In: Proceedings the 8th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, pp. 18–25 (2017)
37. Fang, J., et al.: Adopting OpenCAPI for high bandwidth database accelerators. In: 3rd International Workshop on Heterogeneous High-Performance Reconfigurable Computing (2017)
38. Feist, T.: Vivado design suite. White Paper, vol. 5 (2012)
39. Fossum, G.C., Wang, T., Hofstee, H.P.: A 64GB sort at 28GB/s on a 4-GPU POWER9 node for 16-byte records with uniformly distributed 8-byte keys. In: Proc. International Workshop on OpenPOWER for HPC. Frankfurt, Germany (2018)
40. Fowers, J., Kim, J.Y., Burger, D., Hauck, S.: A scalable high-bandwidth architecture for lossless compression on FPGAs. In: IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2015, pp. 52–59. IEEE (2015)

41. Francisco, P., et al.: The Netezza data appliance architecture: a platform for high performance data warehousing and analytics. http://www.ibmbigdatahub.com/sites/default/files/document/redguide_2011.pdf (2011). Accessed 3 June 2018
42. Franklin, M., Chamberlain, R., Henrichs, M., Shands, B., White, J.: An architecture for fast processing of large unstructured data sets. In: IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings, pp. 280–287. IEEE (2004)
43. Ghodsnia, P., et al.: An in-GPU-memory column-oriented database for processing analytical workloads. In: The VLDB Ph.D. Workshop. VLDB Endowment, vol. 1 (2012)
44. Google: Snappy. https://github.com/google/snappy/. Accessed 03 June 2018
45. Gopal, V., Guilford, J.D., Yap, K.S., Gulley, S.M., Wolrich, G.M.: Systems, methods, and apparatuses for decompression using hardware and software (2017). US Patent 9,614,544
46. Gopal, V., Gulley, S.M., Guilford, J.D.: Technologies for efficient lz77-based data decompression (2017). US Patent App. 15/374,462
47. Greenberg, M.: LPDDR3 and LPDDR4: How Low-Power DRAM Can Be Used in High-Bandwidth Applications. https://www.jedec.org/sites/default/files/M_Greenberg_Mobile%20Forum_May_%202013_Final.pdf (2013). Accessed 17 Oct 2017
48. Gupta, P.: Accelerating datacenter workloads. In: 26th International Conference on Field Programmable Logic and Applications (FPL) (2016)
49. Halstead, R.J., Absalyamov, I., Najjar, W.A., Tsotras, V.J.: FPGA-based Multithreading for In-Memory Hash Joins. In: CIDR (2015)
50. Halstead, R.J., Sukhwani, B., Min, H., Thoennes, M., Dube, P., Asaad, S., Iyer, B.: Accelerating join operation for relational databases with FPGAs. In: 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines, pp. 17–20. IEEE (2013)
51. He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational query coprocessing on graphics processors. ACM Trans. Database Syst. **34**(4), 21 (2009)
52. He, B., Yu, J.X.: High-throughput transaction executions on graphics processors. Proc. VLDB Endow. **4**(5), 314–325 (2011)
53. Heimel, M., Saecker, M., Pirk, H., Manegold, S., Markl, V.: Hardware-oblivious parallelism for in-memory column-stores. Proc. VLDB Endow. **6**(9), 709–720 (2013)
54. Huebner, M., Ullmann, M., Weissel, F., Becker, J.: Real-time configuration code decompression for dynamic FPGA self-reconfiguration. In: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, p. 138. IEEE (2004)
55. IBM: IBM power advanced compute (AC) AC922 server. https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=POD03143USEN&. Accessed 03 Sept 2018
56. Intel, F.: SDK for OpenCL. Programming guide. UG-OCL002 **31** (2016)
57. István, Z.: The glass half full: using programmable hardware accelerators in analytics. IEEE Data Eng. Bull. **42**(1), 49–60 (2019). http://sites.computer.org/debull/A19mar/p49.pdf
58. Jang, H., Kim, C., Lee, J.W.: Practical speculative parallelization of variable-length decompression algorithms. In: ACM SIGPLAN Notices, vol. 48, pp. 55–64. ACM (2013)
59. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the cell multiprocessor. IBM J. Res. Dev. **49**(4.5), 589–604 (2005)
60. Kara, K., Alonso, G.: Fast and robust hashing for database operators. In: 26th International Conference on Field Programmable Logic and Applications (FPL), 2016, pp. 1–4. IEEE (2016)
61. Kara, K., Giceva, J., Alonso, G.: FPGA-based data partitioning. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 433–445. ACM (2017)
62. Katz, P.W.: String searcher, and compressor using same. US Patent 5,051,745 (1991)
63. Kickfire: Kickfire. http://www.kickfire.com. Accessed 3 June 2018
64. Kim, C., Kaldewey, T., Lee, V.W., Sedlar, E., Nguyen, A.D., Satish, N., Chhugani, J., Di Blas, A., Dubey, P.: Sort vs. hash revisited: fast join implementation on modern multi-core CPUs. Proc. VLDB Endow. **2**(2), 1378–1389 (2009)
65. Kim, J., Kim, Y.: HBM: Memory Solution for Bandwidth-Hungry Processors. https://doc.xdevs.com/doc/Memory/HBM/Hynix/HC26.11.310-HBM-Bandwidth-Kim-Hynix-Hot%20Chips%20HBM%202014%20v7.pdf (2014). Accessed 29 Aug 2018
66. Kinetica: Kinetica. http://www.kinetica.com/. Accessed 3 June 2018
67. Kocberber, O., Grot, B., Picorel, J., Falsafi, B., Lim, K., Ranganathan, P.: Meet the Walkers. PROC of the 46th MICRO pp. 1–12 (2013)
68. Koch, D., Beckhoff, C., Teich, J.: Hardware decompression techniques for FPGA-based embedded systems. ACM Trans. Reconf. Technol. Syst. **2**(2), 9 (2009)
69. Koch, D., Torresen, J.: FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In: Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 45–54. ACM (2011)
70. Kruger, F.: CPU bandwidth: the worrisome 2020 trend. https://blog.westerndigital.com/cpu-bandwidth-the-worrisome-2020-trend/ (March 23, 2016). Accessed 03 May 2017
71. Lang, H., Leis, V., Albutiu, M.C., Neumann, T., Kemper, A.: Massively parallel NUMA-aware hash joins. In: In Memory Data Management and Analysis, pp. 3–14. Springer (2015)
72. Lee, J., Kim, H., Yoo, S., Choi, K., Hofstee, H.P., Nam, G.J., Nutter, M.R., Jamsek, D.: ExtraV: boosting graph processing near storage with a coherent accelerator. Proc. VLDB Endow. **10**(12), 1706–1717 (2017)
73. Lei, J., Chen, Y., Li, Y., Cong, J.: A high-throughput architecture for lossless decompression on FPGA designed using HLS. In: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 277–277. ACM (2016)
74. Lin, M.B., Chang, Y.Y.: A New Architecture of a Two-Stage Lossless Data Compression and Decompression Algorithm. IEEE Trans. VLSI Syst. **17**(9), 1297–1303 (2009)
75. Liu, H.Y., Carloni, L.P.: On learning-based methods for design-space exploration with high-level synthesis. In: Proceedings of the 50th Annual Design Automation Conference, p. 50. ACM (2013)
76. Mahajan, D., Kim, J.K., Sacks, J., Ardalan, A., Kumar, A., Esmaeilzadeh, H.: In-RDBMS hardware acceleration of advanced analytics. Proc. VLDB Endow. **11**(11), 1317–1331 (2018)
77. Mahony, A.O., Tringale, A., Duquette, J.J., O'carroll, P.: Reduction of execution stalls of LZ4 decompression via parallelization. US Patent 9,973,210 (2018)
78. Marcelino, R., Neto, H.C., Cardoso, J.M.: Unbalanced FIFO sorting for FPGA-based systems. In: 16th IEEE International Conference on Electronics, Circuits, and Systems, 2009. ICECS 2009, pp. 431–434. IEEE (2009)
79. Mashimo, S., Van Chu, T., Kise, K.: High-performance hardware merge sorter. In: IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017, pp. 1–8. IEEE (2017)
80. Mellanox Technologies: Mellanox Innova™-2 flex open programmable SmartNIC. http://www.mellanox.com/page/products_dyn?product_family=276&mtag=programmable_adapter_cards_innova2flex. Accessed 28 Apr 2019
81. Mostak, T.: An overview of MapD (massively parallel database). White Paper, Massachusetts Institute of Technology (2013)

82. Mueller, R., Teubner, J.: FPGA: what's in it for a database? In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, pp. 999–1004. ACM (2009)
83. Mueller, R., Teubner, J.: FPGAs: a new point in the database design space. In: Proceedings of the 13th International Conference on Extending Database Technology, pp. 721–723. ACM (2010)
84. Mueller, R., Teubner, J., Alonso, G.: Data processing on FPGAs. Proc. VLDB Endow. **2**(1), 910–921 (2009)
85. Mueller, R., Teubner, J., Alonso, G.: Streams on wires: a query compiler for FPGAs. Proc. VLDB Endow. **2**(1), 229–240 (2009)
86. Mueller, R., Teubner, J., Alonso, G.: Glacier: a query-to-hardware compiler. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, pp. 1159–1162. ACM (2010)
87. Mueller, R., Teubner, J., Alonso, G.: Sorting networks on FPGAs. VLDB J. **21**(1), 1–23 (2012)
88. Mulder, Y.: Feeding high-bandwidth streaming-based FPGA accelerators. Master's Thesis, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands (2018)
89. Nallatech: OpenCAPI enabled FPGAs—the perfect bridge to a data centric world. https://openpowerfoundation.org/wp-content/uploads/2018/10/Allan-Cantle.Nallatech-Presentation-2018-OPF-Summit_Amsterdam-presentation.pdf (2018). Accessed 25 Oct 2018
90. Nane, R., Sima, V.M., Olivier, B., Meeuws, R., Yankova, Y., Bertels, K.: DWARV 2.0: a CoSy-based C-to-VHDL hardware compiler. In: 22nd International Conference on Field Programmable Logic and Applications (FPL), pp. 619–622. IEEE (2012)
91. Nane, R., Sima, V.M., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y.T., Hsiao, H., Brown, S., Ferrandi, F., et al.: A survey and evaluation of FPGA high-level synthesis tools. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **35**(10), 1591–1604 (2016)
92. Napatech: Napatech SmartNIC solution for hardware offload. https://www.napatech.com/support/resources/solution-descriptions/napatech-smartnic-solution-for-hardware-offload/. Accessed 28 Apr 2019
93. Nikhil, R.: Bluespec System Verilog: efficient, correct RTL from high level specifications. In: Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-design, 2004. MEMOCODE'04., pp. 69–70. IEEE (2004)
94. Nyberg, C., Shah, M., Govindaraju, N.: Sort benchmark home page. http://sortbenchmark.org/. Accessed 03 Aug 2018
95. OpenPOWER: SNAP framework hardware and software. https://github.com/open-power/snap/. Accessed 03 June 2018
96. Ouyang, J., Qi, W., Yong, W., Tu, Y., Wang, J., Jia, B.: SDA: software-defined accelerator for general-purpose distributed big data analysis system. In: Hot Chips: A Symposium on High Performance Chips, Hotchips (2016)
97. Owaida, M., Sidler, D., Kara, K., Alonso, G.: Centaur: a framework for hybrid CPU-FPGA databases. In: IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017, pp. 211–218. IEEE (2017)
98. Papaphilippou, P., Luk, W.: Accelerating database systems using FPGAs: a survey. In: 2018 28th International Conference on Field Programmable Logic and Applications (FPL), pp. 125–1255. IEEE (2018)
99. PCI-SIG: PCI-SIG ® announces upcoming PCI express ® 6.0 specification to reach 64 GT/s. https://www.businesswire.com/news/home/20190618005945/en/PCI-SIG%C2%AE-Announces-Upcoming-PCI-Express%C2%AE-6.0-Specification. Accessed 01 July 2019
100. PCI-SIG: Specifications PCI-SIG. https://pcisig.com/specifications. Accessed 01 July 2019
101. Peltenburg, J., van Straten, J., Brobbel, M., Hofstee, H.P., Al-Ars, Z.: Supporting columnar in-memory formats on FPGA: the hardware design of fletcher for Apache Arrow. In: International Symposium on Applied Reconfigurable Computing, pp. 32–47. Springer (2019)
102. Pilato, C., Ferrandi, F.: Bambu: a modular framework for the high level synthesis of memory-intensive applications. In: 2013 23rd International Conference on Field programmable Logic and Applications, pp. 1–4. IEEE (2013)
103. Putnam, A., Caulfield, A.M., Chung, E.S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., et al.: A reconfigurable fabric for accelerating large-scale datacenter services. ACM SIGARCH Comput. Archit. News **42**(3), 13–24 (2014)
104. Qiao, W., Du, J., Fang, Z., Wang, L., Lo, M., Chang, M.C.F., Cong, J.: High-throughput lossless compression on tightly coupled CPU-FPGA platforms. In: FPGA, p. 291 (2018)
105. Qiao, Y.: An FPGA-based snappy decompressor-filter. Master's Thesis, Delft University of Technology (2018)
106. Scofield, T.C., Delmerico, J.A., Chaudhary, V., Valente, G.: XtremeData dbX: an FPGA-based data warehouse appliance. Comput. Sci. Eng. **12**(4), 66–73 (2010)
107. Sadoghi, M., Javed, R., Tarafdar, N., Singh, H., Palaniappan, R., Jacobsen, H.A.: Multi-query stream processing on FPGAs. In: 2012 IEEE 28th International Conference on Data Engineering, pp. 1229–1232. IEEE (2012)
108. Saitoh, M., Elsayed, E.A., Van Chu, T., Mashimo, S., Kise, K.: A high-performance and cost-effective hardware merge sorter without feedback datapath. In: 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 197–204. IEEE (2018)
109. Salami, B., Arcas-Abella, O., Sonmez, N.: HATCH: hash table caching in hardware for efficient relational join on FPGA. In: IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2015, pp. 163–163. IEEE (2015)
110. Salami, B., Arcas-Abella, O., Sonmez, N., Unsal, O., Kestelman, A.C.: Accelerating hash-based query processing operations on FPGAs by a hash table caching technique. In: Latin American High Performance Computing Conference, pp. 131–145. Springer (2016)
111. Salomon, D.: Data Compression: The Complete Reference. Springer, Berlin (2004)
112. Sharma, D.D.: Compute express link. https://docs.wixstatic.com/ugd/0c1418_d9878707bbb7427786b70c3c91d5fbd1.pdf (2019). Accessed 15 Apr 2019
113. Sidler, D., István, Z., Owaida, M., Alonso, G.: Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 403–415. ACM (2017)
114. Sidler, D., István, Z., Owaida, M., Kara, K., Alonso, G.: doppioDB: a hardware accelerated database. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 1659–1662. ACM (2017)
115. Singh, D.P., Czajkowski, T.S., Ling, A.: Harnessing the power of FPGAs using altera's OpenCL compiler. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 5–6. ACM (2013)
116. Sitaridi, E., Mueller, R., Kaldewey, T., Lohman, G., Ross, K.A.: Massively-parallel lossless data decompression. In: 2016 45th International Conference on Parallel Processing (ICPP), pp. 242–247. IEEE (2016)
117. Sommer, L., Korinth, J., Koch, A.: OpenMP device offloading to FPGA accelerators. In: 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 201–205. IEEE (2017)
118. Song, W., Koch, D., Luján, M., Garside, J.: Parallel hardware merge sorter. In: IEEE 24th Annual International Symposium

on Field-Programmable Custom Computing Machines (FCCM), 2016, pp. 95–102. IEEE (2016)
119. Srivastava, A., Chen, R., Prasanna, V.K., Chelmis, C.: A hybrid design for high performance large-scale sorting on FPGA. In: International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2015, pp. 1–6. IEEE (2015)
120. Stephenson, M., Amarasinghe, S.: Predicting unroll factors using supervised classification. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 123–134. IEEE Computer Society (2005)
121. Stone, J.E., Gohara, D., Shi, G.: OpenCL: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(1–3), 66–73 (2010)
122. Storer, J.A., Szymanski, T.G.: Data compression via textual substitution. J. ACM **29**(4), 928–951 (1982)
123. Stuecheli, J.: A new standard for high performance memory, acceleration and networks. http://opencapi.org/2017/04/opencapi-new-standard-high-performance-memory-acceleration-networks/. Accessed 3 June 2018
124. Sukhwani, B., Min, H., Thoennes, M., Dube, P., Brezzo, B., Asaad, S., Dillenberger, D.E.: Database analytics: a reconfigurable-computing approach. IEEE Micro **34**(1), 19–29 (2014)
125. Sukhwani, B., Min, H., Thoennes, M., Dube, P., Iyer, B., Brezzo, B., Dillenberger, D., Asaad, S.: Database analytics acceleration using FPGAs. In: Proceedings of the 21st international conference on Parallel architectures and compilation techniques, pp. 411–420. ACM (2012)
126. Sukhwani, B., Thoennes, M., Min, H., Dube, P., Brezzo, B., Asaad, S., Dillenberger, D.: Large payload streaming database sort and projection on FPGAs. In: 25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2013, pp. 25–32. IEEE (2013)
127. Teubner, J., Woods, L.: Data processing on FPGAs. Synth. Lect. Data Manag. **5**(2), 1–118 (2013)
128. Teubner, J., Woods, L., Nie, C.: XLynx-an FPGA-based XML filter for hybrid XQuery processing. ACM Trans. Database Syst. **38**(4), 23 (2013)
129. Thompto, B.: POWER9: processor for the cognitive era. In: Hot Chips 28 Symposium (HCS), 2016 IEEE, pp. 1–19. IEEE (2016)
130. Ueda, T., Ito, M., Ohara, M.: A dynamically reconfigurable equi-joiner on FPGA. IBM Technical Report RT0969 (2015)
131. Van Lunteren, J., Rohrer, J., Atasu, K., Hagleitner, C.: Regular expression acceleration at multiple tens of Gb/s. In: 1st Workshop on Accelerators for High-Performance Architectures in Conjunction with ICS (2009)
132. Wang, W., Zhang, M., Chen, G., Jagadish, H., Ooi, B.C., Tan, K.L.: Database meets deep learning: challenges and opportunities. ACM SIGMOD Rec. **45**(2), 17–22 (2016)
133. Wang, Z., He, B., Zhang, W.: A study of data partitioning on OpenCL-based FPGAs. In: 2015 25th International Conference on Field Programmable Logic and Applications (FPL), pp. 1–8. IEEE (2015)
134. Watanabe, Y., Lee, J., Boku, T., Sato, M.: Trade-off of offloading to FPGA in OpenMP task-based programming. In: International Workshop on OpenMP, pp. 96–110. Springer (2018)
135. Welch, T.A.: A technique for high-performance data compression. Computer **17**(6), 8–19 (1984). https://doi.org/10.1109/MC.1984.1659158
136. Wenzel, L., Schmid, R., Martin, B., Plauth, M., Eberhardt, F., Polze, A.: Getting started with CAPI SNAP: hardware development for software engineers. In: European Conference on Parallel Processing, pp. 187–198. Springer (2018)
137. Wirbel, L.: Xilinx SDAccel: a unified development environment for tomorrow's data center. Technical Report, The Linley Group Inc. (2014)
138. Wissolik, M., Zacher, D., Torza, A., Da, B.: Virtex UltraScale+ HBM FPGA: a revolutionary increase in memory performance. Xilinx Whitepaper (2017)
139. Woods, L., István, Z., Alonso, G.: Ibex: an intelligent storage engine with support for advanced SQL offloading. Proc. VLDB Endow. **7**(11), 963–974 (2014)
140. Wu, L., Barker, R.J., Kim, M.A., Ross, K.A.: Hardware-accelerated range partitioning. Columbia University Computer Science Technical Reports (2012)
141. Xilinx: GZIP/ZLIB/Deflate data compression core. http://www.cast-inc.com/ip-cores/data/zipaccel-d/cast-zipaccel-d-x.pdf (2016). Accessed 03 Aug 2018
142. Xilinx: UltraScale FPGA product tables and product selection guide. https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf (2018). Accessed 03 Sept 2018
143. Yuan, Y., Lee, R., Zhang, X.: The Yin and Yang of processing data warehousing queries on GPU devices. Proc. VLDB Endow. **6**(10), 817–828 (2013)
144. Zacharopoulos, G., Barbon, A., Ansaloni, G., Pozzi, L.: Machine learning approach for loop unrolling factor prediction in high level synthesis. In: 2018 International Conference on High Performance Computing & Simulation (HPCS), pp. 91–97. IEEE (2018)
145. Zeng, X.: FPGA-based high throughput merge sorter. Master's Thesis, Delft University of Technology (2018)
146. Zhang, C., Chen, R., Prasanna, V.: High throughput large scale sorting on a CPU-FPGA heterogeneous platform. In: Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International, pp. 148–155. IEEE (2016)
147. Zhang, S., He, J., He, B., Lu, M.: Omnidb: towards portable and efficient query processing on parallel cpu/gpu architectures. Proc. VLDB Endow. **6**(12), 1374–1377 (2013)
148. Zhou, X., Ito, Y., Nakano, K.: An efficient implementation of LZW decompression in the FPGA. In: IEEE International Parallel and Distributed Processing Symposium Workshops, 2016, pp. 599–607. IEEE (2016)
149. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977)
150. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Trans. Inf. Theory **24**(5), 530–536 (1978)

3

ACCELERATING SNAPPY DECOMPRESSION

SUMMARY

Recently, system interconnect has improved to such a level that the host-to-accelerator interface bandwidth might reach and even exceed the bandwidth of the host memory. It is unclear whether FPGAs can help computation-intensive database operations saturate such a large bandwidth. Thus, new accelerator architectures should be developed to improve the performance of the computation-intensive operations, e.g. (de)compression.
This chapter proposes an FPGA-based Snappy decompressor that can process multiple tokens in parallel and operates on each FPGA block RAM (BRAM) that holds a portion of the decompressed data, independently. A first stage efficiently refines the tokens into commands that operate on a single BRAM and steers the commands to the appropriate one. In a second stage, a relaxed execution model is used where each BRAM command executes immediately and the ones that return with invalid data are recycled to avoid stalls caused by a read-after-write dependency. The proposed method achieves up to 7.2 GB/s output throughput per engine on a CAPI2-attached Xilinx VU3P FPGA. A mid-sized device, integrated on a host bridge adapter and instantiating multiple engines, can keep pace with the full OpenCAPI 3.0 bandwidth of 25 GB/s.
We also propose a Parquet-to-Arrow converter implemented in an FPGA to allow a fast conversion between the Parquet format common for storage of columnar data and the column-oriented in-memory Apache Arrow standardized data format. The design is modular and extendable to support different Parquet formats. Experimental results show that the proposed architecture can achieve more than 7GB/s throughput per stream which is limited by the bandwidth of the connection to device memory.

