Scalable Data Management using GPUs with Fast Interconnects

vorgelegt von
MSc ETH CS
Clemens Lutz
ORCID: 0000-0002-6193-4734

an der Fakultät IV — Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
- Dr. rer. nat. -

genehmigte Dissertation

Tag der wissenschaftlichen Aussprache: 10. November 2022

Berlin 2022

To my beloved Irina.

Acknowledgments

I will be forever grateful to all who supported and nurtured my academic growth.

First and foremost, I am deeply indebted to my advisor Volker Markl, who took me under his wing as his doctoral student. He established an environment in the DIMA group in which I was able to thrive, and steadfastly guided me towards success.

In the final phase of my studies, Anastasia Ailamaki, Matthias Böhm, Stefan Manegold, and Tilmann Rabl honored me greatly by forming my doctoral committee. Their counsel on research and life has already been invaluable to inform my future path.

Throughout this time, Sebastian Breß, Tilmann Rabl, and Steffen Zeuch mentored me on high-impact research. They helped me find my direction, and endured my first paper drafts. In short, I could not have prevailed on this journey without them.

A special thanks to my excellent students for their trust in me. Adrian Michalke pulled me aboard his EcoJoin research. Phillip Grote, Alexander Kumaigorodski, and Apostolos Planas chose me as thesis adviser. Apostolos assisted in my SIGMOD reproducibility submission. Together, Alexander and I received the BTW Best Paper award.

My sincere thanks to my fellow colleagues, who injected the very necessary portion of optimism into the everyday grind. I especially thank Philipp Grulich, Martin Kiefer, Bonaventura Del Monte, and Viktor Rosenfeld, with whom I reflected on ideas. Many thanks to Sebastian Breß, Martin Kiefer, and Steffen Zeuch for reviewing this thesis.

I am thankful to the administrative staff for their assistance. Particularly, Lutz Friedel, Claudia Gantzer, Melanie Neumann, and Anna Weymann backed me over many years.

I'd like to acknowledge Anna Kubik, Jonas Pfefferle, and Animesh Trivedi, who encouraged me to take the leap and start my doctoral studies.

From the outset, my endeavor has been aided and abetted by my family. I am grateful to my parents Doris and Werner Lutz, for their love, belief in me, and unwavering support. This extends to my sister Christina Lutz, my grandparents Kreszentia and Friedrich Bischof, Lucia and Peter Schmid, my godparents Claudia Katz and Florinus Bischof, and my father-in-law Alexander Zyablov.

Finally, my wife Irina Lutz has become my life companion and best friend. We have shared moments of happiness and sorrow. She has endured my absence when working long hours, but never fails to brighten my mood with her presence. My wholehearted thanks for always being there for me, caring and loving.

Abstract

Modern *database management systems (DBMSs)* are tasked with analyzing terabytes of data, employing a rich set of relational and machine learning operators. To process data at large scales, research efforts have strived to leverage the high computational throughput and memory bandwidth of specialized co-processors such as *graphics processing units (GPUs)*. However, scaling data management on GPUs is challenging because (1) the on-board memory of GPUs has too little capacity for storing large data volumes, while (2) the interconnect bandwidth is not sufficient for ad hoc transfers from main memory. Thus, data management on GPUs is limited by a *data transfer bottleneck*. In practice, CPUs process large-scale data faster than GPUs, reducing the utility of GPUs for DBMSs.

In this thesis, we investigate how a new class of *fast interconnects* can address the data transfer bottleneck and scale GPU-enabled data management. Fast interconnects link GPU co-processors to a CPU with high bandwidth and cache-coherence. We apply our insights to process stateful and iterative algorithms out-of-core by the examples of a hash join and *k*-means clustering.

We first analyze the hardware properties. Our experiments show that the high interconnect bandwidth enables the GPU to efficiently process large data sets stored in main memory. Furthermore, cache-coherence facilitates new DBMS designs that tightly integrate CPU and GPU via shared data structures and pageable memory allocations. However, irregular accesses from the GPU to main memory are not efficient. Thus, the operator state of, e.g., joins does not scale beyond the GPU memory capacity.

We scale joins to a large state by contributing our new *Triton join* algorithm. Our main insight is that fast interconnects enable GPUs to efficiently spill the join state by partitioning data out-of-core. Thus, our Triton join breaks through the GPU memory capacity limit and increases throughput by up to 2.5× compared to a radix-partitioned join on the CPU.

We scale *k*-means to large data sets by eliminating two key sources of overhead. In existing strategies, execution crosses from the GPU to the CPU on each iteration, which results in the *cross-processing* and *multi-pass* problems. In contrast, our solution requires only a single data pass per iteration and speeds-up throughput by up to 20×.