The content of this chapter is based on the following papers:

J. Fang, J. Chen, J. Lee, Z. Al-Ars, H.P. Hofstee, *Refine and Recycle: A Method to Increase*

Decompression Parallelism, The 30th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), New York, NY, USA, July 15-17, 2019

L.T.J. van Leeuwen, J. Peltenburg, **J. Fang**, Z. Al-Ars, H.P. Hofstee, *High-throughput conversion of Apache Parquet files to Apache Arrow in-memory format using FPGAs*, Doorn, the Netherlands, 2019 International Conference on Computing Systems, June 3-5, 2019

Refine and Recycle: A Method to Increase Decompression Parallelism

Jian Fang*, Jianyu Chen*, Jinho Lee†, Zaid Al-Ars* and H. Peter Hofstee*†
{j.fang-1, z.al-ars, h.p.hofstee}@tudelft.nl, j.chen-13@student.tudelft.nl, leejinho@us.ibm.com
*Delft University of Technology, Delft, the Netherlands
†IBM Austin, Texas, USA

3

***Abstract*—Rapid increases in storage bandwidth, combined with a desire for operating on large datasets interactively, drives the need for improvements in high-bandwidth decompression. Existing designs either process only one token per cycle or process multiple tokens per cycle with low area efficiency and/or low clock frequency.**

We propose two techniques to achieve high single-decoder throughput at improved efficiency by keeping only a single copy of the history data across multiple BRAMs and operating on each BRAM independently. A first stage efficiently refines the tokens into commands that operate on a single BRAM and steers the commands to the appropriate one. In the second stage, a relaxed execution model is used where each BRAM command executes immediately and those with invalid data are recycled to avoid stalls caused by the read-after-write dependency.

We apply these techniques to Snappy decompression and implement a Snappy decompression accelerator on a CAPI2-attached FPGA platform equipped with a Xilinx VU3P FPGA. Experimental results show that our proposed method achieves up to 7.2 GB/s output throughput per decompressor, with each decompressor using 14.2% of the logic and 7% of the BRAM resources of the device. Therefore, a single decompressor can easily keep pace with an NVMe device (PCIe Gen3 x4) on a small FPGA, while a larger device, integrated on a host bridge adapter and instantiating multiple decompressors, can keep pace with the full OpenCAPI 3.0 bandwidth of 25 GB/s.

***Index Terms*—Snappy, decompression, FPGA, Acceleration**

I. INTRODUCTION

While much prior work has studied how to improve the compression speed of lossless data compression [1]–[3], the common case is to compress the data once for storage and decompress it multiple times whenever it is processed.

Recent studies [4]–[8] illustrate that FPGAs are a promising platform for lossless data decompression. The customizable capability, the feasibility of bit-level control, and high degrees of parallelism of the FPGA allow designs to have many light-weight customized cores, enhancing performance. Leveraging these advantages, the pipelined FPGA designs of LZSS [4], [5], LZW [6] and Zlib [7], [9] all achieve good decompression throughput. However, these prior designs only process one token per FPGA cycle, resulting in limited speedup compared to software implementations. The studies [8] and [10] propose solutions to handle multiple tokens per cycle. However, both solutions require multiple copies of the history buffer and require extra control logic to handle BRAM bank conflicts caused by parallel reads/writes from different tokens, leading to low area efficiency and/or a low clock frequency.

A compressed Snappy file consists of tokens, where a token contains the original data itself (literal token) or a back reference to previously written data (copy token). Even with a large and fast FPGA fabric, decompression throughput is degraded by stalls introduced by *read-after-write* (RAW) data dependencies. When processing tokens in a pipeline, copy tokens may need to stall and wait until the prior data is valid. In this paper, we propose two techniques to achieve efficient high single-decompressor throughput by keeping only a single BRAM-banked copy of the history data and operating on each BRAM independently. A first stage efficiently refines the tokens into commands that operate on a single BRAM and steers the commands to the appropriate one. In the second stage, rather than spending a lot of logic on calculating the dependencies and scheduling operations, a recycle method is used where each BRAM command executes immediately and those that return with invalid data are recycled to avoid stalls caused by the RAW dependency. We apply these techniques to Snappy [11] decompression and implement a Snappy decompression accelerator on a CAPI2-attached FPGA platform equipped with a Xilinx VU3P FPGA. Experimental results show that our proposed method achieves up to 7.2 GB/s throughput per decompressor, with each decompressor using 14.2% of the logic and 7% of the BRAM resources of the device. One decompressor keeps pace with an NVMe device (PCIe Gen3 x4) on a small FPGA. Compared to the software implementation, significant performance improvement is achieved.

Specifically, this paper makes the following contributions.

- We increase decompression parallelism by breaking tokens into BRAM commands that operate independently.
- We propose a recycle method to reduce the stalls caused by the intrinsic data dependencies in the compressed file.
- We apply these techniques to develop a Snappy decompressor that can process multiple tokens per cycle.
- We evaluate end-to-end performance. Our decompressor achieves up to 7.2 GB/s throughput.

In the remainder of this paper, Section II introduces Snappy and summarizes related work. Section III discusses solutions to address BRAM bank conflicts and RAW dependencies. Section IV details the Snappy decompressor architecture. Section V presents experimental results and Section VI contains a summary and conclusions.

II. BACKGROUND

A. Snappy (De)compression

Snappy is an LZ77-based [12] byte-level (de)compression algorithm widely used in big data systems, especially in the Hadoop ecosystem, and is supported by big data formats such as Parquet [13] and ORC [14]. Snappy works with a fixed uncompressed block size (64KB) without any delimiters to imply the block boundary. Thus, a compressor can easily partition the data into blocks and compress them in parallel, but achieving concurrency in the decompressor is difficult because block boundaries are not known due to the variable compressed block size. Because the 64kB blocks are individually compressed, there is a fixed (64kB) history buffer during decompression, unlike the sliding history buffers used in LZ77, for example. Similar to the LZ77 compression algorithm, the Snappy compression algorithm reads the incoming data and compares it with the previous input. If a sequence of repeated bytes is found, Snappy uses a (length, offset) tuple, *copy* token, to replace this repeated sequence. The length indicates the length of the repeated sequence, while the offset is the distance from the current position back to the start of the repeated sequence, limited to the 64kB block size. For those sequences not found in the history, Snappy records the original data in another type of token, the $literal$ tokens.

TABLE I: Procedure of Snappy decompression

```
1  while(!eof) {
2    reset(&history);
3    while(!end_of_block()){
4      read_tag_byte(&ptr, &type, &extra_len, &lit_len, &copy_len);
5      read_extra_bytes(&ptr, extra_len, &lit_len, &copy_offset);
6      if(type==copy){
7        read_history(history, copy_offset, copy_len, &buffer);
8        update_history_c(&history, buffer, copy_len);
9      }
10     else // type==lit
11       update_history_l(&history, &ptr, lit_len);
12   }
13   output(history);
14 }
```

Snappy decompression is the reverse process of the compression. It translates a stream with literal tokens and copy tokens into uncompressed data. Even though Snappy decompression is less computationally intensive than Snappy compression, the internal dependency limits the decompression parallelism. To the best of our knowledge, the highest Snappy decompression throughput is reported in [15] using the "lzbench" [16] benchmark, where the throughput reaches 1.8GB/s on a Core i7-6700K CPU running at 4.0GHz. Table I shows the pseudo code of the Snappy decompression, which can also be applied to other LZ-based decompression algorithms. The first step is to parse the input stream (variable ptr) into tokens (Line 4 & 5). During this step, as shown in Line 4 of Table I, the tag byte (the first byte of a token) is read and parsed to obtain the information of the token, e.g. the token type ($type$), the length of the literal string (lit_len), the length of copy string ($copy\_len$), and the length of extra bytes of this token ($extra\_len$). Since the token length varies and might be larger than one byte, if the token requires extra bytes (length indicated by $extra\_len$ in Line 5) to store the information, it needs to read and parse these bytes to extract and update the token information. For a literal token, as it contains the uncompressed data that can be read directly from the token, the uncompressed data is extracted and added to the history buffer (Line 11). For a copy token, the repeated sequence can be read according to the offset (variable $copy\_offset$) and the length (variable $copy\_len$), after which the data will be updated to the tail of the history (Line 7 & 8). When a block is decompressed (Line 3), the decompressor outputs the history buffer and resets it (Line 13 & 2) for the decompression of the next block.

There are three data dependencies during decompression. The first dependency occurs when locating the block boundary (Line 3). As the size of a compressed block is a variable, a block boundary cannot be located until the previous block is decompressed, which brings challenges to leverage the block-level parallelism. The second dependency occurs during the generation of the token (Line 4 & 5). A Snappy compressed file typically contains different sizes of tokens, where the size of a token can be decoded from the first byte of this token (known as the tag byte), exclusive the literal content. Consequently, a token boundary cannot be recognized until the previous token is decoded, which prevents the parallel execution of multiple tokens. The third dependency is the RAW data dependency between the reads from the copy token and the writes from all tokens (between Line 7 and Line 8 & 11). During the execution of a copy token, it first reads the repeated sequence from the history buffer that might be not valid yet if multiple tokens are processed in parallel. In this case, the execution of this copy token need to be stalled and wait until the request data is valid. In this paper, we focus on the latter two dependencies, and the solutions to reduce the impact of these dependencies are explained in section IV-C (second dependency) and section III-B (third dependency).

B. Related Work

Many recent studies consider improving the speed of lossless decompression. To address the block boundary problems, [17] explores the block-level parallelism by performing pattern matching on the delimiters to predict the the block boundaries. Unfortunately, this technique cannot be applied to Snappy because Snappy uses a fixed uncompressed block size (64KB) without any delimiters. Another way to utilize the block-level parallelism is to add some constraints during the compression, e.g adding padding to make fixed size compressed blocks [18] or add some meta data to indicate the boundary of the blocks [19]. A drawback of these methods is that it is only applicable to the modified compression algorithms (add padding) or even not compatible to the original (de)compression algorithms (add meta data).

The idea of using FPGAs to accelerate decompression has been studied for years. On the one hand, FPGAs provide a high-degree of parallelism by adopting techniques such

as task-level parallelization, data-level parallelization, and pipelining. On the other hand, the parallel array structure in an FPGA offers tremendous internal memory bandwidth. One approach is to pipeline the design and separate the token parsing and token execution stages [4]–[7]. However, these methods only process one token each FPGA cycle, limiting throughput.

Other works study the possibility of processing multiple tokens in parallel. [20] proposes a parallel LZ4 decompression engine that has separate hardware paths for literal tokens and copy tokens. The idea builds on the observation that the literal token is independent since it contains the original data, while the copy token relies on the previous history. A similar two-path method for LZ77-based decompression is shown in [21], where a slow-path routine is proposed to handle large literal tokens and long offset copy tokens, while a fast-path routine is adopted for the remaining cases. [10] introduces a method to decode variable length encoded data streams that allows a decoder to decode a portion of the input streams by exploring all possibilities of bit spill. The correct decoded streams among all the possibilities are selected as long as the bit spill is calculated and the previous portion is correctly decoded. [8] proposes a token-level parallel Snappy decompressor that can process two tokens every cycle. It uses a similar method as [10] to parse an eight-byte input into tokens in an earlier stage, while in the later stages, a conflict detector is adopted to detect the type of conflict between two adjacent tokens and only allow those two tokens without conflict to be processed in parallel. However, these works cannot easily scale up to process more tokens in parallel because it requires very complex control logic and duplication of BRAM resources to handle the BRAM bank conflicts and data dependencies.

The GPU solution proposed in [19] provides a multi-round resolution method to handle the data dependency. In each round, all the tokens with read data valid are executed, while those with data invalid will be pending and wait for the next round execution. This method allows out-of-order execution and does not stall when a request needs to read the invalid data. However, this method requires specific arrangement of the tokens, and thus requires modification of the compression algorithm.

This paper presents a new FPGA decompressor architecture that can process multiple tokens in parallel and operate at a high clock frequency without duplicating the history buffers. It adopts a refine and recycle method to reduce the impact of the BRAM conflicts and data dependencies, and increases the decompression parallelism, while conforming to the Snappy standard. This paper improves on our previous proof-of-concept work [22] and integrates it in a CAPI 2.0-enabled POWER 9 system.

III. The Refine and Recycle Method

A. The Refine Method for BRAM Bank Conflict

Typically, in FPGAs, the large history buffers (e.g. 32KB in GZIP and 64KB in Snappy) can be implemented using BRAMs. Taking Snappy as an example, as shown in Fig. 1, to construct a 64KB history buffer, a minimum number of BRAMs are required: 16 4KB blocks for the Xilinx Ultrascale Architecture [23]. These 16 BRAMs can be configured to read/write independently, so that more parallelism can be achieved. However, due to the structure of BRAMs, a BRAM block supports limited parallel reads or writes, e.g. one read and one write in the simple dual port configuration. Thus, if more than one read or more than one write need to access different lines in the same BRAM, a conflict occurs (e.g. conflict on bank 2 between read $R1$ and read $R2$ in Fig. 1). We call this conflict a *BRAM bank conflict* (BBC).

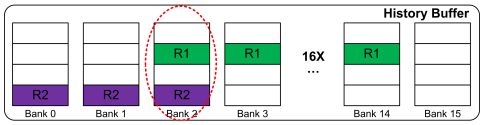


Fig. 1: An example of BRAM bank conflicts in Snappy

For Snappy specifically, the maximum literal length for a literal token and the maximum copy length for copy tokens in the current Snappy version is 64B. As the BRAM can only be configured to a maximum 8B width, there is a significant possibility that a BBC occurs when processing two tokens in the same cycle, and processing more tokens in parallel further increases the probability of a BBC. A naive way to deal with the BBCs is to only process one of the conflicting tokens and stall the others until the this token completes. For example, in Fig. 1, when a read request from a copy token ($R1$) has a BBC with another read request from another copy token ($R2$), the execution of $R2$ stalls and waits until $R1$ is finished. Obviously, this method sacrifices some parallelism and even leads to a degradation from parallel processing to sequential processing. Duplicating the history buffers can also relieve the impact of BBCs. The previous work [8] uses a double set of history buffers, where two parallel reads are assigned to different set of history. So, the two reads from the two tokens never have BBCs. However, this method only solves the read BBCs but not the write BBCs, since the writes need to update both sets of history to maintain the data consistency. Moreover, to scale this method to process more tokens in parallel, additional sets (linearly proportional to the number of tokens being processed in parallel) of BRAMs are required.

To reduce the impact of BBCs, we present a refine method to increase token execution parallelism without duplicating the history buffers. The idea is to break the execution of tokens into finer-grain operations, the BRAM copy/write commands, and for each BRAM to execute its own reads and writes independently. As illustrated in Fig. 1, $R1$ and $R2$ only have a BBC in bank 2, while the other parts of these two reads do not conflict. We refine the token into BRAM commands operating on each bank independently. As a result, for the reads in the non-conflicting banks of $R2$ (bank 0 & 1), we allow the execution of the reads on these banks from $R2$. For the conflicting bank 2, $R1$ and $R2$ cannot be processed concurrently.

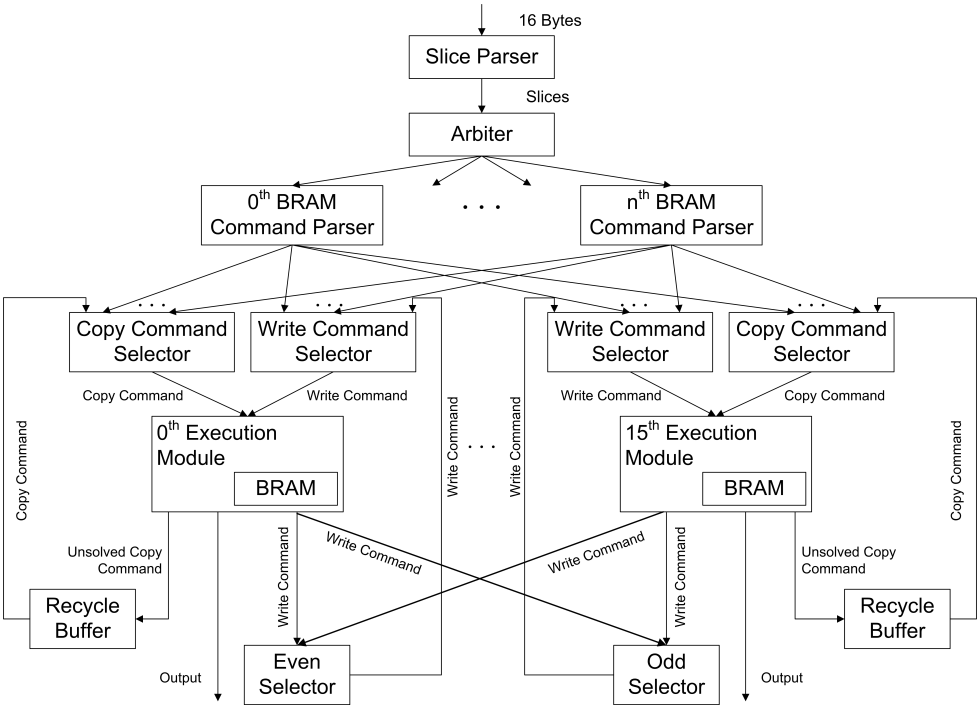


Fig. 2: Architecture overview

The proposed method takes advantage of the parallelism of the array structure in FPGAs by operating at a finer-grained level, the single BRAM read/write level, compared with the token-level. It supports partially executing multiple tokens in parallel even when these tokens have BBCs. In the extreme case, the proposed method can achieve up to 16 BRAM operations in parallel, meaning generating the decompressed blocks at a speed of 128B per cycle. This refine method can also reduce the read-after-write dependency impact mentioned in section III-B. If the read data of a read request from a copy token is partially valid, this method allows this copy token to only read the valid data and update the corresponding part of the history, instead of waiting until all the bytes are valid.

B. The Recycle Method for RAW Dependency

The *Read-After-Write* (RAW) dependency between data reads and writes on the history buffer is another challenge for parallelization. If a read needs to fetch data from some memory address that the data has not yet been written to, a hazard occurs, and thus this read needs to wait until the data is written. A straightforward solution [8] is to execute the tokens sequentially and perform detection to decide whether the tokens can be processed in parallel. If a RAW hazard is detected between two tokens that are being processed in the same cycle, it forces the latter token to stall until the previous token is processed. Even though we can apply the forwarding technique to reduce the stall penalty, detecting multiple tokens and forwarding the data to the correct position requires complicated control logic and significant hardware resource.

Another solution is to allow out-of-order execution. That is when a RAW hazard occurs between two tokens, the follow-up tokens are allowed to be executed without waiting these two tokens are finished, which is very similar to out-of-order execution in the CPU architecture. Fortunately, in the decompression case, this does not require a complex textbook solution such “Tomasulo” or “Scoreboarding” to store the state of the pending tokens. Instead, rerunning pending tokens after the execution of all or some of the follow-up tokens guarantees the correction of this out-of-order execution. This is because there is no write-after-write or write-after-read dependency during the decompression, or two different writes never write the same place and the write data never changes after the data is read. So, there is no need to record the write data states, and thus a simpler out-of-order execution model can satisfy the requirement, which saves logic resources.

In this paper, we present the recycle method to reduce the impact of RAW dependency in a BRAM command granularity. Specifically, when a command needs to read the history data

that is not valid yet, the decompressor executes this command immediately without checking if all the data is valid. If the data that has been read is detected to be not entirely valid, this command (invalid data part) should be recycled and stored in a recycle buffer, where it will be executed again (likely after a few other commands are executed). If the data is still invalid in the next execution, this decompressor performs this recycle-and-execute precedure repeatedly until the read data is valid.

This method executes the commands in a relaxed model and allows continuous execution on the commands without stalling the pipeline. The method provides more parallelism since it does not need to be restricted to the degree of parallelism calculated by dependency detection.

IV. Snappy Decompressor Architecture

A. Architecture Overview

Fig. 2 presents an overview of the proposed architecture. It can be divided into two stages. The first stage parses the input stream lines into tokens and refines these tokens into BRAM commands that will be executed in the second stage. It contains a slice parser to locate the boundary of the tokens, multiple BRAM command parsers(BCPs) to refine the tokens into BRAM commands, and an arbiter to drive the output of the slice parser to one of the BCPs. In the second stage, the BRAM commands are executed to generate the decompressed data under the recycle method. The execution modules, in total 16 of them, are the main components in this stage, in which recycle buffers are utilized to perform the recycle mechanism.

The procedure starts with receiving a 16B input line in the slice parser. Together with the first 2B of the next input line, this 18B is parsed into a “slice” that contains token boundary information including which byte is a starting byte of a token, whether any of the first 2B have been parsed in the previous slice, and whether this slice starts with literal content, etc. After that, an arbiter is used to distribute each slice to one of the BCPs that work independently, and there the slice is split into one or multiple BRAM commands. There are two types of BRAM commands, write commands and copy commands. The write command is generated from the literal token, indicating a data write operation on the BRAM, while the copy command is produced from the copy token which leads to a read operation and a follow-up step to generate one or two write commands to write the data in the appropriate BRAM blocks.

In the next stage, write selectors and copy selectors are used to steer the BRAM commands to the appropriate execution module. Once the execution module receives a write command and/or a copy command, it executes the command and performs BRAM read/write operations. As the BRAM can perform both a read and a write in the same cycle, each execution module can simultaneously process a write command and one copy command (only the read operation) at the same time. The write command will always be completed successfully once the execution module receives it, which is not the case for the copy command. After performing the read operation of the copy command, the execution module runs two optional extra tasks according to the read data, including generating new write/copy commands and recycling the copy command. If the read data contains some valid bytes, new write commands are generated to write this data to its destination. If some bytes are still invalid, the copy command will be renewed (removing the completed portion from the command) and collected by a recycle unit, and sent back for the next round of execution. Once a 64KB history is built, this 64KB data is output as the decompressed data block. After that, a new data block is read, and this procedure will be repeated until all the data blocks are decompressed.

B. History Buffer Organization

The 64KB history buffer consists of 16 4KB BRAM blocks, using the FPGA 36Kb BRAM primitives in the Xilinx Ultra-Scale fabric. Each BRAM block is configured to have one read port and one write port, with a line width of 72bits (8B data and 8bits flags). Each bit from the 8bits flags indicates whether the corresponding byte is valid. To access a BRAM line, 4 bits of BRAM bank address, and 9 bits of BRAM line address is required. The history data is stored in these BRAMs in a striped manner to balance the BRAM read/write command workload and to enhance parallelism.

C. Slice Parser

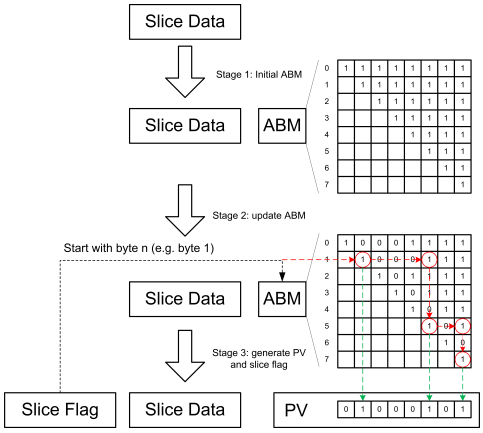


Fig. 3: Procedure of the slice parser and structure of the Assumption Bit Map

The slice parser aims to decode the input data lines into tokens in parallel. Due to the variety of token sizes, the starting byte of a token needs to be calculated from the previous token. This data dependency presents an obstacle for the parallelization of the parsing process. To solve this problem, we assume all 16 input bytes are starting bytes, and to parse this input data line based on this assumption. The correct branch will be chosen once the first token is recognized. To achieve a high frequency for the implementation, we propose

a bit map based byte-split detection algorithm by taking advantage of bit-level control in FPGA designs.

A bit map is utilized to represent the assumption of starting bytes, which is called the Assumption Bit Map (ABM) in the remainder of this paper. For a N bytes input data line, we need a $N * N$ ABM. As shown in Fig 3, taking an 8B input data line as an example, cell(i, j) being equal to '1' in the ABM means that if corresponding byte i is a starting byte of one token, byte j is also a possible starting byte. If a cell has a value '0', it means if byte i is a starting byte, byte j cannot be a starting byte.

This algorithm has three stages. In the first stage, an ABM is initialized with all cells set to '1'. In the second stage, based on the assumption, each row in the ABM is updated in parallel. For row i, if the size of the token starts with the assumption byte is L, the following $L-1$ bits are set to be 0. The final stage merges the whole ABM along with the slice flag from the previous slice, and calculate a Position Vector (PV). The PV is generated by following a cascading chain. First of all, the slice flag from the previous slice points out which is the starting byte of the first token in the current slice (e.g. byte 1 in Fig. 3). Then the corresponding row in the ABM is used to find the first byte of the next token (byte 5 in Fig. 3), and its row in the ABM is used for finding the next token. This procedure is repeated (all within a single FPGA cycle) until all the tokens in this slice are found. The PV is an N-bit vector that its ith bit equal to '1' means the ith byte in the current slice is a starting byte of a token. Meanwhile, the slice flag will be updated. In addition to the starting byte position of the first token in the next slice, the slice flag contains other informations such as whether the next slice starts with literal content, the unprocessed length of the literal content, etc.

D. BRAM Command Parser

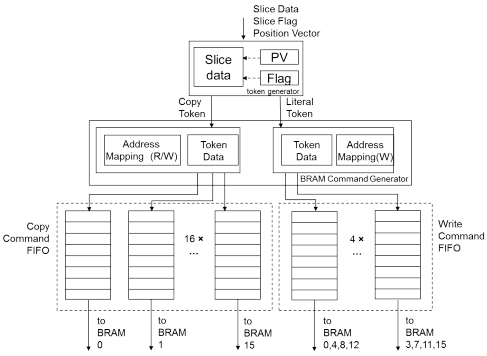


Fig. 4: Structure of BRAM command parser

The BRAM command parser refines the tokens and generates BRAM commands based on the parsed slice. The structure of the BCP is demonstrated in Fig. 4. The first step is to generate tokens based on the token boundary information that is stored in the PV. Literal tokens and copy tokens output from the token generator are assigned to different paths for further refining in the BRAM command generator. In the literal token path, the BRAM command generator calculates the token write address and length, and splits this write operation into multiple ones to map the write address to the BRAM address. Within a slice, the maximum length of the literal token is 16B, i.e. the largest write is 16B, which can generate up to 3 BRAM write commands. In the copy token path, the BRAM command generator performs a similar split operation but maps both the read address and the write address to the BRAM address. A copy token can copy up to 64B data. Hence, it generates up to 9 BRAM copy commands.

Since multiple commands are generated each cycle, to prevent stalling the pipeline, we use multiple sets of FIFOs to store them before sending them to the corresponding execution module. Specifically, 4 FIFOs are used to store the literal commands which is enough to store all 3 BRAM write commands generated in one cycle. Similarly, 16 copy command FIFOs are used to handle the maximum 9 BRAM copy commands. To keep up with the input stream rate (16B per cycle), multiple BCPs can work in parallel to enhance the parsing throughput.

E. Execution Module

The execution module performs BRAM command execution and the recycle mechanism. Its structure is illustrated in Fig. 5. It receives up to 1 write command from the write command selector and 1 copy command from the copy command selector. Since each BRAM has one independent read port and one independent write port, each BRAM can process one read command and one copy command each clock cycle. For the write command, the write control logic extracts the write address from the write command and performs a BRAM write operation. Similarly, the read control logic extracts the read address from the read command and performs a BRAM read operation.

While the write command can always be processed successfully, the copy command can fail when the target data is not ready in the BRAM. So there should be a recycle mechanism for failed copy commands. After reading the data, the unsolved control logic checks whether the read data is valid. There are three different kinds of results: 1) all the target data is ready (hit); 2) only part of the target data are ready (partial hit); 3) none of the target data is ready (miss). In the hit case and the partial hit case, the new command generator produces one or two write commands to write the copy results to one or two BRAMs, depending on the alignment of the write data. In the partial hit case and the miss case, a new copy command is generated and recycled, waiting for the next round of execution.

F. Selector Selection Strategy

The BRAM write commands and copy commands are placed in separate paths, and can work in parallel. The Write Command Selector gives priority to recycled write commands.

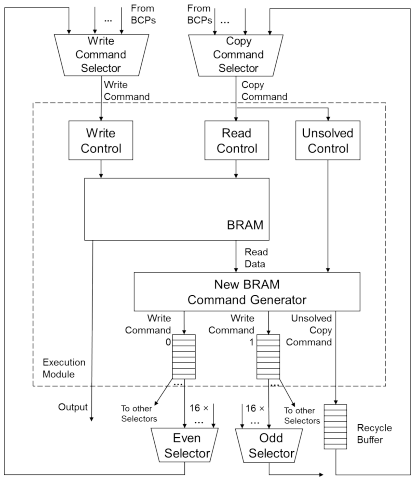


Fig. 5: Structure of execution module

Priority is next given to write commands from one of the BCPs using a round robin method. The Copy Command Selector gives priority to the copy commands from one of the BCPs when there is a small number of copy commands residing in the recycle FIFO. However, when this number reaches a threshold, priority will be given back to the recycle commands. This way, it not only provides enough commands to be issued and executed, but also guarantees the recycle FIFO does not overflow, and no deadlock occurs.

V. EVALUATION

A. Experimental Setup

To evaluate the proposed design, an implementation is created targeting the Xilinx Virtex Ultrascale VU3P-2 device on an AlphaData ADM-PCIE-9V3 board and integrated with the POWER9 CAPI 2.0 [24] interface. The CAPI 2.0 interface on this card supports the CAPI protocol at an effective data rate of approximately 11 GB/s. The FPGA design is compared with an optimized software Snappy decompression implementation [16] compiled by gcc 7.3.0 with "O3" option and running on a POWER9 CPU in little endian mode with Ubuntu 18.04.1 LTS.

We test our Snappy decompressor for functionality and performance on 6 different data sets. The features of the data sets are listed in Table II. The first three data sets are from the "lineitem" table of the TPC-H benchmarks in the database domain. We use the whole table (Table) and two different columns including a long integer column (Integer) and a string column (String). The data set Wiki [25] is an XML file dump from Wikipedia, while the Matrix is a sparse matrix from the Matrix Market [26]. We also use a very high compression ratio file (Geo) which stores geographic information.

TABLE II: Benchmarks used and throughput results

Files	Original size (MB)	Compression ratio	Throughput (GB/s) CPU	Throughput (GB/s) FPGA	Speedup
Integer	45.8	1.70	0.59	4.40	7.46
String	157.4	2.45	0.69	6.02	8.70
Table	724.7	2.07	0.59	6.11	10.35
Matrix	771.3	2.75	0.80	4.80	6.00
Wiki	953.7	1.97	0.56	5.72	10.21
Geo	128.0	5.50	1.41	7.21	5.11

B. Resource Utilization

Table III lists the resource utilization of our design timing at 250MHz. The decompressor configured with 6 BCPs and 16 execution module takes around 14.2% of the LUTs, 7% of the BRAMs, 4.7% of the Flip-Flops in the VU3P FPGA. The recycle buffers, the components that are used to support out-of-order execution, only take 0.3% of the LUTs and 1.2% of the BRAMs. The CAPI 2.0 interface logic implementation takes up around 20.8% of the LUTs and 33% of the BRAMs. Multi-unit designs can share the CAPI 2.0 interface logic between all the decompressors, and thus the (VU3P) device can support up to 5 engines.

TABLE III: Resource utilization of design components

Resource	LUTs	BRAMs[1]	Flip-Flops
Recycle buffer	1.1K(0.3%)	8(1.2%)	1K(0.1%)
Decompressor	56K(14.2%)	50(7.0%)	37K(4.7%)
CAPI2 interface	82K(20.8%)	238(33.0%)	79K(10.0%)
Total	138K(35.0%)	288(40.0%)	116K(14.7%)

[1] One 18kb BRAM is counted as a half of one 36kb BRAM.

C. End-to-end Throughput Performance

We measure the end-to-end decompression throughput reading and writing from host memory. We compare our design with the software implementation running on one POWER9 CPU core (remember that parallelizing Snappy decompression is difficult due to unknown block boundaries).

Fig. 6 shows the end-to-end throughput performance of the proposed architecture configured with 6 BCPs. The proposed Snappy decompressor reaches up to 7.2 GB/s output throughput or 31 bytes per cycle for the file (Geo) with high compression ratio, while for the database application (Table) and web application (Wiki) it achieves 6.1 GB/s and 5.7 GB/s, which is 10 times faster than the software implementation. One decompressor can easily keep pace with a (Gen3 PCIe x4) NVMe device, and the throughput of an implementation containing two of such engines can reach the CAPI 2.0 bandwidth upper bound.

Regarding the power efficiency, the 22-core POWER9 CPU is running under 190 watts, and thus it can provide up to 0.16GB/s per watt. However, the whole ADM 9V3 card can support 5 engines under 25 watts [27], which corresponds to up to 1.44GB/s per watt. Consequently, our Snappy decompressor is almost an order of magnitude more power efficient than the software implementation.

TABLE IV: FPGA decompression accelerator comparison

Design	Frequency	Throughput		History	Area		Efficiency	
	(MHz)	GB/s	bytes/cycle	Size(KB)	LUTs	BRAMs	MB/s per 1K LUT	MB/s per BRAM
ZLIB (CAST) [9]	165	0.495	3.2	32	5.4K	10.5	93.9[1]	48.3
Snappy [8]	140	1.96	15	64	91K	32	22	62.7
This Work	250	7.20	30.9	64	56K	50	131.6	147.5

[1] Please note that ZLIB is more complex than Snappy and takes more LUTs to obtain the same throughput performance in principle.

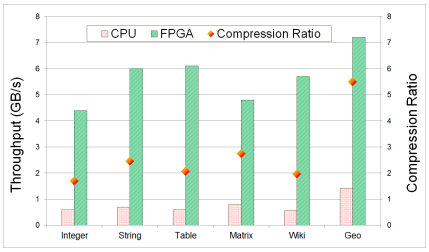


Fig. 6: Throughput of Snappy decompression

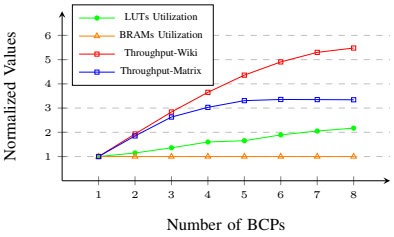


Fig. 7: Impact of number of BCPs

D. Design Trade-off with # of BCPs

As explained in section IV, the number of BCPs corresponds to the number of tokens that can be refined into BRAM commands per cycle. We compare the resource utilization and throughput of different numbers of BCPs, and present the results that are normalized by setting the resource usage and throughput of one BCP as 1 in Fig 7. Increasing from one BCP to two leads to 10% more LUT usage, but results in around 90% more throughput and no changes in BRAM usage. However, the increase of the throughput on Matrix slows down after 3 BCPs and the throughput remains stable after 5 BCPs. A similar trend can be seen in Wiki where the throughput improvement drops after 7 BCPs. This is because after increasing the number of BCPs, the bottleneck moves to the stage of parsing the input line into tokens. Generally, a 16B-input line contains 3-7 tokens depending on the compressed file, while the maximum number of tokens is 8, thus explaining the limited benefits of adding more BCPs. One way to achieve higher performance is to increase both the input-line size and the number of BCPs. However, this might bring new challenges to the resource utilization and clock frequency, and even reach the upper bound of the independent BRAM operations parallelism.

E. Comparison of Decompression Accelerators

We compare our design with state-of-the-art decompression accelerators in Table IV. By using 6 BCPs, a single decompressor of our design can output up to 31B per cycle at a clock frequency of 250MHz. It is around 14.5x and 3.7x faster then the prior work on ZLIB [9] and Snappy [8]. Even scaling up the other designs to the same frequency, our design is still around 10x and 2x faster, respectively. In addition, our design is much more area-efficient, measured in MB/s per 1K LUTs and MB/s per BRAM (36kb), which is 1.4x more LUT efficient than the ZLIB implementation in [9] and 2.4x more BRAM efficient than the Snappy implementation in [8].

VI. Summary and Conclusion

The control and data dependencies intrinsic in the design of a decompressor present an architectural challenge. Even in situations where it is acceptable to achieve high throughput performance by processing multiple streams, a design that processes a single token or a single input byte each cycle becomes severely BRAM limited for (de)compression protocols that assume a sizable history buffer. Designs that decode multiple tokens per cycle could use the BRAMs efficiently in principle, but resolving the data dependencies leads to either very complex control logic, or to duplication of BRAM resources. Prior designs have therefore exhibited only limited concurrency or required duplication of the history buffers.

This paper presented a refine and recycle method to address this challenge and applies it to Snappy decompression to make an FPGA-based Snappy decompressor. In an earlier stage, the proposed design refines the tokens into commands that operate on a single BRAM independently to reduce the impact of the BRAM bank conflicts. In the second stage, a recycle method is used where each BRAM command executes immediately without dependency checking and those that return with invalid data are recycled to avoid stalls caused by the RAW dependency. For a single Snappy input stream our design processes up to 16 input bytes per cycle. The end-to-end evaluation shows that the design achieves up to 7.2 GB/s output throughput or about an order of magnitude faster than the software implementation in the POWER9 CPU. This bandwidth for a single-stream decompressor is sufficient for an NVMe (PCIe x4) device. Two of these decompressor engines, operating on independent streams, can saturate a PCIe Gen4

or CAPI 2.0 x8 interface, and the design is efficient enough to easily support data rates for an OpenCAPI 3.0 x8 interface.

REFERENCES

[1] M. Bartík, S. Ubik, and P. Kubalik, "LZ4 compression algorithm on FPGA," in *Electronics, Circuits, and Systems (ICECS), 2015 IEEE International Conference on*. IEEE, 2015, pp. 179–182.

[2] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, "A scalable high-bandwidth architecture for lossless compression on fpgas," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE, 2015, pp. 52–59.

[3] W. Qiao, J. Du, Z. Fang, M. Lo, M.-C. F. Chang, and J. Cong, "High-Throughput Lossless Compression on Tightly Coupled CPU-FPGA Platforms," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2018, pp. 291–291.

[4] M. Huebner, M. Ullmann, F. Weissel, and J. Becker, "Real-time configuration code decompression for dynamic fpga self-reconfiguration," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 138.

[5] D. Koch, C. Beckhoff, and J. Teich, "Hardware decompression techniques for FPGA-based embedded systems," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 2, no. 2, p. 9, 2009.

[6] X. Zhou, Y. Ito, and K. Nakano, "An efficient implementation of LZW decompression in the FPGA," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2016, pp. 599–607.

[7] J. Yan, J. Yuan, P. H. Leong, W. Luk, and L. Wang, "Lossless compression decoders for bitstreams and software binaries based on high-level synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2842–2855, 2017.

[8] Y. Qiao, "An FPGA-based Snappy Decompressor-Filter," Master's thesis, Delft University of Technology, 2018.

[9] C. Inc., "ZipAccel-D GUNZIP/ZLIB/Inflate Data Decompression Core," http://www.cast-inc.com/ip-cores/data/zipaccel-d/cast-zipaccel-d-x.pdf, 2016, accessed: 2019-03-01.

[10] K. B. Agarwal, H. P. Hofstee, D. A. Jamsek, and A. K. Martin, "High bandwidth decompression of variable length encoded data streams," Sep. 2 2014, uS Patent 8,824,569.

[11] Google, "Snappy," https://github.com/google/snappy/, accessed: 2018-12-01.

[12] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[13] Apache, "Apache Parquet," http://parquet.apache.org/, accessed: 2018-12-01.

[14] Apache, "Apache ORC," https://orc.apache.org/, accessed: 2018-12-01.

[15] "Zstandard," available: http://facebook.github.io/zstd/, accessed: 2019-05-15.

[16] "lzbench," available: https://github.com/inikep/lzbench, accessed: 2019-05-15.

[17] H. Jang, C. Kim, and J. W. Lee, "Practical speculative parallelization of variable-length decompression algorithms," in *ACM SIGPLAN Notices*, vol. 48, no. 5. ACM, 2013, pp. 55–64.

[18] M. Adler, "pigz: A parallel implementation of gzip for modern multiprocessor, multi-core machines," *Jet Propulsion Laboratory*, 2015.

[19] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross, "Massively-parallel lossless data decompression," in *Proc. of the International Conference on Parallel Processing*. IEEE, 2016, pp. 242–247.

[20] A. O. Mahony, A. Tringale, J. J. Duquette, and P. O'carroll, "Reduction of execution stalls of LZ4 decompression via parallelization," May 15 2018, uS Patent 9,973,210.

[21] V. Gopal, S. M. Gulley, and J. D. Guilford, "Technologies for efficient LZ77-based data decompression," Aug. 31 2017, uS Patent App. 15/374,462.

[22] J. Fang, J. Chen, Z. Al-Ars, P. Hofstee, and J. Hidders, "A high-bandwidth Snappy decompressor in reconfigurable logic: work-in-progress," in *Proceedings of the International Conference on Hardware/-Software Codesign and System Synthesis*. IEEE Press, 2018, pp. 16:1–16:2.

[23] S. Leibson and N. Mehta, "Xilinx ultrascale: The next-generation architecture for your next-generation architecture," *Xilinx White Paper WP435*, 2013.

[24] J. Stuecheli, "A new standard for high performance memory, acceleration and networks," available: http://opencapi.org/2017/04/opencapi-new-standard-high-performance-memory-acceleration-networks/, accessed: 2018-06-03.

[25] M. Mahoney, "Large text compression benchmark," available: http://www.mattmahoney.net/text/text.html, 2011.

[26] "Uf sparse matrix collection," available: https://www.cise.ufl.edu/research/sparse/MM/LAW/hollywood-2009.tar.gz.

[27] Alpha Data, "ADM-PCIE-9V3 User Manual," available: https://www.alpha-data.com/pdfs/adm-pcie-9v3usermanual_v2_7.pdf, 2018, accessed: 2019-05-15.

3

High-throughput conversion of Apache Parquet files to Apache Arrow in-memory format using FPGA's

Lars T.J. van Leeuwen, Johan Peltenburg, Jian Fang, Zaid Al-Ars
Delft University of Technology
Delft, The Netherlands
{l.t.j.vanleeuwen@student., j.w.peltenburg@, j.fang-1@,
z.al-ars@}tudelft.nl

Peter Hofstee
IBM Austin & TU Delft
Austin, USA
h.p.hofstee@tudelft.nl

***Abstract*— Recently, persistent storage bandwidth has increased tremendously due to the use of flash technology. In the domain of big data analytics, the bottleneck of converting storage focused file formats to in-memory data structures has shifted from the storage technology to the software components that are tasked with decompression and organization of the data in memory. One commonly used file format is Apache Parquet, and a recently developed in-memory format is Apache Arrow. In order to improve the bandwidth at which such conversions take place, we propose a Parquet-to-Arrow converter implemented in an FPGA. The design is modular and extendable to support different Parquet formats. The resource utilization of the Xilinx XCVU9P device used for the prototype is 4.16% of CLBs and 1.78% BRAMs, leaving ample room to implement analytical kernels that operate in tandem with the file conversion. The prototype shows promising throughput for converting the basic structure of Parquet files with large page sizes, with the throughput being limited by the bandwidth of the connection to device memory.**

I. Introduction

With the arrival of NVMe SSD's, the bandwidth associated with reading data from persistent storage is increasing rapidly. If the bandwidth of persistent storage is no longer a bottleneck, conversion of storage focused formats to data structures usable in memory risks becoming a limiting factor in database systems. In order to improve the performance of database systems we propose performing this conversion on an FPGA. We present a framework that takes Apache Parquet [1] pages as input and creates Apache Arrow [2] format data structures in memory using the Fletcher [3] framework.

II. Background

Parquet is a columnar storage format that supports multiple compression and encoding schemes for stored data [1]. With Parquet's columns divided into individually compressed and encoded pages, analytics applications can benefit from columnar data while still allowing for smaller scale accesses without having to decompress and decode the whole file.

Arrow is a columnar format that is focused on efficient in-memory representation of data [2]. Like Parquet, its columnar format allows fast vectorized operations with the aim of preventing data copies or serialization between different language run-times through shared memory pools.

Fletcher is the framework that allows FPGA's access to Arrow data with a fast and easy to use interface [3]. Instead of byte addresses only column indices are required to read and write data in Arrow format. Fletcher hardware is generated based on a schema describing the data set stored in Arrow format.

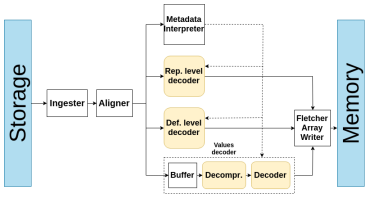


Fig. 1. High-level architecture of the hardware

III. Design

A. High-level architecture

With Parquet and Arrow both being columnar formats the hardware can read consecutive Parquet pages from memory, interpret the page headers, and perform decompression and decoding steps on the page data without having to do significant data reordering.

A Parquet page consists of four distinct, variable-length blocks of data. The header, the repetition levels, the definition levels, and the actual values. The header contains information on (among other things) the size of the following three blocks in the page and the number of entries in the page. The repetition levels and definition levels encode the structure (in case of nested data structures) and nulls in the page respectively according to the algorithm described in Google's Dremel paper [4]. These levels are not encoded in the case of non-nested or non-nullable data. Finally the compressed and encoded values complete the page.

There are three requirements for the design. First, the design should be modular and expandable to enable acceleration of the many different schemes Parquet supports. Second, the design should be area-efficient in order to fit many instances on an FPGA for parallel workloads, or leave room for analytical kernels. Third, the high-level architecture should maximize throughput so any decompressors and decoders can be fed data at a rate close to system bandwidth (e.g. PCIe bandwidth).

The proposed high-level architecture is seen in Figure 1. The aligner ensures that each of the modules responsible for reading one of the four main blocks of data in a Parquet page in turn receives their data correctly aligned. The

aligner requires these modules to report back the amount of bytes they used after they are finished. The rounded blocks are replaceable or omissible modules to enable support for different compression and encoding schemes.

Although Figure 1 shows the decoder directly streaming the decoded values into Fletcher for immediate writing to memory this does not have to be the case. Any hardware supporting Fletcher style streams can be inserted in between to allow for operations on the data (e.g. maps, filters, etc.) before it is written to memory, allowing for optimal use of the FPGA's resources in accelerating any data analytics application starting from a Parquet file.