Overall, GPU-enabled DBMSs are able to overcome the data transfer bottleneck by employing new out-of-core algorithms that take advantage of fast interconnects.

Zusammenfassung

Moderne *Datenbankverwaltungssysteme (DBMS)* werden verwendet um Terabytes von Daten zu analysieren, wobei eine Vielzahl von relationalen und maschinell lernenden Operatoren eingesetzt werden. Um Daten in großen Massen zu verarbeiten, strebten Forschungsversuche an, den hohen Rechendurchsatz und Speicherbandbreite von spezialisierten Coprozessoren wie beispielsweise *Grafikprozessoren (GPUs)* zu nutzen. Jedoch stellt die Skalierung der Datenverwaltung auf GPUs eine Herausforderung dar, weil (1) der integrierte Speicher von GPUs zu wenig Kapazität für die Speicherung großer Datenmengen hat, wohingegen (2) die Bandbreite des Interconnects nicht für eine Ad-hoc-Übertragung aus dem Hauptspeicher ausreicht. Somit ist die Datenverwaltung auf GPUs durch einen *Datentransfer-Engpass* begrenzt. In der Praxis verarbeiten daher *Hauptprozessoren (CPUs)* große Datenmengen schneller als GPUs, was die Nützlichkeit von GPUs für DBMSs verringert.

In dieser Dissertation untersuchen wir, wie eine neue Klasse von *schnellen Interconnects* den Datentransfer-Engpass beheben und die GPU-gestützte Datenverwaltung skalieren kann. Schnelle Interconnects verbinden GPU-Coprozessoren zu einer CPU mit hoher Bandbreite und Cache-Kohärenz. Wir wenden unsere Erkenntnisse an, um zustandsbehaftete und iterative Algorithmen aus dem Coprozessor ausgelagert zu verarbeiten, beispielshalber an einem Hash Join und an dem *k*-Means Clustering-Verfahren.

Wir analysieren zunächst die Hardware-Eigenschaften. Unsere Experimente zeigen, dass die hohe Transferbandbreite der GPU ermöglicht, große, im Hauptspeicher gespeicherte Datensätze effizient zu verarbeiten. Zudem fördert die Cache-Kohärenz neue DBMS-Designs, welche die CPU und die GPU über gemeinsam genutzte Datenstrukturen und auslagerbare Speicherallokationen eng integrieren. Allerdings sind ungleich verteilte Zugriffe von der GPU aus auf den Hauptspeicher nicht effizient. Deshalb lässt sich der Operator-Zustand von z.B. Joins nicht über die Speicherkapazität der GPU hinaus skalieren.

Wir skalieren Joins auf einen großen Zustand, indem wir unseren neuen *Triton Join-Algorithmus* einführen. Unsere wichtigste Erkenntnis ist, dass schnelle Interconnects GPUs befähigen, den Join-Zustand durch ein externes Datenpartitionierungverfahren effizient auszulagern. Somit durchbricht unser Triton Join die durch die GPU-Speicherkapazität gegebene Begrenzung und steigert den Durchsatz um bis zu dem 2,5-fachen im

Vergleich zu einem Radix-partitionierten Join auf der CPU.

Wir skalieren *k*-Means auf große Datensätze, indem wir zwei Hauptlimitationen beseitigen. Bei bestehenden Strategien wechselt die Ausführung bei jeder Iteration zwischen der GPU und der CPU hin und her, was zu den *Cross-Processing*- und *Multi-Pass-Problemen* führt. Im Gegensatz dazu erfordert unsere Lösung nur einen einzigen Datendurchlauf pro Iteration und beschleunigt den Durchsatz um bis zu 20 Mal.

Insgesamt sind GPU-gestützte DBMSs in der Lage, den Datentransfer-Engpass zu überwinden, indem sie neue, Zustands-auslagerbare Algorithmen einsetzen, die die Vorteile schneller Interconnects ausnutzen.

Contents

"There are no bounds to human thought."

— Sergey Pavlovich Korolyov

1 Introduction

Large-scale data management has become a pillar of science and industry, enabling new research fields, services, and business models [185, 242]. Earth monitoring satellites [89] and genome sequencers [405] generate terabytes of data every day. Online services such as Google Search [156] and Uber [143] are backed by petabytes of data. *Database management systems (DBMS)* routinely ingest and manage these large data volumes[12, 38, 111, 164].