B. Decompression

The first decompressor being integrated with the Parquet to Arrow converter is a Snappy decompressor implemented in previous work [5]. Preliminary benchmarks show a throughput of 3GB/s input and 5GB/s output for Snappy compressed files for a single instance of the decompressor. This is an order of magnitude higher than a single thread on a Core i7 processor.

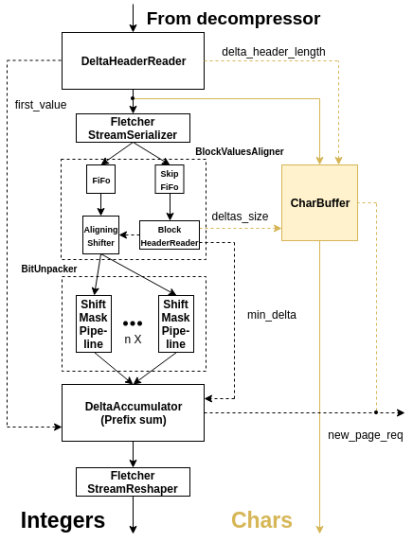


Fig. 2. Delta decoder

C. Decoding

In order to make the Parquet to Arrow converter usable out of the box, decoders will be included that can decode at least one encoding for floats, doubles, 32 and 64 bit integers, and strings. Because of the prevalence of delta encoding in Parquet (for integers and string lengths) a delta decoder as seen in fig. 2 is proposed. In delta decoding the values are encoded as variable-width bit-packed deltas with respect to the previous value. After narrowing the bit-width of the stream from the decompressor for easier manipulation using a serializer, the bit-packed values will be correctly aligned based on the bit-packing width and block header length data. Hereafter, the values can be unpacked in a separate shift and mask pipeline for each value in the aligned data. These values will be added to the minimum delta received from the BlockHeaderReader to create the final deltas that can be added to the previously decoded integers in the DeltaAccumulator.

A buffer for the character data can be added to make this decoder work for strings as the raw character data directly follows the encoded string lengths.

IV. Preliminary Results

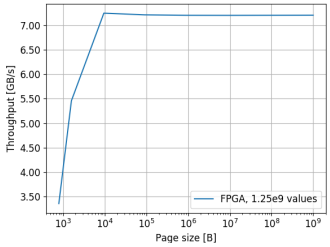


Fig. 3. Throughput for different page sizes

To establish the overhead of processing the Parquet page headers, an implementation was made to convert simple (uncompressed, non-nullable, plainly encoded) Parquet files containing only a column of 64 bit integers on an AWS f1.2xlarge instance with a Xilinx Ultrascale+ FPGA. The resulting throughput is seen in fig. 3. This implementation required only 4.16% of CLB's and 1.78% of BRAM while timing at 250MHz. For files containing small Parquet pages the conversion rate is limited by the latency of reading the page headers. This effect stops being significant at 100kB pages, after which it starts being limited by the bandwidth of the connection to device memory. This suggests decompression and decoding modules can make full use of the read/write bandwidth of the FPGA if large enough pages are used in the Parquet file.

V. Conclusion

Preliminary results show that the converter can process the basic structure of a Parquet file at a high throughput. Continuing work is focused on implementing decompression and decoding modules that make full use of the strengths of an FPGA to create Arrow format data from a Parquet file significantly faster than a general processor. Through the use of the proposed converter, big data analytics applications may alleviate the software bottlenecks resulting from decompression and conversion overhead of the Parquet storage format.

References

[1] Apache. Apache Parquet. [Online]. Available: https://parquet.apache.org/ (Accessed 2019-04-29).

[2] Apache. Apache Arrow. [Online]. Available: https://arrow.apache.org/ (Accessed 2019-04-29).

[3] J. Peltenburg, J. van Straten, M. Brobbel, H. P. Hofstee, and Z. Al-Ars, "Supporting columnar in-memory formats on fpga: The hardware design of fletcher for apache arrow," in *Applied Reconfigurable Computing*, C. Hochberger, B. Nelson, A. Koch, R. Woods, and P. Diniz, Eds. Cham: Springer International Publishing, 2019, pp. 32–47.

[4] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive analysis of web-scale datasets," in *Proc. of the 36th Int'l Conf on Very Large Data Bases*, 2010, pp. 330–339. [Online]. Available: http://www.vldb2010.org/accept.htm

[5] J. Fang, J. Chen, Z. Al-Ars, P. Hofstee, and J. Hidders, "Work-in-progress: A high-bandwidth snappy decompressor in reconfigurable logic," in *2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Sep. 2018, pp. 1–2.

4

HASH JOIN ANALYSIS

SUMMARY

FPGAs are known to accelerate computation-intensive operations. However, for memory-intensive operations, it is not clear to what degree an FPGA can help to improve their performance. To arrive at an answer, performance analysis on memory-intensive operations running on the CPU architecture is required. We investigate the main impact factors and how they influence the performance of database operations. This chapter takes the hash join as an example and analyzes its performance, which can then guide designing FPGA-based hash join accelerators.

The hash join is a commonly used database operator and significantly impacts database performance. Predicting the performance of join algorithms on modern hardware is challenging. In this chapter, we focus on main-memory no-partitioning and partitioning hash join algorithms executing on multi-core platforms. We discuss the main parameters impacting performance, and present an effective performance model. This model can be used to select the most appropriate algorithm for different input data sets for current and future hardware configurations. We find that for modern systems an optimized no-partition hash join often outperforms an optimized radix partitioning hash join. The presented model can serve as a guideline for predicting how future development in hardware platforms, such as increasing bandwidth, will impact the performance of these algorithms and how they could be adapted.

The content of this chapter is based on the following paper:

J. Fang, J. Lee, H.P. Hofstee, J. Hidders, *Analyzing In-Memory Hash Joins: Granularity Matters*, 2017 International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS), Munich, Germany, Sep 1, 2017

Analyzing In-Memory Hash Joins: Granularity Matters

Jian Fang
Delft University of Technology
Delft, the Netherlands
j.fang-1@tudelft.nl

Jinho Lee, Peter Hofstee*
IBM Research
*and Delft Univ. of Technology
Austin, TX 78758, USA
leejinho,hofstee@us.ibm.com

Jan Hidders
Vrije Universiteit Brussel
1050 Brussels, Belgium
jan.hidders@vub.be

ABSTRACT

Predicting the performance of join algorithms on modern hardware is challenging. In this work, we focus on main-memory no-partitioning and partitioning hash join algorithms executing on multi-core platforms. We discuss the main parameters impacting performance, and present an effective performance model. This model can be used to select the most appropriate algorithm for different input data-sets for current and future hardware configurations. We find that for modern systems an optimized no-partition hash join often outperforms an optimized radix partitioning hash join.

1. INTRODUCTION

In the past decades, as new computer architectures emerged, the competition between different main-memory hash join algorithms has become more interesting. Significant research [1, 16, 12, 2, 3, 10, 9] has gone into developing algorithms that efficiently utilize the underlying hardware and this has provided a guidance to choosing better performing algorithms on modern hardware. However, the hardware develops so fast that the best algorithm today is probably not the best in the future. One obvious change that may lead to this is increased memory bandwidth.

In the past few years, development of hash join algorithms has been focused on partitioning the data to the size that fits within the last-level cache. However in modern systems, contrary to prior findings, we find that partitioning hash join is not always better than no-partitioning hash join. In this paper, we analyze the performance of hash join algorithms by a bandwidth-driven model and provide a guideline for choosing the right algorithm according to the dataset and architecture characteristics.

The contributions of the paper are as follows:

- We analyze the impact factors for the hash join algorithms. We discuss the importance of granularity.
- We build a performance model that considers both computation and memory accesses.
- Based on the proposed model, we study a no-partitioning hash join and a radix partitioning hash join.
- We predict the performance changes along with the changes of the data set and the hardware.
- We validate the model with hardware-based experiments on two processor architectures.

2. BACKGROUND

See [6] for early work on in-memory databases and hash-join.

2.1 Hash Join Algorithms

Classical hash join The classical hash join contains a build phase and a probe phase. During the build phase, tuples in the smaller relation R are scanned to build a hash table, and assigned to the corresponding hash table bucket. After the hash table is built, the probe phase scans the other relation S. For each tuple in S, it probes the hash table and validates the matches.

No-partitioning hash join No-partitioning hash join [4] is a parallel version of classical hash join. Each relation is equally divided into shares. In the build phase, each share in the smaller relation R is scanned by a thread. Together the threads build a shared hash table. Similarly, in the probe phase, each thread gets a piece of data from relation S and does the probe in parallel.

Partitioning hash join To limit cache misses, [16] introduces the partitioning hash join. Before building the hash table, both relations are divided into small partitions. Hash tables are built separately for each partition of relation R. After the hash table is built for one partition, the corresponding partition in relation S is scanned to do the probe. If the hash table for each partition is small enough to fit in the cache, this reduces cache misses, at the cost of partitioning overhead. The partitioning hash join can be further optimized as the radix partitioning hash join [12] which uses multiple partitioning passes to reduce the translation lookaside buffer (TLB) misses.

2.2 Related Work

The discussion between no-partitioning hash join and partitioning hash join never stops since the underlying hardware develops at a fast pace. Much research effort has been made on tuning the algorithms to leverage the underlying hardware platforms. Prior work [16, 9, 1, 5] shows how to use cache in a more efficient way for partitioning hash

join. Shatdal et al. [16] point out that if the hash table for each partition is small enough to store in the last level cache (LLC), the number of cache misses is reduced. Chen et al. [5] use prefetch to reduce the cache misses. Based on the cache optimization, Kim et al. [9] and Manegold et al. [12] further focus on the TLB problem. They show that using multiple passes partitioning can reduce the TLB misses caused by accessing too many pages in a one pass. Lang et al. [10] propose a NUMA-aware hash join algorithm based on their finding on NUMA effect that unreasonably distributed data in multiple NUMA nodes decreases performance.

Earlier research [2, 3, 4, 15] also makes comparisons between no-partition hash join and partitioning hash join. Blanas et al. [4] evaluate both classes of algorithms on multi-core platforms. They claim that no-partitioning hash join is competitive because partitioning hash join introduces extra cost such as computation for partitioning and synchronization which is higher than the penalty of cache misses in no-partitioning hash join. In [2] and [3], partitioning hash join, categorized as hardware-conscious algorithm, is well configured to compare with no-partitioning hash join which is hardware-oblivious. The authors argue that the partition hash join runs fast in most of the cases, and the hardware-oblivious hash join only performs well when the ratio between the size of two relations is significantly different. However, it is interesting to compare a well-configured partitioning hash join with a optimized no-partitioning hash join such as using prefetch. [15] compares 13 different join algorithms including hash join and sort merge join. Even though partitioning hash join performs better than no partitioning hash join, they point out that there is not a 100% best algorithm.

Another way to compare the algorithms is to build a performance model and use it for prediction. In [14, 13, 8, 7], performance models are set up describing the main cost of different hash join algorithms. These models consider both processing and disk I/O cost. It is increasingly feasible to store the entire database [6] in memory. For this organization, I/O cost is not the dominating part, but memory accesses are. Prior work[11] presents a cost model based on the cache lines transferred, but this work only covers the no-partitioning hash join algorithms. In this paper, we focus on in-memory processing and build a model to estimate the performance of in-memory hash join algorithms with special attention to granularity effects.

3. PERFORMANCE MODEL

In this section, we present a performance model to describe the cost of hash join algorithms. We first analyze the factors that should be considered to estimate the performance and how they influence the algorithms. Based on these impact factors, we propose a general performance model, followed by the study of two specific hash join algorithms: the parallel no-partitioning hash join and radix partitioning hash join.

3.1 Impact factors

Previous work has shown that there are many factors affecting the performance of hash join algorithms [3, 10, 15]. We divide them into two categories, application characteristics and hardware features. Some algorithms' parameters can be tuned according to these factors to gain better performance.

3.1.1 Application Characteristics

Relation size Relation size depends on the tuple size and the number of tuples. However, we can minimize the size with some pre-processing operations such as filtering and abstracting the payload of each tuple to a fixed size pointer that points to the original tuple. In our performance model, we assume that the relations have been pre-processed by filtering and each tuple contains a key and a pointer-style payload. When we compare hash-join algorithms we do so for same-size inputs, but we need to be aware that algorithms may scale differently with respect of each of the input relations.

Distribution of dataset. Within a dataset the distribution of values within a certain column can be even or skewed. Building a hash table for skewed data causes more hash collisions, while probing a hash table for skewed data increases cache hits. In this paper, although we know the data skew is important and interesting, we assume that the dataset is evenly distributed, since it is easier to explain the principle of our performance model. In the future, we plan to work out a refinement of the model to describe the cost of hash joins for skewed data.

3.1.2 Hardware Features

Granularity. Typical modern server processors have cache line sizes of 64 or 128 bytes, and (cached) accesses to main memory occur at this level of granularity. Randomly accessing elements smaller than a cache line, or quantities that cross a cache line, introduces granularity effect, incurring a granularity overhead, also referred to as read- or write-amplification.

Cache size. Probing hash tables requires randomly accessing the hash table. If the cache is not large enough to contain the whole hash table, cache misses occur, decreasing the performance of the hash join. To reduce the number of cache misses, we can partition the data before building the hash table, in order to fit the hash table of each partition in the cache.

TLB entries and virtual page size. The TLB caches virtual memory page address to physical page address translations. If the translation table does not cache the requested translation entry, a TLB miss occurs. The requested data cannot be loaded until the translation entry is updated. More misses are likely if more pages are accessed. Main memory hash join algorithms, especially the partitioning hash join, are sensitive to the number of TLB entries [12]. Using a large page size can reduce the TLB entries required [2, 3], increasing the TLB hit rate.

Memory bandwidth. Memory bandwidth determines the rate at which data can be transferred between main memory and the CPU. For memory-bound algorithms, the overhead of transferring data dominates.

CPU processing rate. CPU processing rate indicates how fast the processors process data. For hash join on modern hardware we believe optimized algorithms are usually memory bandwidth bound. However, memory bandwidth may increase in the near future, which means algorithms may become compute bound, and we need to explore trade-offs between computation and memory accesses.

Other features. There are many other potential impact factors such as NUMA topologies, simultaneous multithreading (SMT), synchronization, and load balancing. These factors have been proved to be influential in [3, 10].

In this paper, to simplify the model, we limit the algorithms to running on a single-node machine, and assume that the SMT, synchronization and load balancing factors contribute only a few percent of the overall cost. While we do not include these factors in the proposed performance model, we show that despite this the model can provide a reliable prediction of the performance.

3.2 Model

3.2.1 Platform Assumptions

Our model assumes a basic multi-core machine with a memory size that is large enough to store all the data, and we assume no intermediate results need to be written back to the disk. As we want to study the comparison between no-partitioning hash join and partitioning hash join, to make them competitive, we assume that neither of the two relations can fit in the last level cache (LLC). Each data transfer between the LLC and main memory should be an integer multiple of smallest main memory transfer size, namely granularity, normally the same as the cache line size. Although NUMA architectures play an important role in the hash join performance [10], we focus on a single NUMA node system for now, and defer the analysis of NUMA effects and multi-node systems to future work.

3.2.2 The formulas and their explanation

Table 1 shows the notation of the parameters used in our performance model. Please note that to predict relative performance of algorithms, our model does not use any empirically fitted parameters. To describe different hash join algorithms and their variants, we adopt a uniform description. The total running time T of a hash join algorithm consists of the running time of each phase:

$$T = \sum_{i=1}^{n} T_i \tag{1}$$

For the running time of each phase T_i, we consider both the cost of memory accesses and that of computation. The cost of each phase should be the sum of the time the system was computing and the time it was accessing memory, minus the time it was doing both.

$$T_i = T_{com} + T_{mem} - T_{overlap} \tag{2}$$

Suppose $\beta = \frac{min\{T_{com}, T_{mem}\} - T_{overlap}}{max\{T_{com}, T_{mem}\}}$, then

$$T_i = (1+\beta) max\{T_{com}, T_{mem}\} \tag{3}$$

For a well-optimized algorithm we assume maximum overlap, i.e., $\beta = 0$. So, formula 2 reduces to:

$$T_i = max\{T_{com}, T_{mem}\} \tag{4}$$

The computation cost can be represented as:

$$T_{com} = \frac{D}{P} + C \tag{5}$$

where D denotes the processing data amount, and P indicates the base processing rate. We use C to represent the other costs such as cache miss penalty, TLB miss penalty, synchronization cost, etc. The memory accesses in each phase consist of different passes of read and write. For a multi-pass memory access we have:

Table 1: Model Parameters

Parameters	Description
T	total running time
T_i	running time of each phase
n	total number of phases
T_{com}	computation time
T_{mem}	memory accesses time
$T_{overlap}$	the overlap between T_{com} and T_{mem}
C	penalty cost during computing
m	total passes of memory access in each phase
D	required data amount, equal to relation size
D_r	data amount for read
D_w	data amount for write
P	processing rate
B	memory bandwidth
B_r	read bandwidth
B_w	write bandwidth
W	tuple size
G	granularity(size of cache line)
α	the number of cache lines the data span across
R, S	relation R, S
$\|S\|$, $\|R\|$	tuple number of relation R, S

$$T_{mem} = \sum f(D_r, B_r, D_w, B_w) \tag{6}$$

The function $f(D_r, B_r, D_w, B_w)$ indicates the data transfer time of each data pass. (D_r/D_w) is the read/write data amount and (B_r/B_w) the read/write bandwidth. We consider both a memory channel shared between read and write and a (buffered) architecture with separate read and write channels. Thus,

$$f(D_r, B_r, D_w, B_w) = \begin{cases} \frac{D_r + D_w}{B} & \text{Shared channel} \\ max\left\{\frac{D_r}{B_r}, \frac{D_w}{B_w}\right\} & \text{Non-shared channel} \end{cases}$$

When calculating D_r and D_w access granularity must be taken into account. This means that instead of calculating these as the size of the elements transferred times the number of elements transferred, D_r or D_w become αG times the number of elements where α accounts for the number of granules transferred per element because of size and alignment, and G is the granule size.

3.3 Study of Hash Join Algorithms

We analyze two prevalent parallel hash join algorithms mentioned in [2, 4], a parallel no-partitioning hash join and a histogram-based 2-pass partitioning hash join. For this case study, we assume, that relation R is not larger than relation S, or $|R| \leq |S|$. The hash table is built based on relation R. In section 3.3.1 and 3.3.2, we analyze both algorithms with a shared memory channel machine, and extend this to the non-shared channel machine in section 3.3.3.

3.3.1 No Partitioning Hash Join

The parallel no partitioning hash consists of two phases: the build phase and the probe phase.

$$T_{NP} = T_{build} + T_{probe}$$

During the build phase, the CPUs scan all the tuples in relation R, and a hash function is applied to each tuple's key to build the hash table. After that, the hash table is written back to main memory. Thus, the memory accesses in the build phase contain a sequential read of relation R, and for each tuple in R, it reads and writes the corresponding hash table buckets with granularity effect. As we can reduce

hash collision by techniques such as using larger hash tables or a better hash function, we assume that there are no or few hash collisions. So, for the build phase or the probe phase, only one hash table bucket is accessed for each tuple. Hence the actual data amount for read is:

$$W|R| + \alpha G|R|$$

The actual data amount for write is:

$$\alpha G|R|$$

The total data transfer time for the build phase is:

$$T_{mem1} = \frac{W|R| + \alpha G|R| + \alpha G|R|}{B} = \frac{(W + 2\alpha G)|R|}{B} \quad (7)$$

The cost of computation contains the pure processing part and the other cost C_1. We use P_i to indicate the processing rate of each phase. According to previous research based on the modern multi-cores architecture [2], the cache miss penalty caused by randomly accessing the hash table takes up most part of C_1, making the computation dominating. Therefore,

$$T_{com1} = \frac{W|R|}{P_1} + C_1 \quad (8)$$

In the probe phase, tuples in relation S are read, and after a same hash function being run on each tuple's key, the corresponding hash table buckets are read from the hash table for probing. Therefore, only a read is needed in this phase and the memory access cost is:

$$T_{mem2} = \frac{W|S| + \alpha G|S|}{B}$$

Similar with 8, the cache miss penalty is significant in the probe phase. The computation cost in this phase is:

$$T_{com2} = \frac{W|S|}{P_2} + C_2$$

The total cost for parallel no partitioning hash join is:

$$\begin{aligned} T_{NP} = & max\left\{\frac{(W + 2\alpha G)|R|}{B}, \frac{W|R|}{P_1} + C_1\right\} \\ & + max\left\{\frac{(W + \alpha G)|S|}{B}, \frac{W|S|}{P_2} + C_2\right\} \end{aligned} \quad (9)$$

3.3.2 Radix Partitioning Hash Join

The radix partitioning hash join we mention in this section consists of three phases, the first partitioning phase, the second partitioning phase, and the build-probe phase. To reduce the potential contention between different threads in the first partitioning phase, an extra histogram phase is added to build a global histogram for each thread. The second partitioning phase is similar with the first pass. One difference is that in the second partitioning phase, the histogram is built locally for each thread. The build-probe phase can be divided into a couple of sub-phases since it repeats the build phase and the probe phase for each partition. However, we can combine the build phase for all partitions into a build phase during the cost calculation, as well as the probe phase. As not all the partitions need to do the build-probe phase, the data needed to process and access in this phase is some percentage of the relation size. To simplify the calculation in this case study, we assume that it needs to operate the build-probe phase on all the partitions, namely, the whole relations. The total cost of radix partitioning hash join is:

$$T_{RP} = T_{histogram} + T_{partition1} + T_{partition2} + T_{build} + T_{probe}$$

Following reasoning similar to the no-partition case the total cost of radix partitioning hash join is:

$$\begin{aligned} T_{RP} = & max\left\{\frac{W(|R| + |S|)}{B}, \frac{W(|R| + |S|)}{P_1} + C_1\right\} \\ & + max\left\{\frac{3W(|R| + |S|)}{B}, \frac{W(|R| + |S|)}{P_2} + C_2\right\} \\ & + max\left\{\frac{4W(|R| + |S|)}{B}, \frac{W(|R| + |S|)}{P_3} + C_3\right\} \\ & + max\left\{\frac{W|R|}{B}, \frac{W|R|}{P_4} + C_4\right\} \\ & + max\left\{\frac{W|S|}{B}, \frac{W|S|}{P_5} + C_5\right\} \end{aligned} \quad (10)$$

We briefly describe the phases to explain each of the components of this equation.

The histogram phase (phase 1) is done separately on both relations. For each relation, the processors read all the tuples and builds the histogram. The histogram is small enough to store in the cache and there is no need to write back to the memory. So, only a sequential read is done on both tables. While included in the formula we expect the compute cost to be dominated by the transfer cost.

During the first partitioning phase (phase 2), all tuples in both relations are scanned and assigned to the corresponding partition, and then the partitions are written back to the memory. Even though the access to each partition is random, the cache is large enough to maintain a small bucket for each partition. When the small bucket is full, it is written back to the main memory and the another empty bucket will be read for other tuples. So, the granularity-size data block is fully used in this phase. Assuming the size of the partitioned relation is equal to the original relation size, there is a full read and a full write for both relations. Also, the data block should be read before it is written back. If the number of partitions is too large, it may cause many TLB misses, the overhead of which can cause the computation cost dominating. However, multi-passes approaches can be used to limit the fanout of each pass partitioning, minimizing the number of TLB misses. Therefore, in the later parts of this paper, we assume that the memory access costs dominate the total running time.

The second pass partitioning phase (phase 3) is similar with the the first pass partitioning, but here we include the second histogram build with the partitioning.

For the build-probe phase (phase 4 & 5), we assume that in this case the hash table for each partition is small enough to fit in the cache and there are few cache misses when accessing the hash table. Otherwise, the cache misses penalty may take up a significant part of the cost, turning it back the the case of no partitioning hash join for each partition. The build sub-phase (phase 4) and probe sub-phase (phase 5) are modeled separately.

3.3.3 Non-shared Channels

If the two algorithms analyzed in section 3.3.1 and 3.3.2 run on a non-shared memory channel machine, the overlap

between read and write should be considered. In the no-partitioning hash join, the write only happens during the build phase. The cost for transferring data depends on the dominating part between reading both the tuples and the hash table and writing the hash table. So, formula 7 is changed to:

$$T_{mem1} = max\left\{\frac{W|R| + \alpha G|R|}{B_r}, \frac{\alpha G|R|}{B_w}\right\}$$

Correspondingly, the overall cost of no-partitioning hash join is modified to:

$$\begin{aligned} T_{NP} = & max\left\{\frac{W|R| + \alpha G|R|}{B_r}, \frac{\alpha G|R|}{B_w}, \frac{W|R|}{P_1} + C_1\right\} \\ & + max\left\{\frac{(W + \alpha G)|S|}{B_r}, \frac{W|S|}{P_2} + C_2\right\} \end{aligned} \tag{11}$$

The write happens during all partitioning phases in the radix partitioning hash join. As we analyze above, the read and write amount for the first partitioning phase are $2W(|R| + |S|)$ and $W(|R| + |S|)$, respectively, the memory access time for either pass should change to:

$$T_{mem2} = max\left\{\frac{2W(|R| + |S|)}{B_r}, \frac{W(|R| + |S|)}{B_w}\right\}$$

Similarly, cost for the second partitioning phase is:

$$T_{mem3} = max\left\{\frac{3W(|R| + |S|)}{B_r}, \frac{W(|R| + |S|)}{B_w}\right\}$$

Consequently, the overall performance for radix partitioning hash join is:

$$\begin{aligned} T_{RP} = & max\left\{\frac{W(|R| + |S|)}{B_r}, \frac{W(|R| + |S|)}{P_1} + C_1\right\} \\ & + max\left\{\frac{2W(|R| + |S|)}{B_r}, \frac{W(|R| + |S|)}{B_w}, \frac{W(|R| + |S|)}{P_2} + C_2\right\} \\ & + max\left\{\frac{3W(|R| + |S|)}{B_r}, \frac{W(|R| + |S|)}{B_w}, \frac{W(|R| + |S|)}{P_3} + C_3\right\} \\ & + max\left\{\frac{W|R|}{B_r}, \frac{W|R|}{P_4} + C_4\right\} \\ & + max\left\{\frac{W|S|}{B_r}, \frac{W|S|}{P_5} + C_5\right\} \end{aligned} \tag{12}$$

3.4 Model Analysis

In this section, we validate the proposed model and show how to make a prediction with the model. We assume we can use techniques such as prefetch and multi-pass partitioning, to reduce the cost of computation, and both algorithms are limited by the memory bandwidth. We use a 16B tuple size W and 64B cache line size G as an example. We assume the hash table, the tuples and the meta data are within the same cache line. Each access to a 16B tuple in the hash table causes transfer of a 64B block, or $\alpha = 1$. For a non-shared memory channel machine, according to formula 9 the running time of the no-partitioning hash join is:

$$T_{NP} = \frac{(W + 2\alpha G)|R|}{B} + \frac{(W + \alpha G)|S|}{B} = \frac{144|R| + 80|S|}{B}$$

In a two-pass partitioning radix hash join, assume a good partition number is chosen to reduce the cache misses and TLB misses, so that the algorithm is memory bandwidth limited. According to formula 10, the running time of the radix partitioning hash join is:

$$\begin{aligned} T_{RP} = & \frac{W(|R| + |S|)}{B} + \frac{3W(|R| + |S|)}{B} + \frac{4W(|R| + |S|}{)}B \\ & + \frac{W|R|}{B} + \frac{W|S|}{B} = \frac{144(|R| + |S|)}{B} \end{aligned}$$

We can conclude that for this case the no-partition hash join will always perform better.

There are some differences when running the algorithms in a non-shared channel machine since the read and write have overlap. We select a cache line size of 128 Bytes. Suppose the read bandwidth is twice as large as the write bandwidth. That means in one phase if the read data amount is not twice larger than the write data amount, that phase is dominated by the write cost. According to formula 11, the cost of the no-partitioning hash join in a non-shared channel machine is:

$$T_{NP} = \frac{\alpha G|R|}{B_w} + \frac{(W + \alpha G)|S|}{B_r} = \frac{128|R|}{B_w} + \frac{144|S|}{B_r}$$

Similarly, based on formula 12, the running time of the radix partitioning hash join in a non-shared channel machine is:

$$\begin{aligned} T_{RP} = & \frac{W(|R| + |S|)}{B_r} + \frac{W(|R| + |S|)}{B_w} + \frac{3W(|R| + |S|)}{B_r} \\ & + \frac{W|R|}{B_r} + \frac{W|S|}{B_r} = \frac{80(|R| + |S|)}{B_r} + \frac{16(|R| + |S|)}{B_w} \end{aligned}$$

It is also be useful to look at granularity effects . Assuming that the tuple does not exceed the size of a cache line (i.e., $\alpha = 1$), and setting $X = \frac{|S|}{|R|}$,

$$\begin{aligned} T_{NP} = & \frac{|R|}{B}((W + 2G) + (W + G)X) \\ T_{RP} = & \frac{|R|}{B}(W(1 + X) + W(3 + 3X) \\ & + W(4 + 4X) + W(1 + X)) \end{aligned}$$

solving $T_{NP} < T_{RP}$ yields

$$W > \frac{G}{8}(\frac{X + 2}{X + 1}) \tag{13}$$

especially when $X = \frac{|S|}{|R|}$ is large enough,

$$W_{break\_even} \simeq \frac{G}{8} \tag{14}$$

Thus, unlike the change in $\frac{|S|}{|R|}$, change in the tuple size plays a significant role in which algorithm performs best. For G=64B, the break-even tuple size is about 8B.

4. EXPERIMENT SETUP

4.1 Platform

We validate the proposed model on two different multi-core machines. The HP Proliant DL-360P has 10 cores per node, with SMT2 configuration. Different with HP Proliant DL-360P, the IBM POWER8 S824L has fewer cores but more threads. Each of 4 NUMA nodes in the IBM POWER8 S824L has 5 cores with up to SMT8 setting per cores. Cache and TLB configurations are summarized in table 2.

Both platforms have multiple NUMA nodes. As we want to eliminate the NUMA effect, we only use one node in both

Table 2: Hardware Platforms Feature

	HP Proliant DL-360p Gen8	IBM POWER8 S824L
CPU	Intel Xeon E5-2670 v2 2.5GHz	IBM POWER8E 3.7GHz
Cores	10/20	5/20
Threads	20/40	40/160
Cache L1	32 KiB	64 KiB
L2	256 KiB	512 KiB
L3	25 MiB	8 MiB
TLB L1	64	48(96)
L2	512	256
L3	N/A	2048
Page size	4 KiB	4 KiB
Memory	192 GB	256 GB
Mem. BW	42/84 GB/s	Read: 76.8/307.2 GB/s Write: 38.4/153.6 GB/s
Cacheline	64B	128B

platforms. The memory bandwidth in HP Proliant DL-360P is around 42 GB/s within each NUMA node. It is shared between the read and write. The IBM POWER8 S824L has seperate read channels and write channels with each node supporting 76.8 GB/s read and 38.4 GB/s write bandwidth.

4.2 Workload

For evaluating both algorithms analyzed in section 3.3, we use the workload from [4] and extend it to support various tuple sizes from 4B to 128B. We assume the workload is in a column-oriented mode with each tuple in a form of $< key, value >$. Except for the 4B tuple case, which has a 4B key and no value, we assume an 8B key. We assume the key in each relation is unique and uniformly distributed.

5. EXPERIMENTAL RESULTS

In this section, the proposed model is evaluated. We first analyze both algorithms by tuning prefetch distance, radix bits, and SMT configuration. After that, we demonstrate the granularity effect by changing the tuple size. Finally, we vary the relation size ratio and show that the relation size ratio does not change the winner in the competition between no-partitioning hash join and radix partitioning hash join.

5.1 Impact of Software Prefetch

One of the dominating costs in a no-partitioning hash join is the result of cache misses. Since the latency is high for getting the data from main memory instead of from cache, cache misses cause stalls in the processor. The processor cannot continue to work until the data is returned. Using software prefetch is a way to reduce the cache miss penalty.

Figure 1 shows the impact of using prefetch in the no-partitioning hash join. A prefetch distance i means that when processing the current tuple, the processors do the prefetch of the hash bucket for the next ith tuple. We can see from the figure that, the performance improves more than 25% in the Intel machine and 35% in the POWER8 machine, respectively. For the Intel machine, the performance remains stable when the prefetch distance reaches 4, and after 10, the performance nearly doesn't increase any more. This is because it reaches the memory bandwidth limitation. The prefetch effect is more obvious in the POWER machine. The running time drops sharply from no-prefetch to only adopting a prefetch distance of 1. In the rest of the paper, the no-partitioning hash join uses prefetch distance of 10 in the Intel machine and 6 in the POWER machine unless otherwise specified.

The performance improvement is the result of latency hiding. When randomly accessing the hash table either during the build phase or the probe phase, as the cache is not large enough to hold the whole hash table, a lot of cache misses occur. Doing prefetch can access the data before it is needed, reducing the wait cycles for the data response. If the prefetch distance is large enough, it can reduce most of the waiting time. A drawback of utilizing prefetch is the increase of the number of instructions. However, this penalty is far less than the performance increase introduced by prefetch.

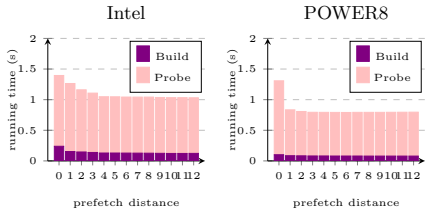


Figure 1: Running time for no-partitioning hash join with different prefetch distance (Intel, 10 cores, 20 threads, POWER8, 5 cores, 40 threads)

5.2 Number of Partitions and Partitioning Passes

Radix partitioning hash join is a hardware-sensitive algorithm. The radix bit, indicating the number of partitions ($2^{radix\ bit}$), is an important tuning parameter. It should be well configured to gain good performance. On the one hand, If the partition number is too small, the size of each partition may not fit in the cache, leading to cache misses. On the other hand, if the number of partition is too large, each partitioning pass may cause a lot of TLB misses.

Figure 2 shows how the performance changed running against the Intel and the POWER8 machine by varying the radix bit. We can see that, in both figures, the partitioning time increases when the radix bit goes up, along with the build-probe time decreasing. The best trade off radix bit configuration for the Intel machine is 10, after which the partitioning time jump sharply due to the TLB miss penalty. The POWER8 machine is more robust with respect to this parameter. A radix bit of 14 is the best configuration, and when the radix bit is larger than 19, the partitioning time increases significantly. The build-probe time, on the other hand, declines rapidly at the beginning when radix bit increases beyond 6. The reduction of cache misses contributes to this performance enhancement. However, the build-probe cost does not keep shrinking when the partition number is too large (radix bit is 20 in both machines). This is because the computation for each partition takes up a larger percentage of the overhead and starts to dominate.

5.3 SMT effect

Figure 3 demonstrates the curve of the SMT effect in the POWER8 machine. We run the no-partitioning hash join both with and without prefetch, and radix partitioning hash join. We can see that the no-partitioning hash join benefits

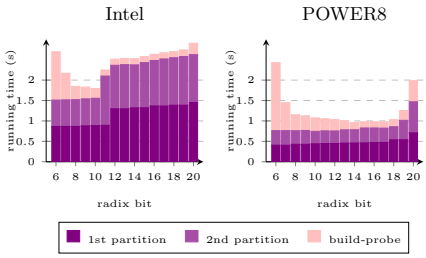


Figure 2: Running time for partitioning hash join with different radix bits (Intel, 10 cores, 10 threads, POWER, 5 cores, 15 threads). The cost for building histogram is included in the first partitioning phase.

4

from the SMT, especially the case without prefetch. This logical because both threading and prefetching hide memory latency. However, it keeps stable when the threads number is more than 20 for the no-partitioning hash join without prefetch and more than 10 for that with prefetch. Conversely, the radix partitioning hash join is more oblivious to the SMT configuration. The running time does not change a lot when the SMT is in different configurations. It even increases slightly when the thread number reaches 20 which means SMT4 configuration. For the rest of this paper, we use SMT8 for the no-partitioning hash join and SMT3 for the radix partitioning hash join in the POWER8 machine as they are the best selection based on this experiment.

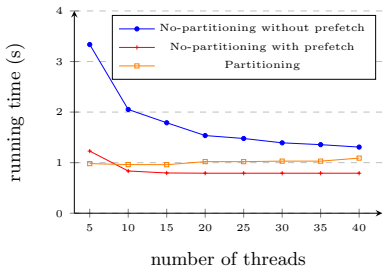


Figure 3: Running time with different numbers of threads (POWER, 5 cores)

5.4 Granularity

Because we cannot change the cache line size we vary the size of the tuples. Figure 4 shows how the running time changes as a function of the tuple size changing from 4B to 128B for measured data and our model on the Intel machine. For larger tuple sizes the no-partitioning hash-join performs better. This is expected because for larger sizes there is increased bandwidth pressure and the amplification benefit of the partitioned hash join is reduced. The measured and predicted curves have similar shape. The measured break-even point is around 8B, close to 9B provided by the model.

Figure 5 shows the performance of both hash join algorithms running on the POWER8 machine. The running time of radix partitioning hash join increases smoothly in a same pace with the tuple size. The running time of the no-partitioning hash join, showing a different curve, climbing stably at the beginning when the tuple size is small. From a 32B tuple size to 64B size, there is a jump, almost doubling the running time, which is as shown in the 32B size in the Intel machine test. This is because the tuple and the meta data in a same hash table bucket are split over two cache lines. Consequently, each access to the hash table actually transfers two cache lines instead of one. Measured radix partitioning hash join win when the tuple size is 16B or smaller, while the no-partitioning hash join win when the tuple size is 32B or larger. The modeled break even is around 21B which matches the prediction analyzed in section 3.4

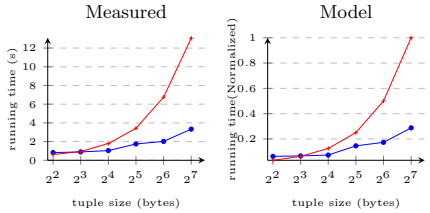


Figure 4: Running time with different tuple size (Intel, 10 cores, radix (red) and no-partition (blue))

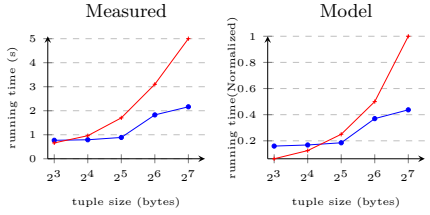


Figure 5: Running time with different tuple size (POWER, 5 cores, radix (red) and no-partition (blue))

5.5 Relation Size Ratio

In this section, we explore how the relation size ratio impacts the performance. We run both algorithms in both platforms with various $|R|$ from $16 * 2^{20}$ to $256 * 2^{20}$ and keep the size of S at $256 * 2^{20}$. Figure 6 and figure 7 summarize the prediction trend of both algorithms' performance with different R size according to the proposed model. We can see from figure 6, consistent with the model, that when the tuple size is 16B, the no-partitioning hash join is always better than the radix partitioning hash join. Conversely, when the tuple size is 8B, the radix partitioning hash join outperforms the no-partitioning hash join. In figure 7, the radix partitioning hash join runs faster than no-partitioning hash join both when the tuple size is in 8B and in 16B.

Figure 6 and figure 7 illustrate the experiment results of the relation size ratio effect. Both figures have similar shapes with the proposed model prediction. The curve of no-partitioning hash join in 8B tuple size is close to that

in 16B tuple size due to the granularity effect, while the running time doubles for the radix partitioning hash join when the tuple size changes from 8B to 16B. The radix partitioning hash join runs slower than the estimation by the model, likely because the overlap between computation and memory accesses cannot be ignored. So all the curves for radix partitioning hash join in the model should be shifted up. Consequentially, as shown in figure 6 and figure 7, the no-partitioning hash join can win in some cases when the difference between the size of both relations are large.

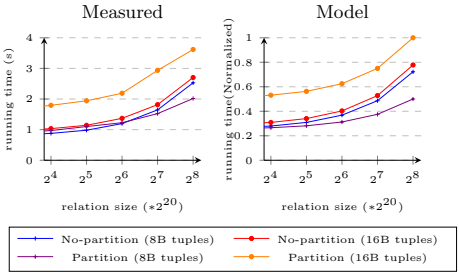


Figure 6: Running time with different relation size ratio (Intel, 10 cores)

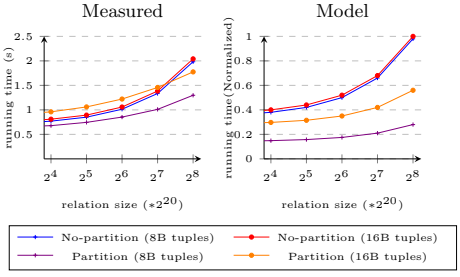


Figure 7: Running time with different relation size ratio (POWER, 5 cores.)

6. CONCLUSIONS AND FUTURE WORK

In this paper, we analyze the performance of in-memory hash join algorithms on multi-core platforms. We discuss the factors that impact the performance and find that the granularity is one of the main impact factors. Based on this finding, we propose a performance model considering both computation and memory accesses. According to the model, no-partitioning hash join should be more competitive than the partitioning hash join when the tuple size is large and the granularity is small. The results show that our model can accurate predict the winner between no-partitioning hash join and partitioning hash join. In the future, we expect to extend the proposed model to account for NUMA effects and skewed data distributions.

7. ACKNOWLEDGMENTS

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC, visualization, or storage resources that have contributed to the research results reported within this paper. URL: http://www.tacc.utexas.edu

8. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[2] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, pages 362–373. IEEE, 2013.

[3] Ç. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on modern processor architectures. *IEEE TKDE*, 27(7):1754–1766, 2015.

[4] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*, pages 37–48. ACM, 2011.

[5] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM TODS*, 32(3):17, 2007.

[6] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *SIGMOD*, pages 1–8. ACM, 1984.

[7] L. M. Haas, M. J. Carey, M. Livny, and A. Shukla. Seeking the truth about ad hoc join costs. *VLDB*, 6(3):241–256, 1997.

[8] K. A. Hua, W. Tavanapong, and Y.-L. Lo. Performance of load balancing techniques for join operations in shared-noting database management systems. *Elsevier Journal of Parallel and Distributed Computing*, 56(1):17–46, 1999.

[9] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: fast join implementation on modern multi-core CPUs. *VLDB*, 2(2):1378–1389, 2009.

[10] H. Lang, V. Leis, M.-C. Albutiu, T. Neumann, and A. Kemper. Massively parallel NUMA-aware hash joins. In *IMDM*, pages 3–14. Springer, 2015.

[11] F. Liu and S. Blanas. Forecasting the cost of processing multi-join queries via hashing for main-memory databases. In *SoCC*, pages 153–166. ACM, 2015.

[12] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *IEEE TKDE*, 14(4):709–730, 2002.

[13] E. Omiecinski. Performance analysis of a load balancing hash-join algorithm for a shared memory multiprocessor. In *VLDB*, pages 375–385, 1991.

[14] J. M. Patel, M. J. Carey, and M. K. Vernon. Accurate modeling of the hybrid hash join algorithm. In *SIGMETRICS*, pages 56–66. ACM, 1994.

[15] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *SIGMOD*, pages 1961–1976. ACM, 2016.

[16] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing.

5

ACCELERATING HASH JOINS

SUMMARY

The previous chapter gave a detailed study of performance analysis on hash joins. According to the study, we can can draw a preliminary conclusion that FPGAs can also accelerate some of the memory-intensive operations such as hash joins. However, it is not trivial to use this knowledge in designing a hash-join accelerator on FPGAs. In addition, it is interesting to know how much performance improvement can be gained from implementing such operations on FPGAs. To answer these questions, this chapter studies the design of an FPGA-based hash-join accelerator.
Recent studies on hash-join algorithms implemented in reconfigurable logic have improved their throughput performance. However, performance limitations caused by random accesses to the main memory creates a bottleneck that limits the performance. This chapter presents an FPGA-based high-throughput hash join accelerator. The proposed accelerator stores the hash table that needs to be randomly accessed in HBMs, where all HBM channels operate independently to enable full HBM bandwidth. Furthermore, the accelerator uses a pre-partitioning method to steer HBM access requests to the corresponding HBM channel to avoid the traffic contentions across different HBM channels, and thus mitigating memory access as a bottleneck to limit performance.

The content of this chapter is based on the following paper:

J. Fang, J. Lee, Z. Al-Ars, H.P. Hofstee, *Optimizing FPGA-based Hash Joins with HBMs*, ready for submission.

OPTIMIZING FPGA-BASED HASH JOINS WITH HBMS

Jian Fang
Delft University of Technology
Delft, the Netherlands
j.fang-1@tudelft.nl

Jinho Lee
Yonsei University
Seoul, Korea
leejinho@us.ibm.com

Zaid Al-Ars
Delft University of Technology
Delft, the Netherlands
z.al-ars@tudelft.nl

H. Peter Hofstee
IBM Austin
Delft University of Technology
Texas, USA
h.p.hofstee@tudelft.nl

5

ABSTRACT

Hash join is a commonly used database operator and significantly impacts the database performance. Recently, FPGA community has studied and developed new hardware algorithms to improve its throughput. However, the performance limitations caused by the random access to the main memory is challenging this progress. In this paper, we present a high throughput FPGA-based hash join accelerator. The proposed architecture stores the hash table in HBMs and allows all HBM channels to operate independently. We propose a pre-partition method to steer the traffic to the corresponding channels to relieve contentions. In a result, the proposed method should efficiently utilize the HBM bandwidth.

***K*eywords** hash join · FPGA · HBM · random access · database

1 Introduction

Hash join is a commonly used operation in database systems and significantly impacts the performance of databases. It combines two tables into one compound table under specific constraints, where the most frequently used one is to combine two tables by a common field, called equi-join. Recently, researchers have been looking into using application-specific processors to improve the hash join performance, among which the FPGA is a promising one. Previous studies [1–3] have explored the potential of using FPGAs to accelerate hash joins, and show some performance improvements, but yet still face challenges.

For small size problems, the hash table can be stored in the FPGA internal memory or the *block RAM* (BRAM) which provides lower-latency but higher-bandwidth data accesses than the main memory. Thus, an optimized pipelined FPGA designs should be able to keep up with the rate of reading the data from the main memory. However, the BRAMs in an FPGA typically contain a few mega bytes, which limits the problem size (the smaller table) to tens to hundreds thousand tuples.

For large size problems for which the hash table would exceed the BRAM size in an FPGA, the hash table can be stored in the main memory. Building and probing an in-memory hash table requires a large number of random accesses to the main memory, which might lead to poor throughput performance. Another way to handle the large size problems is to do a partitioning phase on both tables before joining them. If the partition is small enough to store its hash table in the FPGA internal memory, it can make use of the locality to avoid random accesses to the main memory. A drawback of this method is that it demands an extra partition phase or extra passes transferring the data between the main memory and the FPGA, especially for those large problem cases that needs multiple partitioning phases.

In this paper, we explore the potential of using a new kind of memory, the *high bandwidth memory* (HBM), to accelerate FPGA-based the hash joins. Integrating HBMs with FPGAs brings FPGAs large capacity local memory along with

huge amount of bandwidth. One of the latest Xilinx FPGAs [4] can support up to 32 channels accessing two 4GB HBM stacks at an aggregate peak bandwidth of 460GB/s. Consequently, it is possible to store larger hash tables in the FPGA side (hundred million tuples), meaning that larger size problems can be handle within the FPGA. Even if the HBM is not large enough to keep the hash table of the smaller table, utilizing HBM for the partitioning hash join can reduce the requirement for the number of partitions, and thus fewer partitioning passes are required. However, to efficiently utilize such huge bandwidth post two main challenges. First, as the 460GB/s bandwidth is an accumulation of all HBM channels' bandwidth, a design should activate as many channels as possible. Second, traffic across different channels might cause contention which will lead to significant performance drops [5]. This paper presents a new FPGA-based hash join accelerator architecture. The proposed accelerator stores the hash table in HBM and utilizes a pre-partition method to steer the traffic to the appropriate memory channel to reduce the contention. Specifically, we make the following contributions.

- We present a hash join architecture on the FPGA that can efficiently use the HBM bandwidth.
- We propose a pre-partition method to steer the traffic to the corresponding channels to relieve contentions.
- We present a performance model for the proposed hash join architecture. Based on the estimation, the proposed method can reach 22GB/s throughput which is limited by the host-to-accelerator interface.

2 Related Work

The studies on optimization of the hash join performance started early on the CPU architecture. There are two main classes of hash join algorithms, including no-partitioning hash joins and partitioning hash joins. Previous studies have been done discussing the impact of cache misses [6], TLB misses [7], the *Non-uniform memory access* (NUMA) effect [8], and the granularity effect [9], as well as comparing both classes of hash join algorithms [10, 11].

In the context of FPGA, the hash join performance relies on the hash table organization. An efficient hash table structure can avoid stalls in the pipeline and allows continuous data processing. The hash join in [1] utilizes a bit vector table to store the hash entries and an address table to maintain the linked lists. The access to the hash bit vector table and that to the address table are placed in different stages which allows non-stall processing. The study in [12] proposed a grouping method to construct a biggest hash table in BRAMs. In this method, the BRAMs are diveded into groups that are linked by a chain. Tuples with hash collisions will be assigned to the next group until the hash table overflows. Note that, both methods construct the hash table in BRAMs, which limits the problem size to only a few hundred thousand tuples.

If the hash table is too large to keep in BRAMs, the hash table might need to be kept in the main memory which might lead to random accesses. To reduce the penalty caused by the random accesses, the BRAM can be used to act as a cache [3] and store the most frequently used entries locally. However, the cache miss ratio might increase when the hash table becomes larger. Thus, for large data sets, the system might not be able to benefit from the cache mechanism. The study from [2] proposes a multithreading hash join algorithm running on the Convey-MX system with memory bandwidth of 76.8GB/s, which achieves a throughput of up to 12GB/s. In this architecture the join engine maintains thousands of threads in the FPGA to hide hide the long latency accessing the main memory. However, this method might suffer bandwidth waste and become less efficient because of granularity effect, or data transferred in practice is larger than the request amount [9]. Another way to avoid the random accesses is to partition the data sets into small enough partitions, so that the hash table of each partition can be built in the FPGA internal memory. The partitioning method proposed in [13] can partition the data into eight thousand partitions which can saturate the QPI bandwidth (6GB/s) and can be scaled up to fit faster interfaces. However, due to the limited size of BRAMs in the FPGA, large tables need to be partitioned into numerous small partitions. Thus, it might require multiple passes of partitioning which reduces the end-to-end performance.

In this paper, we focus on medium size hash joins for which the hash table can be stored in the HBMs. For larger data sets, as the HBM can keep much larger hash tables than the BRAM, fewer partitions are requied during the partitioning phase. Consequently, it can handle much larger data sets(e.g. several TB), within one pass partitioning. To the best of our knowledge, this paper present the first hash join on HBM in the context of FPGAs. Previously, using HBM to optimize hash join are studied in the CPU [14] and GPU [15] architecture.

3 Proposed Architecture

The proposed hash join accelerator consists of a build engine to build the hash table and a probe engine to do the probing and matching. The details are shown in Fig. 2 and Fig. 3, respectively. We assume both join tables are in column-oriented format where each tuple contains a fixed size key field and a fixed size value field (can be a pointer pointing to the original record in the database).

3.1 Hash Table Organization

In the proposed architecture, hash collisions are handled by the separate chaining technique using linked lists. The hash table organization is illustrated in Fig. 1. It contains a *head pointer table* (HPT) to store the hash entries and a *linked list table* (LLT) to hold the linked lists. Both tables are stored in the HBM. Each entry in the HPT records the pointer that points to the corresponding linked list, as well as the first tuple of each linked list. Similarly, each entry in the LLT contains the collision tuple and a “next" pointer. Reaching a next pointer of an “END" means it is the tail of the linked list, and there is no more tuple in the rest of the linked list. When updating the linked list, the new tuple is inserted from the head, so that it can update the hash table in constant time during the build phase.

The hash table is built in the build phase. It first reads the hash entry according to the hash value of the new tuple calculated by the hash function. Then it writes the new tuple to the hash entry along with the updated head pointer. Meanwhile, the old data read from the hash entry is copied to the empty location in the LLT. The next empty location then move to a next slot (in sequence). When probing the hash table, the first step is to read the hash entry in the HPT and validate whether it finds a match. Then, it locates the linked list using the head pointer and traverses it based on the next pointer. The traversal procedure stops when the next pointer gives an “END" ($N - to - M$ mapping) or the first match is found ($N - to - 1$ mapping).

The hash table is spread over the HMB sections in a stripe style. Each section in the hash table corresponds to a section or a group of sections in the HBM, and the accesses to a specific hash table section can use its own channel. This way, it can avoid the contention caused by the accesses across different channels, and thus improve the bandwidth efficiency.

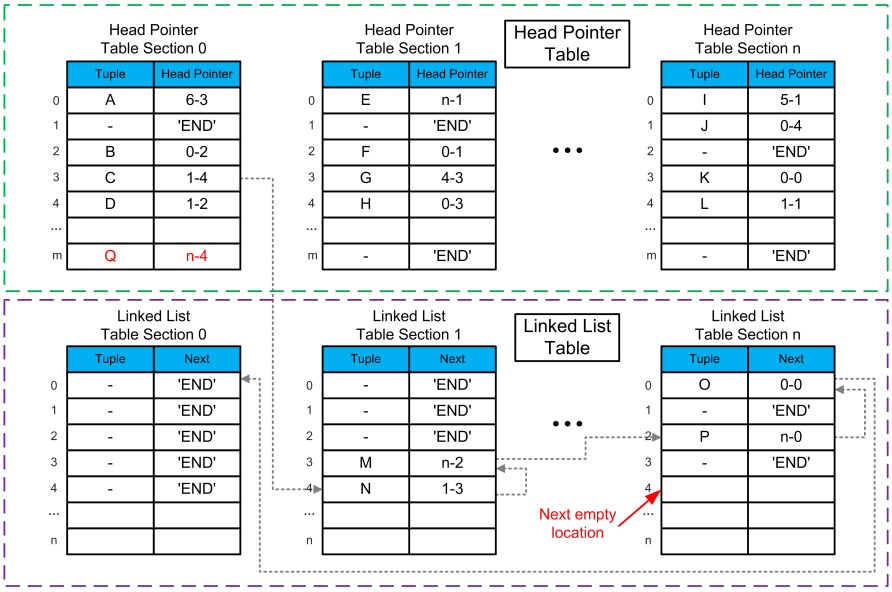


Figure 1: Hash Table Organization

3.2 the build engine

Fig. 2 demonstrates the components and architecture of the build engine. The build engine reads a line from the interface which contains multiple tuples. These tuples are sent to the hash function components to calculate the hash value using the same hash function. The hash value indicates the address of the hash entry. In the pre-partitioner, each tuple is steered to one of FIFOs based on the location of its hash entry in the HBM. Note that, it is possible that some of the tuples in a line might need to access the same HBM section. Thus, the FIFOs might receive unbalanced workloads.

In the next stage, the HPT update unit and the LLT update unit work together to build the hash table in the HBM. It works in three steps. First, the HPT update unit sends a read request to read the existing (old) hash entry base on the

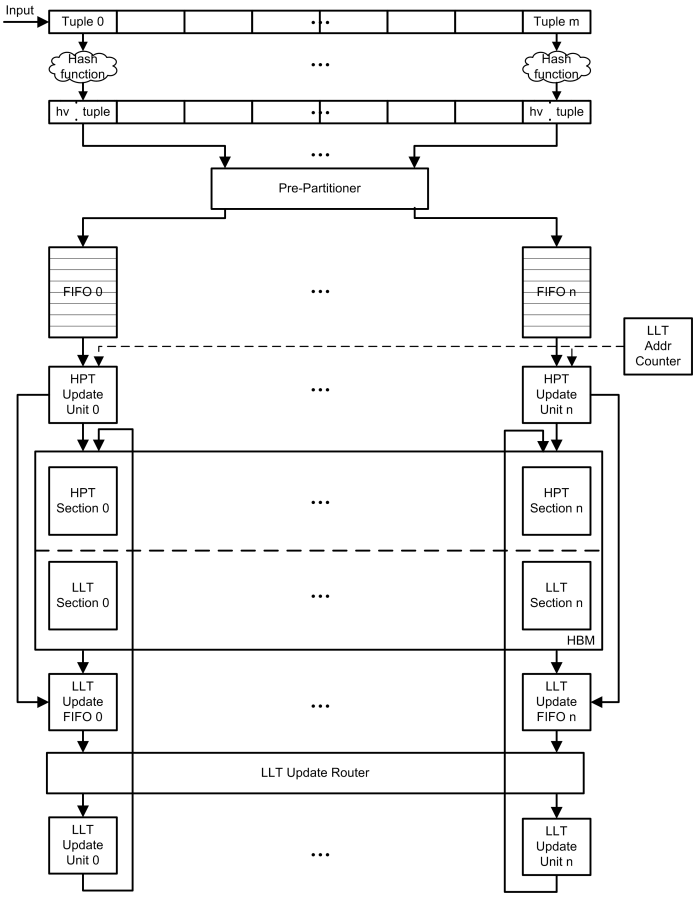


Figure 2: The Build Engine

address calculated from the hash value. After the read request is sent, a write request follows to update the hash entry. The new hash entry includes the tuple itself and the pointer that pointer to the linked list. As the linked list is updated from the head, this pointer can be decided when the next empty location is allocated, which is managed by the LLT address counter. In the third step, when the old hash entry is read, it is stored to the LLT update FIFO, after which the LLT update router drives it to the corresponding LLT update unit according to the LLT section it should be written. Then, an update/write request is sent to the HBM to update the linked list.

During the update of the HPT, it is possible that a read request might arrive faster that a previous write request on the same address. Even though the HBM memory controller can handle this dependency, the communication needs to follow the AXI rules between the user logic and the AXI interface on the HBM memory controller side, which means the user needs to take care the request order acknowledged by the AXI interface. In our design, we adopt a simple control mechanism by maintaining a unfinished write request window. Whenever a write to the HPT, it records this write in the window, while a write acknowledgement leads to remove this write record. With this unfinished write request window, if a read matches one of the addresses in the window, this read request will be stalled until the write record is removed from the window.

3.3 the probe engine

When the build phase is done, the system enters the probe phase. Fig. 3 describes the architecture of the probe engine. Most of components can be shared between the build engine and the probe engine such as the hash function units, FIFO, pre-partitioner, and the router. Similar to the build engine in the earlier stage, it reads a whole input line and assigns the tuples to the corresponding FIFO depending on the output of the hash function.

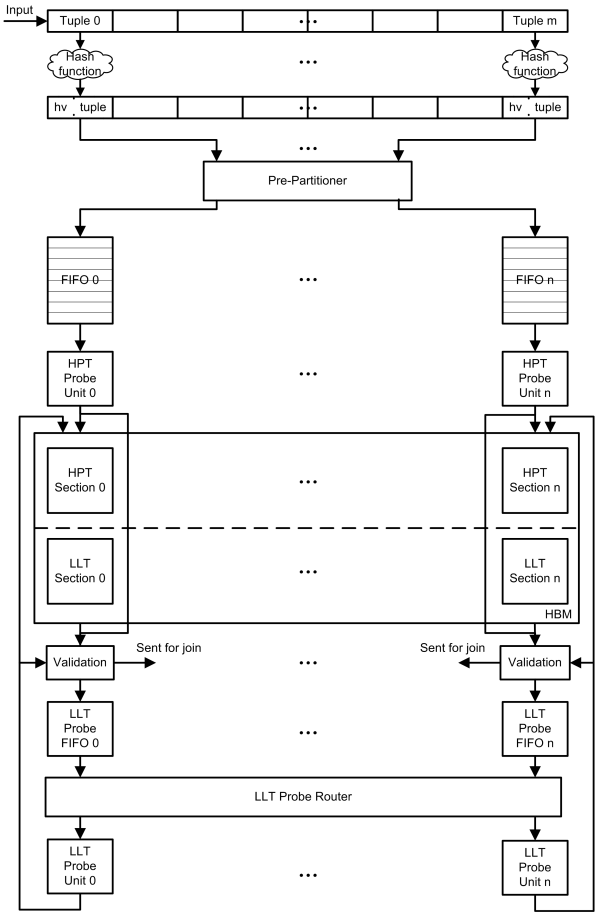


Figure 3: The Probe Engine

The HPT probe unit then sends a read request to read hash entry in the HPT and start the probing. The address of the hash entry is calculated by the hash value. The HPT probe unit also forwards the original input tuple to the validation unit, where the tuple will be validated. After the read data arrives, the validation unit compares the data read out from the HPT and the original tuple to find whether there is a match.

If no match is found, the next pointer, as well as the original data, will be sent to the LLT probe unit, generating a new read request and preparing for the validation on the next tuple in the linked list. Since the next tuple in the linked list might sit in a different section of the HBM, a router is required to drive this read request to the appropriate LLT probe unit. This procedure continues until a match is found or the probing reach the end of the linked list.

If a match is found, the validation unit sends both tuple for joining. In our experiment, we simply count the number of joined pairs instead of materializing the results, different applications would have different join result format requirement. Depending on the application, we can stop the probing on this linked list if the data set only allows one to one mapping or the application only needs to find one match. Otherwise, the traversal on the linked list will continue until it reaches the end of the chain.

4 Evaluation

4.1 Performance Model

Suppose the join is performed between table R and table S. We estimate the build phase throughput (T_{build}) and probe phase throughput (T_{probe}) of the proposed method separately using the table size (S_R and S_S) over the running time (t_{build} and t_{probe}).

$$T_{build} = \frac{S_R}{t_{build}},$$
$$T_{probe} = \frac{S_S}{t_{probe}}.$$

For the build phase, suppose the design is fully pipelined and the computation can be overlapped by the data transition. The running time of the build phase depends on either the accesses to HBM or to the host. That is

$$t_{build} = max\{t_{HBM}, t_{host}\},$$

where t_{host} can be simply calculated using the host to accelerator bandwidth, or $t_{host} = \frac{S_R}{B_{host}}$. During the build phase, as described in Section 3.2, the hash entries and linked lists need to be updated, which leads to two reads and one write. Thus,

$$t_{HBM} = \frac{3\alpha S_R}{B_{HBM}}.$$

Note that each random access to the HBM needs to follow the granularity rule [9]. Thus, we use α as an amplifying factor to cover the wasted data amount. Consequently, the throughput of the build phase can be explained as

$$T_{build} = min\{B_{host}, \frac{B_{HBM}}{3\alpha}\}.$$

Similarly, the probe phase requires one read to the HPT and at lease one read to the LLT depending by the collision degree. Assuming that each tuple gets β collisions, the throughput of the probe phase can be presented as

$$T_{probe} = min\{B_{host}, \frac{B_{HBM}}{(2+\beta)\alpha}\}.$$

4.2 Performance Estimation

Assume that the proposed method is implemented in a frequency of 250MHz in the Xilinx Virtex Ultrascale VU37P-2 device on an AlphaData ADM 9H7 board. We can enable both HBM stacks with combined total 32 HBM channels, which provide 8GB capacity and up to 460GB/s aggregate raw bandwidth (or 14GB/s per channel) through a 450MHz interface. If the interface frequency is adjusted to 250 MHz which is the same as the logic design, the HBM bandwidth becomes 255GB/s in total or 8GB/s per channel. We measure the random access throughput of the HBM using the memory control from [5]. We only gain 4GB/s random access throughput per channel or 128GB/s in total which is only half of the maximum.

In addition, we assume that both tables are stored in a column-oriented format that each tuple contains an eight-byte key and an eight-byte value. As the access to HBM is in a granularity size of 32B, each access to the hash entry or the linked list should be 32B, causing α to be 2. Supposing a perfect hash function is chosen so that there is no hash collision, or $\beta = 0$ Accordingly, setting $B_{HBM} = 128GB/s$, we can estimate the throughput of the build phase and the probe phase as,

$$T_{build} = min\{B_{host}, 21.3GB/s\},$$
$$T_{probe} = min\{B_{host}, 32GB/s\}.$$

If the size of the probe table is much larger than the smaller one, the running time will be dominated by the probe phase, and we can approximately use the probing throughput to represent the hash join throughput. If the proposed architecture connects to the host through the latest OpenCAPI 3.0 interface where an effective data rate about 22GB/s can be achieved, the throughput of the proposed architecture should be 22GB/s, which is limited by this host-to-accelerator interface.

5 Conclusions

FPGAs have been used to accelerate the hash join performance in database systems. However, the current studies either can only handle very small data sets or suffer performance drop due to the frequent random accesses to the main memory. In this paper, we proposed an hash join accelerator with HBMs. The proposed accelerator can activate all the HBM channels to gain large bandwidth. It allows all HBM sections operate independently. A pre-partitioning method is used to steer the HBM access requests to the corresponding HBM channels to avoid the traffic contentions across different HBM channels, and thus can efficiently utilize the bandwidth.

References

[1] Robert J Halstead, Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Sameh Asaad, and Balakrishna Iyer. Accelerating join operation for relational databases with fpgas. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 17–20. IEEE, 2013.

[2] Robert J Halstead, Ildar Absalyamov, Walid A Najjar, and Vassilis J Tsotras. Fpga-based multithreading for in-memory hash joins. In *CIDR*, 2015.

[3] Behzad Salami, Oriol Arcas-Abella, Nehir Sonmez, Osman Unsal, and Adrian Cristal Kestelman. Accelerating hash-based query processing operations on fpgas by a hash table caching technique. In *Latin American High Performance Computing Conference*, pages 131–145. Springer, 2016.

[4] Mike Wissolik, Darren Zacher, Anthony Torza, and Brandon Da. Virtex UltraScale+ HBM FPGA: A revolutionary increase in memory performance. *Xilinx Whitepaper*, 2017.

[5] Xilinx. AXI High Bandwidth Memory Controller v1.0. `https://www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf`, 2019. Accessed: 2019-08-01.

[6] Spyros Blanas, Yinan Li, and Jignesh M Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*, pages 37–48. ACM, 2011.

[7] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, pages 362–373. IEEE, 2013.

[8] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. Massively parallel NUMA-aware hash joins. In *IMDM*, pages 3–14. Springer, 2015.

[9] Jian Fang, Jinho Lee, Harm Peter Hofstee, and Jan Hidders. Analyzing In-Memory Hash Joins: Granularity Matters. In *Proceedings the 8th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures*, pages 18–25, 2017.

[10] Çağrı Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. Main-memory hash joins on modern processor architectures. *IEEE TKDE*, 27(7):1754–1766, 2015.

[11] Stefan Schuh, Xiao Chen, and Jens Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *SIGMOD*, pages 1961–1976. ACM, 2016.

[12] Takanori Ueda, Megumi Ito, and Moriyoshi Ohara. A dynamically reconfigurable equi-joiner on FPGA. *IBM Technical Report RT0969*, 2015.

[13] Kaan Kara, Jana Giceva, and Gustavo Alonso. FPGA-based data partitioning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 433–445. ACM, 2017.

[14] Constantin Pohl and Kai-Uwe Sattler. Joins in a heterogeneous memory hierarchy: Exploiting high-bandwidth memory. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, page 8. ACM, 2018.

[15] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. Hardware-conscious hash-joins on gpus. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 698–709. IEEE, 2019.

6

CONCLUSIONS

This chapter summarizes our contributions and draws conclusions. It also highlights possible future research directions. The summary and conclusions are presented in Section 6.1, after which future work is discussed in Section 6.2.

6.1. SUMMARY AND CONCLUSION

This thesis explores the potential of using FPGAs to accelerate database systems. It consists of an investigation of the limitations of recent FPGA-accelerated database systems, related technology development, and state-of-the-art database operator acceleration. Selecting decompression and hash join as examples, we design and implement accelerators for both operations, and perform an in-depth performance analysis. Specifically, we make the following contributions.

In **Chapter 2**, we give a comprehensive survey on using FPGAs to accelerate database systems, emphasizing the possibilities of new interface and memory-related technologies. We present the FPGA background in detail and analyze FPGA-accelerated database system architecture alternatives and point out the bottlenecks in different system architectures. After that, we discuss the reasons why FPGAs are not being widely used in the database field, followed by a study of memory-related technology trends which have the potential to make FPGAs attractive again. Thereafter, an overview of the state-of-the-art research on FPGA-accelerated database operators is presented. Finally, we discuss the major challenges and possible future research directions. We can summarize this chapter and draw conclusions as follows.

- Most of the overhead in a number of database applications is due to transferring data from the host to the FPGA, resulting in only a small speedup when the FPGA is integrated in database systems.
- In addition to the communication overhead, the weak programmability is a main engineering reason that prevents FPGAs from being widely used in the database field. Database vendors have switched to a more engineering-friendly accelerator, GPUs, which can also provide decent performance gains.
- With the new technologies including the interconnect and new types of memory, plus progress in FPGA design automation, FPGAs should become a reasonable accelerator for database systems.
- To push this progress a step forward, studies in different directions including new database architectures, new types of accelerators, deep performance analysis, and the development of tool chains is required.

In **Chapter 3**, we propose a “refine" technique and a “recycle" technique to achieve efficient high single-decompressor throughput by keeping only a single BRAM-banked copy of the history data and operating on each BRAM independently. We apply these two techniques to a widely used decompression algorithm in big data and database applications. We also proposed a proof-of-concept Parquet-to-Arrow converter architecture in FPGAs to improve the conversion throughput. We can summarize this chapter and draw conclusions as follows.

- The proposed method efficiently refines the tokens into commands that operate on a single BRAM and steers the commands to the appropriate one. Thus, it can activate multiple BRAMs and execute the commands independently.

- The recycle method is used where each BRAM command executes immediately and those that return with invalid data are recycled to avoid stalls caused by the RAW dependency.
- According to experimental results, the proposed Snappy decompressor which adopts these two techniques achieves up to 7.2 GB/s throughput per decompressor, with each decompressor using 14.2% of the logic and 7% of the BRAM resources of the Xilinx VU3P FPGA. One decompressor keeps pace with an NVMe device (PCIe Gen3 x4) on a small FPGA.
- The proposed Parquet-to-Arrow converter can achieve more than 7GB/s throughput which is limited by the the bandwidth of the connection to device memory. It can be extended to support more features such as decompression, which can benefit from integrating the proposed Snappy decompressor.

In **Chapter 4**, we analyze the performance of hash join algorithms by using a bandwidth-driven model and provide a guideline for choosing the right algorithm according to the dataset and architecture characteristics. We analyze and validate the different performance impact factors and point out the importance of the granularity factor. We propose a performance model that considers both computation and memory accesses according to these factors, and study different hash join algorithms and validate the proposed model in different processor architectures. We can summarize this chapter and draw conclusions as follows.

- We analyze the impact factors for the hash join algorithms. Based on the analysis, we point out that the granularity factor can significantly impact the performance of different hash join algorithms.
- We build a performance model based on these impact factors which considering both the computation and data transition. Based on the proposed model, we study a no-partitioning hash join and a radix partitioning hash join algorithm, and find out that no-partitioning hash join should be more competitive than the partitioning hash join when the tuple size is large and the granularity is small.
- We validate the model with hardware-based experiments on an x86 CPU and a Power8 CPU. The results show that our model can accurately predict the winner between no-partitioning hash join and partitioning hash join operations.

In **Chapter 5**, an FPGA-based hash join accelerator architecture that makes use of HBMs is presented. The proposed accelerator stores the hash table in HBM and utilizes a pre-partition method to steer the traffic to the appropriate memory channel to reduce the contention between the requests across different HBM channels, and thus improve the HBM bandwidth efficiency. We can summarize this chapter and draw conclusions as follows.

- We study the performance of HBMs. While the sequential memory accesses to HBMs can achieve the peak bandwidth if all HBM channels and sections are activated, random accesses suffer obvious performance drops, especially for the cases where the requests need to cross different channels.

- We present a hash join architecture on the FPGA that can efficiently use the HBM bandwidth. The proposed architecture enable all the HBM sections and channels, and maps each channel to only one HBM section to avoid requests crossing different channels.

- We propose a pre-partition method to steer the traffic to the corresponding channels to reduce contentions. As a result, the proposed method should efficiently utilize the HBM bandwidth

6.2. FUTURE WORK

In this section, we discuss the future work. Here, we list a number of possible recommendations to extend the work in this thesis in new directions.

- **Accelerator Design**
  New accelerator memory interfaces and other new hardware bring new features that could change the existing balance in accelerator-based systems. For example, new accelerator memory interfaces, such as OpenCAPI, bring fast speed in accessing the data sitting in the host memory. For some operators, more engines or more powerful engines are required which might consume the limited resources in an FPGA. Sort is one of them. Related work shows that, even though a strong sort engine [30] can merge multiple sorted streams at the speed of the interconnection bandwidth, the interface to efficiently fetch the data might consume too many resources to deploy in a practical system [31]. Thus, we need to look for a new design or even a new class of algorithms, e.g. the partitioning sort [32]. The partitioning sort is widely used in GPUs and in the cloud to sort large data sets. An efficient partitioner in an FPGA might be able to partition a TB-size data set in one pass into multiple MB to GB level partitions, while another pass can sort each set of partitions with corresponding ranges within the FPGA with the help of HBMs.

  Another interesting direction is to explore the potential of accelerating more complex operations such as multi-way join, regular path query, skyline, and even machine learning. Such operations are time-consuming but not well studied yet in the context of FPGAs. Related work extends the analysis of hash joins presented in this thesis to multi-way hash joins and draws similar conclusions. The most recent work [33] on multi-way hash joins also shows the value and the possibility of using reconfigurable hardware to accelerate such operations.

- **Performance Analysis**
  Unlike CPUs that has been well studied in the database field, FPGAs in databases are in the start-up stage. Deeper study is required on how FPGAs perform in the system, what the impact factors are, and how the architectures and different parameters influence the performance. Additionally, FPGAs might not be a perfect accelerator for all operations. Some operations might perform better on GPUs, and some even should be kept on the CPU. Thus, it is worth comparing the performance of different operations running against different architectures. The intuition is that latency-sensitive streaming processing applications might benefit

more from FPGAs than GPUs, since GPUs need to process data in batch mode with well-formatted data for throughput gains. However, this requirement can perfectly meet the features of the FPGA with data-flow designs where format conversion is often free. Moreover, FPGAs might be able to outperform GPUs in specific domains such as security and text data analytics.

- **Compilers and Frameworks**
  As stated earlier, an important reason why FPGAs are not widely used in the database field is the weak programmability. Programming FPGAs required developers to have full-stack skills, from high-level algorithm design to low-level circuit implementations. Consequently, the continued development of improved tool chains is crucial to further propagating the use of FPGAs. First of all, it is necessary for HLS tools to support stronger features and more features. For instance, supporting OpenMP in HLS tools might introduce the ability to compile parallel programs into FPGA accelerators that support shared memory. Second, developing query-to-hardware compilers is a good way to make FPGAs easy to use. These compilers can generate the circuits automatically from queries, and even help to manage the hardware resources, which can greatly reduce the workload for developers and shorten the development cycles. However, there are only a few recent such compilers in academia (e.g. [34]), let alone in the industry. Last, frameworks for specific functionality, such as supporting in-memory/storage data format and auto interface generation, might be a good direction to help hardware designers. Frameworks such as Fletcher [35] help by automatically generating the interface to support Apache Arrow [36], an in-memory standard data format. Consequently, hardware designers can focus on the kernel design and its optimization. A recent application of Fletcher is the Parquet-to-Arrow converter [37] which can convert a standard storage format, Apache Parquet [5], into Apache Arrow.

REFERENCES

[1] J. M. Hellerstein, M. Stonebraker, J. Hamilton *et al.*, "Architecture of a database system," *Foundations and Trends® in Databases*, vol. 1, no. 2, pp. 141–259, 2007.

[2] J. Fang, Y. T. Mulder, J. Hidders *et al.*, "In-memory database acceleration on fpgas: a survey," *The VLDB Journal*, pp. 1–27, 2019.

[3] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[4] Google, "Snappy," https://github.com/google/snappy/, accessed: 2018-12-01.

[5] Apache, "Apache Parquet," http://parquet.apache.org/, accessed: 2018-12-01.

[6] Apache, "Apache ORC," https://orc.apache.org/, accessed: 2018-12-01.

[7] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Application of hash to data base machine and its architecture," *New Generation Computing*, vol. 1, no. 1, pp. 63–74, 1983.

[8] A. Shatdal, C. Kant, and J. F. Naughton, *Cache conscious algorithms for relational query processing*. University of Wisconsin-Madison, Computer Sciences Department, 1994.

[9] S. Manegold, P. Boncz, and M. Kersten, "Optimizing main-memory join on modern hardware," *IEEE TKDE*, vol. 14, no. 4, pp. 709–730, 2002.

[10] Kinetica, "Kinetica," http://www.kinetica.com/, accessed June 3, 2018.

[11] T. Mostak, "An overview of mapd (massively parallel database)," *White paper. Massachusetts Institute of Technology*, 2013.

[12] Y. Yuan, R. Lee, and X. Zhang, "The yin and yang of processing data warehousing queries on gpu devices," *Proceedings of the VLDB Endowment*, vol. 6, no. 10, pp. 817–828, 2013.

[13] P. Francisco *et al.*, "The netezza data appliance architecture: A platform for high performance data warehousing and analytics," http://www.ibmbigdatahub.com/sites/default/files/document/redguide_2011.pdf, 2011.

[14] T. C. Scofield, J. A. Delmerico, V. Chaudhary *et al.*, "Xtremedata dbx: an fpga-based data warehouse appliance," *Computing in Science & Engineering*, vol. 12, no. 4, pp. 66–73, 2010.

[15] M. Wissolik, D. Zacher, A. Torza *et al.*, "Virtex ultrascale+ hbm fpga: A revolutionary increase in memory performance," *Xilinx Whitepaper*, 2017.

[16] D. Sidler, Z. István, M. Owaida *et al.*, "doppiodb: A hardware accelerated database," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1659–1662.

[17] P. A. Boncz, M. Zukowski, and N. Nes, "Monetdb/x100: Hyper-pipelining query execution." in *Cidr*, vol. 5, 2005, pp. 225–237.

[18] M. Owaida, D. Sidler, K. Kara *et al.*, "Centaur: A framework for hybrid cpu-fpga databases," in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*. IEEE, 2017, pp. 211–218.

[19] R. Mueller, J. Teubner, and G. Alonso, "Data processing on fpgas," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 910–921, 2009.

[20] C. Dennl, D. Ziener, and J. Teich, "Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*. IEEE, 2013, pp. 25–28.

[21] R. Mueller, J. Teubner, and G. Alonso, "Streams on wires: a query compiler for fpgas," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 229–240, 2009.

[22] B. Sukhwani, H. Min, M. Thoennes *et al.*, "Database analytics: A reconfigurable-computing approach," *IEEE Micro*, vol. 34, no. 1, pp. 19–29, 2014.

[23] B. Sukhwani, M. Thoennes, H. Min *et al.*, "Large payload streaming database sort and projection on fpgas," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2013 25th International Symposium on*. IEEE, 2013, pp. 25–32.

[24] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. ACM, 2014, pp. 151–160.

[25] M. Saitoh, E. A. Elsayed, T. Van Chu *et al.*, "A high-performance and cost-effective hardware merge sorter without feedback datapath," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 197–204.

[26] R. J. Halstead, I. Absalyamov, W. A. Najjar *et al.*, "Fpga-based multithreading for in-memory hash joins." in *CIDR*, 2015.

[27] T. Ueda, M. Ito, and M. Ohara, "A dynamically reconfigurable equi-joiner on fpga," *IBM Technical Report RT0969*, 2015.

[28] J. Stuecheli, "A new standard for high performance memory, acceleration and networks," http://opencapi.org/2017/04/opencapi-new-standard-high-performance-memory-acceleration-networks/, accessed: 2019-08-01.

[29] D. D. Sharma, "Compute Express Link," https://docs.wixstatic.com/ugd/0c1418_d9878707bbb7427786b70c3c91d5fbd1.pdf, 2019, accessed: 2019-08-01.

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

[31] Y. Mulder, "Feeding High-Bandwidth Streaming-Based FPGA Accelerators," Master's thesis, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands, 1 2018.

[32] G. C. Fossum, T. Wang, and H. P. Hofstee, "A 64GB sort at 28GB/s on a 4-GPU POWER9 node for 16-byte records with uniformly distributed 8-byte keys," in *Proc. International Workshop on OpenPOWER for HPC*, Frankfurt, Germany, June 2018.

[33] K. Olukotun, R. Prabhakar, R. Singhal *et al.*, "Efficient multiway hash join on reconfigurable hardware," *arXiv preprint arXiv:1905.13376*, 2019.

[34] R. Mueller, J. Teubner, and G. Alonso, "Glacier: a query-to-hardware compiler," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 1159–1162.

[35] J. Peltenburg, J. van Straten, M. Brobbel *et al.*, "Supporting columnar in-memory formats on fpga: The hardware design of fletcher for apache arrow," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2019, pp. 32–47.

[36] Apache, "Apache Arrow," https://arrow.apache.org/, accessed: 2019-03-01.

[37] L. van Leeuwen, "High-Throughput Big Data Analytics Through Accelerated Parquet to Arrow Conversion," Master's thesis, Delft University of Technology, 2019.

LIST OF PUBLICATIONS

International Journals

1. **J. Fang**, Y.T.B. Mulder, J. Hidders, J. Lee, H.P. Hofstee, *In-Memory Database Acceleration on FPGAs: A Survey*, International Journal on Very Large Data Bases (VLDBJ), 2019, https://doi.org/10.1007/s00778-019-00581-w.

2. **J. Fang**, J. Chen, J. Lee, Z. Al-Ars, H.P. Hofstee, *An Efficient High-Throughput LZ77-Based Decompressor in Reconfigurable Logic*, submitted to the Journal of Signal Processing Systems.

Conferences and Workshops

1. **J. Fang**, J. Chen, J. Lee, Z. Al-Ars, H.P. Hofstee, *Refine and Recycle: A Method to Increase Decompression Parallelism*, The 30th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), New York, NY, USA, July 15-17, 2019

2. **J. Fang**, J. Chen, J. Lee, Z. Al-Ars, H.P. Hofstee, *A Fine-Grained Parallel Snappy Decompressor for FPGAs Using a Relaxed Execution Model*, 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), San Diego, CA, USA, April 28-May 1, 2019

3. **J. Fang**, J. Chen, Z. Al-Ars, H.P. Hofstee, J. Hidders, *Work-in-Progress: A High-Bandwidth Snappy Decompressor in Reconfigurable Logic*, 2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS), Torino, Italy, Sep 30-Oct 5, 2018

4. **J. Fang**, Y.T.B. Mulder, K. Huang, Y. Qiao, X. Zeng, H.P. Hofstee, J. Lee, J. Hidders, *Adopting OpenCAPI for High Bandwidth Database Accelerators*, The Third International Workshop on Heterogeneous High-performance Reconfigurable Computing, Denver, CO, USA, Nov 17, 2017

5. **J. Fang**, J. Lee, H.P. Hofstee, J. Hidders, *Analyzing In-Memory Hash Joins: Granularity Matters*, 2017 International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS), Munich, Germany, Sep 1, 2017

6. L.T.J. van Leeuwen, J. Peltenburg, **J. Fang**, Z. Al-Ars, H.P. Hofstee, *High-throughput Conversion of Apache Parquet Files to Apache Arrow In-Memory Format using FPGAs*, Doorn, the Netherlands, 2019 International Conference on Computing Systems, June 3-5, 2019

7. **J. Fang**, J. Lee, Z. Al-Ars, H.P. Hofstee, *Optimizing FPGA-based Hash Joins with HBMs*, ready for submission.

CURRICULUM VITÆ

Jian FANG

Jian Fang was born in Guangdong, China in 1989. He received his B.Sc. degree in computer science and technology in 2011 from the National University of Defense Technology, Changsha, China. After that he received M.Sc. degree in computer science and technology in 2013 from the same university under the supervision of Prof. Weixia Xu. In September 2014, he joined the Department of Quantum and Computer Engineering at the Faculty of Electrical Engineering, Mathematics, Computer Science in Delft University of Technology as a PhD student under the supervision of Prof. H. Peter Hofstee and Dr. Zaid Al-Ars. His research interests include high performance computing, database analytics, and heterogeneous computing.

SIKS DISSERTATIEREEKS

2011 01 Botond Cseke (RUN), Variational Algorithms for Bayesian Inference in Latent Gaussian Models
02 Nick Tinnemeier (UU), Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language
03 Jan Martijn van der Werf (TUE), Compositional Design and Verification of Component-Based Information Systems
04 Hado van Hasselt (UU), Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference
05 Bas van der Raadt (VU), Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.
06 Yiwen Wang (TUE), Semantically-Enhanced Recommendations in Cultural Heritage
07 Yujia Cao (UT), Multimodal Information Presentation for High Load Human Computer Interaction
08 Nieske Vergunst (UU), BDI-based Generation of Robust Task-Oriented Dialogues
09 Tim de Jong (OU), Contextualised Mobile Media for Learning
10 Bart Bogaert (UvT), Cloud Content Contention
11 Dhaval Vyas (UT), Designing for Awareness: An Experience-focused HCI Perspective
12 Carmen Bratosin (TUE), Grid Architecture for Distributed Process Mining
13 Xiaoyu Mao (UvT), Airport under Control. Multiagent Scheduling for Airport Ground Handling
14 Milan Lovric (EUR), Behavioral Finance and Agent-Based Artificial Markets
15 Marijn Koolen (UvA), The Meaning of Structure: the Value of Link Evidence for Information Retrieval
16 Maarten Schadd (UM), Selective Search in Games of Different Complexity
17 Jiyin He (UVA), Exploring Topic Structure: Coherence, Diversity and Relatedness
18 Mark Ponsen (UM), Strategic Decision-Making in complex games
19 Ellen Rusman (OU), The Mind's Eye on Personal Profiles
20 Qing Gu (VU), Guiding service-oriented software engineering - A view-based approach
21 Linda Terlouw (TUD), Modularization and Specification of Service-Oriented Systems
22 Junte Zhang (UVA), System Evaluation of Archival Description and Access
23 Wouter Weerkamp (UVA), Finding People and their Utterances in Social Media
24 Herwin van Welbergen (UT), Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior
25 Syed Waqar ul Qounain Jaffry (VU), Analysis and Validation of Models for Trust Dynamics
26 Matthijs Aart Pontier (VU), Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots
27 Aniel Bhulai (VU), Dynamic website optimization through autonomous management of design patterns
28 Rianne Kaptein (UVA), Effective Focused Retrieval by Exploiting Query Context and Document Structure
29 Faisal Kamiran (TUE), Discrimination-aware Classification
30 Egon van den Broek (UT), Affective Signal Processing (ASP): Unraveling the mystery of emotions
31 Ludo Waltman (EUR), Computational and Game-Theoretic Approaches for Modeling Bounded Rationality
32 Nees-Jan van Eck (EUR), Methodological Advances in Bibliometric Mapping of Science
33 Tom van der Weide (UU), Arguing to Motivate Decisions
34 Paolo Turrini (UU), Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations
35 Maaike Harbers (UU), Explaining Agent Behavior in Virtual Training
36 Erik van der Spek (UU), Experiments in serious game design: a cognitive approach
37 Adriana Burlutiu (RUN), Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference
38 Nyree Lemmens (UM), Bee-inspired Distributed Optimization
39 Joost Westra (UU), Organizing Adaptation using Agents in Serious Games
40 Viktor Clerc (VU), Architectural Knowledge Management in Global Software Development
41 Luan Ibraimi (UT), Cryptographically Enforced Distributed Data Access Control
42 Michal Sindlar (UU), Explaining Behavior through Mental State Attribution
43 Henk van der Schuur (UU), Process Improvement through Software Operation Knowledge
44 Boris Reuderink (UT), Robust Brain-Computer Interfaces
45 Herman Stehouwer (UvT), Statistical Language Models for Alternative Sequence Selection
46 Beibei Hu (TUD), Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work
47 Azizi Bin Ab Aziz (VU), Exploring Computational Models for Intelligent Support of Persons with Depression
48 Mark Ter Maat (UT), Response Selection and Turn-taking for a Sensitive Artificial Listening Agent
49 Andreea Niculescu (UT), Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality

2012 01 Terry Kakeeto (UvT), Relationship Marketing for SMEs in Uganda
02 Muhammad Umair (VU), Adaptivity, emotion, and Rationality in Human and Ambient Agent Models
03 Adam Vanya (VU), Supporting Architecture Evolution by Mining Software Repositories
04 Jurriaan Souer (UU), Development of Content Management System-based Web Applications
05 Marijn Plomp (UU), Maturing Interorganisational Information Systems
06 Wolfgang Reinhardt (OU), Awareness Support for Knowledge Workers in Research Networks
07 Rianne van Lambalgen (VU), When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions
08 Gerben de Vries (UVA), Kernel Methods for Vessel Trajectories
09 Ricardo Neisse (UT), Trust and Privacy Management Support for Context-Aware Service Platforms
10 David Smits (TUE), Towards a Generic Distributed Adaptive Hypermedia Environment
11 J.C.B. Rantham Prabhakara (TUE), Process Mining in the Large: Preprocessing, Discovery, and Diagnostics
12 Kees van der Sluijs (TUE), Model Driven Design and Data Integration in Semantic Web Information Systems
13 Suleman Shahid (UvT), Fun and Face: Exploring non-verbal expressions of emotion during playful interactions
14 Evgeny Knutov (TUE), Generic Adaptation Framework for Unifying Adaptive Web-based Systems
15 Natalie van der Wal (VU), Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes.
16 Fiemke Both (VU), Helping people by understanding them - Ambient Agents supporting task execution and depression treatment
17 Amal Elgammal (UvT), Towards a Comprehensive Framework for Business Process Compliance
18 Eltjo Poort (VU), Improving Solution Architecting Practices
19 Helen Schonenberg (TUE), What's Next? Operational Support for Business Process Execution
20 Ali Bahramisharif (RUN), Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing
21 Roberto Cornacchia (TUD), Querying Sparse Matrices for Information Retrieval
22 Thijs Vis (UvT), Intelligence, politie en veiligheidsdienst: verenigbare grootheden?
23 Christian Muehl (UT), Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction
24 Laurens van der Werff (UT), Evaluation of Noisy Transcripts for Spoken Document Retrieval
25 Silja Eckartz (UT), Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application
26 Emile de Maat (UVA), Making Sense of Legal Text
27 Hayrettin Gurkok (UT), Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games
28 Nancy Pascall (UvT), Engendering Technology Empowering Women
29 Almer Tigelaar (UT), Peer-to-Peer Information Retrieval
30 Alina Pommeranz (TUD), Designing Human-Centered Systems for Reflective Decision Making
31 Emily Bagarukayo (RUN), A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure
32 Wietske Visser (TUD), Qualitative multi-criteria preference representation and reasoning
33 Rory Sie (OUN), Coalitions in Cooperation Networks (COCOON)
34 Pavol Jancura (RUN), Evolutionary analysis in PPI networks and applications
35 Evert Haasdijk (VU), Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics
36 Denis Ssebugwawo (RUN), Analysis and Evaluation of Collaborative Modeling Processes
37 Agnes Nakakawa (RUN), A Collaboration Process for Enterprise Architecture Creation
38 Selmar Smit (VU), Parameter Tuning and Scientific Testing in Evolutionary Algorithms
39 Hassan Fatemi (UT), Risk-aware design of value and coordination networks
40 Agus Gunawan (UvT), Information Access for SMEs in Indonesia
41 Sebastian Kelle (OU), Game Design Patterns for Learning
42 Dominique Verpoorten (OU), Reflection Amplifiers in self-regulated Learning
43 Withdrawn
44 Anna Tordai (VU), On Combining Alignment Techniques
45 Benedikt Kratz (UvT), A Model and Language for Business-aware Transactions
46 Simon Carter (UVA), Exploration and Exploitation of Multilingual Data for Statistical Machine Translation
47 Manos Tsagkias (UVA), Mining Social Media: Tracking Content and Predicting Behavior
48 Jorn Bakker (TUE), Handling Abrupt Changes in Evolving Time-series Data
49 Michael Kaisers (UM), Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions
50 Steven van Kervel (TUD), Ontologogy driven Enterprise Information Systems Engineering
51 Jeroen de Jong (TUD), Heuristics in Dynamic Sceduling; a practical framework with a case study in elevator dispatching

---

2013 01 Viorel Milea (EUR), News Analytics for Financial Decision Support
02 Erietta Liarou (CWI), MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing
03 Szymon Klarman (VU), Reasoning with Contexts in Description Logics
04 Chetan Yadati (TUD), Coordinating autonomous planning and scheduling
05 Dulce Pumareja (UT), Groupware Requirements Evolutions Patterns
06 Romulo Goncalves (CWI), The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience
07 Giel van Lankveld (UvT), Quantifying Individual Player Differences
08 Robbert-Jan Merk (VU), Making enemies: cognitive modeling for opponent agents in fighter pilot simulators

09 Fabio Gori (RUN), Metagenomic Data Analysis: Computational Methods and Applications
10 Jeewanie Jayasinghe Arachchige (UvT), A Unified Modeling Framework for Service Design.
11 Evangelos Pournaras (TUD), Multi-level Reconfigurable Self-organization in Overlay Services
12 Marian Razavian (VU), Knowledge-driven Migration to Services
13 Mohammad Safiri (UT), Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly
14 Jafar Tanha (UVA), Ensemble Approaches to Semi-Supervised Learning Learning
15 Daniel Hennes (UM), Multiagent Learning - Dynamic Games and Applications
16 Eric Kok (UU), Exploring the practical benefits of argumentation in multi-agent deliberation
17 Koen Kok (VU), The PowerMatcher: Smart Coordination for the Smart Electricity Grid
18 Jeroen Janssens (UvT), Outlier Selection and One-Class Classification
19 Renze Steenhuizen (TUD), Coordinated Multi-Agent Planning and Scheduling
20 Katja Hofmann (UvA), Fast and Reliable Online Learning to Rank for Information Retrieval
21 Sander Wubben (UvT), Text-to-text generation by monolingual machine translation
22 Tom Claassen (RUN), Causal Discovery and Logic
23 Patricio de Alencar Silva (UvT), Value Activity Monitoring
24 Haitham Bou Ammar (UM), Automated Transfer in Reinforcement Learning
25 Agnieszka Anna Latoszek-Berendsen (UM), Intention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System
26 Alireza Zarghami (UT), Architectural Support for Dynamic Homecare Service Provisioning
27 Mohammad Huq (UT), Inference-based Framework Managing Data Provenance
28 Frans van der Sluis (UT), When Complexity becomes Interesting: An Inquiry into the Information eXperience
29 Iwan de Kok (UT), Listening Heads
30 Joyce Nakatumba (TUE), Resource-Aware Business Process Management: Analysis and Support
31 Dinh Khoa Nguyen (UvT), Blueprint Model and Language for Engineering Cloud Applications
32 Kamakshi Rajagopal (OUN), Networking For Learning; The role of Networking in a Lifelong Learner's Professional Development
33 Qi Gao (TUD), User Modeling and Personalization in the Microblogging Sphere
34 Kien Tjin-Kam-Jet (UT), Distributed Deep Web Search
35 Abdallah El Ali (UvA), Minimal Mobile Human Computer Interaction
36 Than Lam Hoang (TUe), Pattern Mining in Data Streams
37 Dirk Börner (OUN), Ambient Learning Displays
38 Eelco den Heijer (VU), Autonomous Evolutionary Art
39 Joop de Jong (TUD), A Method for Enterprise Ontology based Design of Enterprise Information Systems
40 Pim Nijssen (UM), Monte-Carlo Tree Search for Multi-Player Games
41 Jochem Liem (UVA), Supporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning
42 Léon Planken (TUD), Algorithms for Simple Temporal Reasoning
43 Marc Bron (UVA), Exploration and Contextualization through Interaction and Concepts

---

2014 01 Nicola Barile (UU), Studies in Learning Monotone Models from Data
02 Fiona Tuliyano (RUN), Combining System Dynamics with a Domain Modeling Method
03 Sergio Raul Duarte Torres (UT), Information Retrieval for Children: Search Behavior and Solutions
04 Hanna Jochmann-Mannak (UT), Websites for children: search strategies and interface design - Three studies on children's search performance and evaluation
05 Jurriaan van Reijsen (UU), Knowledge Perspectives on Advancing Dynamic Capability
06 Damian Tamburri (VU), Supporting Networked Software Development
07 Arya Adriansyah (TUE), Aligning Observed and Modeled Behavior
08 Samur Araujo (TUD), Data Integration over Distributed and Heterogeneous Data Endpoints
09 Philip Jackson (UvT), Toward Human-Level Artificial Intelligence: Representation and Computation of Meaning in Natural Language
10 Ivan Salvador Razo Zapata (VU), Service Value Networks
11 Janneke van der Zwaan (TUD), An Empathic Virtual Buddy for Social Support
12 Willem van Willigen (VU), Look Ma, No Hands: Aspects of Autonomous Vehicle Control
13 Arlette van Wissen (VU), Agent-Based Support for Behavior Change: Models and Applications in Health and Safety Domains
14 Yangyang Shi (TUD), Language Models With Meta-information
15 Natalya Mogles (VU), Agent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems: Applications in Safety and Healthcare
16 Krystyna Milian (VU), Supporting trial recruitment and design by automatically interpreting eligibility criteria
17 Kathrin Dentler (VU), Computing healthcare quality indicators automatically: Secondary Use of Patient Data and Semantic Interoperability
18 Mattijs Ghijsen (UVA), Methods and Models for the Design and Study of Dynamic Agent Organizations
19 Vinicius Ramos (TUE), Adaptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support
20 Mena Habib (UT), Named Entity Extraction and Disambiguation for Informal Text: The Missing Link
21 Kassidy Clark (TUD), Negotiation and Monitoring in Open Environments
22 Marieke Peeters (UU), Personalized Educational Games - Developing agent-supported scenario-based training
23 Eleftherios Sidirourgos (UvA/CWI), Space Efficient Indexes for the Big Data Era
24 Davide Ceolin (VU), Trusting Semi-structured Web Data

25 Martijn Lappenschaar (RUN), New network models for the analysis of disease interaction
26 Tim Baarslag (TUD), What to Bid and When to Stop
27 Rui Jorge Almeida (EUR), Conditional Density Models Integrating Fuzzy and Probabilistic Representations of Uncertainty
28 Anna Chmielowiec (VU), Decentralized k-Clique Matching
29 Jaap Kabbedijk (UU), Variability in Multi-Tenant Enterprise Software
30 Peter de Cock (UvT), Anticipating Criminal Behaviour
31 Leo van Moergestel (UU), Agent Technology in Agile Multiparallel Manufacturing and Product Support
32 Naser Ayat (UvA), On Entity Resolution in Probabilistic Data
33 Tesfa Tegegne (RUN), Service Discovery in eHealth
34 Christina Manteli (VU), The Effect of Governance in Global Software Development: Analyzing Transactive Memory Systems.
35 Joost van Ooijen (UU), Cognitive Agents in Virtual Worlds: A Middleware Design Approach
36 Joos Buijs (TUE), Flexible Evolutionary Algorithms for Mining Structured Process Models
37 Maral Dadvar (UT), Experts and Machines United Against Cyberbullying
38 Danny Plass-Oude Bos (UT), Making brain-computer interfaces better: improving usability through post-processing.
39 Jasmina Maric (UvT), Web Communities, Immigration, and Social Capital
40 Walter Omona (RUN), A Framework for Knowledge Management Using ICT in Higher Education
41 Frederic Hogenboom (EUR), Automated Detection of Financial Events in News Text
42 Carsten Eijckhof (CWI/TUD), Contextual Multidimensional Relevance Models
43 Kevin Vlaanderen (UU), Supporting Process Improvement using Method Increments
44 Paulien Meesters (UvT), Intelligent Blauw. Met als ondertitel: Intelligence-gestuurde politiezorg in gebiedsgebonden eenheden.
45 Birgit Schmitz (OUN), Mobile Games for Learning: A Pattern-Based Approach
46 Ke Tao (TUD), Social Web Data Analytics: Relevance, Redundancy, Diversity
47 Shangsong Liang (UVA), Fusion and Diversification in Information Retrieval

2015 01 Niels Netten (UvA), Machine Learning for Relevance of Information in Crisis Response
02 Faiza Bukhsh (UvT), Smart auditing: Innovative Compliance Checking in Customs Controls
03 Twan van Laarhoven (RUN), Machine learning for network data
04 Howard Spoelstra (OUN), Collaborations in Open Learning Environments
05 Christoph Bösch (UT), Cryptographically Enforced Search Pattern Hiding
06 Farideh Heidari (TUD), Business Process Quality Computation - Computing Non-Functional Requirements to Improve Business Processes
07 Maria-Hendrike Peetz (UvA), Time-Aware Online Reputation Analysis
08 Jie Jiang (TUD), Organizational Compliance: An agent-based model for designing and evaluating organizational interactions
09 Randy Klaassen (UT), HCI Perspectives on Behavior Change Support Systems
10 Henry Hermans (OUN), OpenU: design of an integrated system to support lifelong learning
11 Yongming Luo (TUE), Designing algorithms for big graph datasets: A study of computing bisimulation and joins
12 Julie M. Birkholz (VU), Modi Operandi of Social Network Dynamics: The Effect of Context on Scientific Collaboration Networks
13 Giuseppe Procaccianti (VU), Energy-Efficient Software
14 Bart van Straalen (UT), A cognitive approach to modeling bad news conversations
15 Klaas Andries de Graaf (VU), Ontology-based Software Architecture Documentation
16 Changyun Wei (UT), Cognitive Coordination for Cooperative Multi-Robot Teamwork
17 André van Cleeff (UT), Physical and Digital Security Mechanisms: Properties, Combinations and Trade-offs
18 Holger Pirk (CWI), Waste Not, Want Not! - Managing Relational Data in Asymmetric Memories
19 Bernardo Tabuenca (OUN), Ubiquitous Technology for Lifelong Learners
20 Lois Vanhée (UU), Using Culture and Values to Support Flexible Coordination
21 Sibren Fetter (OUN), Using Peer-Support to Expand and Stabilize Online Learning
22 Zhemin Zhu (UT), Co-occurrence Rate Networks
23 Luit Gazendam (VU), Cataloguer Support in Cultural Heritage
24 Richard Berendsen (UVA), Finding People, Papers, and Posts: Vertical Search Algorithms and Evaluation
25 Steven Woudenberg (UU), Bayesian Tools for Early Disease Detection
26 Alexander Hogenboom (EUR), Sentiment Analysis of Text Guided by Semantics and Structure
27 Sándor Héman (CWI), Updating compressed colomn stores
28 Janet Bagorogoza (TiU), Knowledge Management and High Performance; The Uganda Financial Institutions Model for HPO
29 Hendrik Baier (UM), Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains
30 Kiavash Bahreini (OU), Real-time Multimodal Emotion Recognition in E-Learning
31 Yakup Koç (TUD), On the robustness of Power Grids
32 Jerome Gard (UL), Corporate Venture Management in SMEs
33 Frederik Schadd (TUD), Ontology Mapping with Auxiliary Resources
34 Victor de Graaf (UT), Gesocial Recommender Systems
35 Jungxao Xu (TUD), Affective Body Language of Humanoid Robots: Perception and Effects in Human Robot Interaction

2016 01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines

02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow
03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support
04 Laurens Rietveld (VU), Publishing and Consuming Linked Data
05 Evgeny Sherkhonov (UVA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers
06 Michel Wilson (TUD), Robust scheduling in an uncertain environment
07 Jeroen de Man (VU), Measuring and modeling negative emotions for virtual training
08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data
09 Archana Nottamkandath (VU), Trusting Crowdsourced Information on Cultural Artefacts
10 George Karafotias (VUA), Parameter Control for Evolutionary Algorithms
11 Anne Schuth (UVA), Search Engines that Learn from Their Users
12 Max Knobbout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
13 Nana Baah Gyan (VU), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization
15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
16 Guangliang Li (UVA), Socially Intelligent Autonomous Agents that Learn from Human Reward
17 Berend Weel (VU), Towards Embodied Evolution of Robot Organisms
18 Albert Meroño Peñuela (VU), Refining Statistical Data on the Web
19 Julia Efremova (Tu/e), Mining Social Structures from Genealogical Data
20 Daan Odijk (UVA), Context & Semantics in News & Web Search
21 Alejandro Moreno Célleri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground
22 Grace Lewis (VU), Software Architecture Strategies for Cyber-Foraging Systems
23 Fei Cai (UVA), Query Auto Completion in Information Retrieval
24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior
26 Dilhan Thilakarathne (VU), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
27 Wen Li (TUD), Understanding Geo-spatial Information on Social Media
28 Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control
29 Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
30 Ruud Mattheij (UvT), The Eyes Have It
31 Mohammad Khelghati (UT), Deep web content monitoring
32 Eelco Vriezekolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations
33 Peter Bloem (UVA), Single Sample Statistics, exercises in learning from just one example
34 Dennis Schunselaar (TUE), Configurable Process Trees: Elicitation, Analysis, and Enactment
35 Zhaochun Ren (UVA), Monitoring Social Media: Summarization, Classification and Recommendation
36 Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
37 Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry
38 Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design
39 Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
40 Christian Detweiler (TUD), Accounting for Values in Design
41 Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance
42 Spyros Martzoukos (UVA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
43 Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
44 Thibault Sellam (UVA), Automatic Assistants for Database Exploration
45 Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
46 Jorge Gallego Perez (UT), Robots to Make you Happy
47 Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks
48 Tanja Buttler (TUD), Collecting Lessons Learned
49 Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis
50 Yan Wang (UVT), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains

---

2017 01 Jan-Jaap Oerlemans (UL), Investigating Cybercrime
02 Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation
03 Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines
04 Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store
05 Mahdieh Shadi (UVA), Collaboration Behavior
06 Damir Vandic (EUR), Intelligent Information Systems for Web Product Search
07 Roel Bertens (UU), Insight in Information: from Abstract to Anomaly
08 Rob Konijn (VU) , Detecting Interesting Differences:Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery

09 Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text
10 Robby van Delden (UT), (Steering) Interactive Play Behavior
11 Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment
12 Sander Leemans (TUE), Robust Process Mining with Guarantees
13 Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology
14 Shoshannah Tekofsky (UvT), You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior
15 Peter Berck (RUN), Memory-Based Text Correction
16 Aleksandr Chuklin (UVA), Understanding and Modeling Users of Modern Search Engines
17 Daniel Dimov (UL), Crowdsourced Online Dispute Resolution
18 Ridho Reinanda (UVA), Entity Associations for Search
19 Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval
20 Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility
21 Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)
22 Sara Magliacane (VU), Logics for causal inference under uncertainty
23 David Graus (UVA), Entities of Interest — Discovery in Digital Traces
24 Chang Wang (TUD), Use of Affordances for Efficient Robot Learning
25 Veruska Zamborlini (VU), Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search
26 Merel Jung (UT), Socially intelligent robots that understand and respond to human touch
27 Michiel Joosse (UT), Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors
28 John Klein (VU), Architecture Practices for Complex Contexts
29 Adel Alhuraibi (UvT), From IT-BusinessStrategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT"
30 Wilma Latuny (UvT), The Power of Facial Expressions
31 Ben Ruijl (UL), Advances in computational methods for QFT calculations
32 Thaer Samar (RUN), Access to and Retrievability of Content in Web Archives
33 Brigit van Loggem (OU), Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity
34 Maren Scheffel (OU), The Evaluation Framework for Learning Analytics
35 Martine de Vos (VU), Interpreting natural science spreadsheets
36 Yuanhao Guo (UL), Shape Analysis for Phenotype Characterisation from High-throughput Imaging
37 Alejandro Montes Garcia (TUE), WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy
38 Alex Kayal (TUD), Normative Social Applications
39 Sara Ahmadi (RUN), Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR
40 Altaf Hussain Abro (VUA), Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems
41 Adnan Manzoor (VUA), Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle
42 Elena Sokolova (RUN), Causal discovery from mixed and missing data with applications on ADHD datasets
43 Maaike de Boer (RUN), Semantic Mapping in Video Retrieval
44 Garm Lucassen (UU), Understanding User Stories - Computational Linguistics in Agile Requirements Engineering
45 Bas Testerink (UU), Decentralized Runtime Norm Enforcement
46 Jan Schneider (OU), Sensor-based Learning Support
47 Jie Yang (TUD), Crowd Knowledge Creation Acceleration
48 Angel Suarez (OU), Collaborative inquiry-based learning

2018 01 Han van der Aa (VUA), Comparing and Aligning Process Representations
02 Felix Mannhardt (TUE), Multi-perspective Process Mining
03 Steven Bosems (UT), Causal Models For Well-Being: Knowledge Modeling, Model-Driven Development of Context-Aware Applications, and Behavior Prediction
04 Jordan Janeiro (TUD), Flexible Coordination Support for Diagnosis Teams in Data-Centric Engineering Tasks
05 Hugo Huurdeman (UVA), Supporting the Complex Dynamics of the Information Seeking Process
06 Dan Ionita (UT), Model-Driven Information Security Risk Assessment of Socio-Technical Systems
07 Jieting Luo (UU), A formal account of opportunism in multi-agent systems
08 Rick Smetsers (RUN), Advances in Model Learning for Software Systems
09 Xu Xie (TUD), Data Assimilation in Discrete Event Simulations
10 Julienka Mollee (VUA), Moving forward: supporting physical activity behavior change through intelligent technology
11 Mahdi Sargolzaei (UVA), Enabling Framework for Service-oriented Collaborative Networks
12 Xixi Lu (TUE), Using behavioral context in process mining
13 Seyed Amin Tabatabaei (VUA), Computing a Sustainable Future
14 Bart Joosten (UVT), Detecting Social Signals with Spatiotemporal Gabor Filters
15 Naser Davarzani (UM), Biomarker discovery in heart failure
16 Jaebok Kim (UT), Automatic recognition of engagement and emotion in a group of children
17 Jianpeng Zhang (TUE), On Graph Sample Clustering

18 Henriette Nakad (UL), De Notaris en Private Rechtspraak
19 Minh Duc Pham (VUA), Emergent relational schemas for RDF
20 Manxia Liu (RUN), Time and Bayesian Networks
21 Aad Slootmaker (OUN), EMERGO: a generic platform for authoring and playing scenario-based serious games
22 Eric Fernandes de Mello Araujo (VUA), Contagious: Modeling the Spread of Behaviours, Perceptions and Emotions in Social Networks
23 Kim Schouten (EUR), Semantics-driven Aspect-Based Sentiment Analysis
24 Jered Vroon (UT), Responsive Social Positioning Behaviour for Semi-Autonomous Telepresence Robots
25 Riste Gligorov (VUA), Serious Games in Audio-Visual Collections
26 Roelof Anne Jelle de Vries (UT),Theory-Based and Tailor-Made: Motivational Messages for Behavior Change Technology
27 Maikel Leemans (TUE), Hierarchical Process Mining for Scalable Software Analysis
28 Christian Willemse (UT), Social Touch Technologies: How they feel and how they make you feel
29 Yu Gu (UVT), Emotion Recognition from Mandarin Speech
30 Wouter Beek, The "K" in "semantic web" stands for "knowledge": scaling semantics to the web

2019 01 Rob van Eijk (UL),Web privacy measurement in real-time bidding systems. A graph-based approach to RTB system classification
02 Emmanuelle Beauxis Aussalet (CWI, UU), Statistics and Visualizations for Assessing Class Size Uncertainty
03 Eduardo Gonzalez Lopez de Murillas (TUE), Process Mining on Databases: Extracting Event Data from Real Life Data Sources
04 Ridho Rahmadi (RUN), Finding stable causal structures from clinical data
05 Sebastiaan van Zelst (TUE), Process Mining with Streaming Data
06 Chris Dijkshoorn (VU), Nichesourcing for Improving Access to Linked Cultural Heritage Datasets
07 Soude Fazeli (TUD), Recommender Systems in Social Learning Platforms
08 Frits de Nijs (TUD), Resource-constrained Multi-agent Markov Decision Processes
09 Fahimeh Alizadeh Moghaddam (UVA), Self-adaptation for energy efficiency in software systems
10 Qing Chuan Ye (EUR), Multi-objective Optimization Methods for Allocation and Prediction
11 Yue Zhao (TUD), Learning Analytics Technology to Understand Learner Behavioral Engagement in MOOCs
12 Jacqueline Heinerman (VU), Better Together
13 Guanliang Chen (TUD), MOOC Analytics: Learner Modeling and Content Generation
14 Daniel Davis (TUD), Large-Scale Learning Analytics: Modeling Learner Behavior & Improving Learning Outcomes in Massive Open Online Courses
15 Erwin Walraven (TUD), Planning under Uncertainty in Constrained and Partially Observable Environments
16 Guangming Li (TUE), Process Mining based on Object-Centric Behavioral Constraint (OCBC) Models
17 Ali Hurriyetoglu (RUN),Extracting actionable information from microtexts
18 Gerard Wagenaar (UU), Artefacts in Agile Team Communication
19 Vincent Koeman (TUD), Tools for Developing Cognitive Agents
20 Chide Groenouwe (UU), Fostering technically augmented human collective intelligence
21 Cong Liu (TUE), Software Data Analytics: Architectural Model Discovery and Design Pattern Detection
22 Martin van den Berg (VU),Improving IT Decisions with Enterprise Architecture
23 Qin Liu (TUD), Intelligent Control Systems: Learning, Interpreting, Verification
24 Anca Dumitrache (VU), Truth in Disagreement - Crowdsourcing Labeled Data for Natural Language Processing
25 Emiel van Miltenburg (VU), Pragmatic factors in (automatic) image description
26 Prince Singh (UT), An Integration Platform for Synchromodal Transport
27 Alessandra Antonaci (OUN), The Gamification Design Process applied to (Massive) Open Online Courses
28 Esther Kuinderman (UL), Cleared for take-off: Game-based learning to prepare airline pilots for critical situations
29 Daniel Formolo (VU), Using virtual agents for simulation and training of social skills in safety-critical circumstances
30 Vahid Yazdanpanah (UT), Multiagent Industrial Symbiosis Systems
31 Milan Jelisavcic (VU), Alive and Kicking: Baby Steps in Robotics
32 Chiara Sironi (UM), Monte-Carlo Tree Search for Artificial General Intelligence in Games
33 Anil Yaman (TUE), Evolution of Biologically Inspired Learning in Artificial Neural Networks
34 Negar Ahmadi (TUE), EEG Microstate and Functional Brain Network Features for Classification of Epilepsy and PNES
35 Lisa Facey-Shaw (OUN), Gamification with digital badges in learning programming
36 Kevin Ackermans (OUN), Designing Video-Enhanced Rubrics to Master Complex Skills
37 Jian Fang (TUD), Database Acceleration on FPGAs