1.1 Motivation

In order to continue scaling data management as the progression of Moore's Law slows down [75, 134, 182, 186], co-processors such as GPUs, FPGAs, and ASICs have been gaining adoption in research [47, 203, 257, 356, 415] and industry [90, 211, 238, 244, 355, 379] over the past decade. The entry barrier to co-processors is now low with instant availabilty from all major cloud vendors, including Alibaba Cloud, Amazon EC2, Google Compute Engine, and Microsoft Azure. Despite the growing adoption and availability, in 2019 commercial *GPU-enabled DBMSs* occupied only a tiny 4.5–5.5‰ slice [269, 271, 433] in the $46 billion DBMS market [152]. GPU-enabled DBMSs are found mostly in the form of research prototypes [79, 102, 146, 176, 274, 323], start-ups [66, 82, 140, 184, 227, 313, 348, 392], and peripheral products [210, 380]. Major DBMS vendors, such as Amazon, IBM, Microsoft, Oracle, SAP, and Snowflake, currently do not integrate co-processors into the core of their DBMS products. In contrast, there is wide-spread adoption in the deep learning [120, 289] and high performance computing domains. For instance, 30% of the Top500 supercomputers support co-processors [402].

Motivation 1: Data-intensive Query Processing. To outcompete CPUs in performance benchmarks, GPU experts often assume that their input data are stored in the on-board *GPU memory* [79, 147, 176, 322, 338, 374]. GPU memory provides high-bandwidth access to the data, but has only a limited storage capacity and therefore cannot hold large data volumes [22, 302]. In practice, GPU-enabled DBMSs scale to large data volumes by storing data in *CPU memory* instead of in GPU memory [77, 102, 146, 244, 274, 340, 349, 424]. CPU memory has sufficient capacity to store large data volumes [23, 30, 198, 394], as its capacity is two orders-of-magnitude greater than GPU memory [22, 306]. However, moving data from CPU memory to the GPU reduces query performance because the data are transferred over an interconnect. Consequently, database research points out that a data transfer bottleneck is the main reason behind the comparatively slow adoption of GPU-enabled DBMSs [45, 146, 161, 274, 349, 374, 424].

Motivation 2: Stateful Data Processing. During query processing, DBMSs require additional memory to retain the intermediate state of the query [77, 103, 146]. Queries involving state, e.g., joins, are considered a strong point of GPU-enabled DBMSs, as keeping the state in GPU memory results in high query performance [171, 219, 288, 324, 357, 361, 364, 385, 395, 401, 420]. However, recent investigations reveal that a large state size incurs memory contention [78, 274] and causes commercial GPU-enabled DBMSs to fail query execution [102, 104, 147, 244, 324]. Thus, current GPU-enabled DBMSs are not optimized to handle large state. We consider fragility in scaling the state size an obstacle for building production-ready DBMS products.

Motivation 3: Iterative Algorithms. In addition to relational queries, modern DBMSs [62, 178, 222, 236, 320, 369, 370, 429] and specialized systems [69, 70, 108, 237, 358, 391, 418] target machine learning queries. Machine learning queries differ from relational queries in that they iterate over the same data, i.e., *the working set*, multiple times [71]. Although GPU are able to quickly compute machine learning queries, GPU execution strategies for, e.g., *k*-means, typically assume that the working set fits into GPU memory [31, 40, 209, 233]. In effect, systems scale-out to multiple GPUs to manage large data sets, which increases the cost of processing queries.

Overall, GPU co-processing does not scale to large data volumes. Currently, GPUs are only able to speed-up short queries over small data sets, which have a small scope for improvement. Hence, an opportunity is lost for state-of-the-art DBMSs to achieve faster response times on those long-running, large-scale queries where performance matters.

1.2 Research Challenges

In this thesis, we investigate the scalability limitations of GPU co-processing and analyze how a faster interconnect helps us to overcome them. A new class of *fast interconnects* provide GPUs with high-bandwidth, cache-coherent access to main memory. Recent examples include NVLink 2.0 [297] and 4.0 [81], Infinity Fabric [2], and Compute Express Link 2.0 [110]. Efficiently utilizing a fast interconnect in a GPU-enabled DBMS requires us to reevaluate fundamental design decisions in order to adapt to the new hardware properties.

We address the following research challenges in this work:

Challenge 1: Scalable Data-intensive Query Processing. In Chapter 3, we investigate the principle limitations of GPU interconnects in the context of databases. We introduce fast interconnects, and show by the example of NVLink 2.0 that a fast interconnect improves the basic bandwidth and latency characteristics beyond PCI-e 3.0. Due to these improvements, we conclude that GPUs are now capable of efficiently processing large, out-of-core data sets. However, our measurements show that fast interconnects also lead to new challenges, such as handling operators with a large state efficiently. The content described in Chapter 3 was published at SIGMOD 2020 [261].

Challenge 2: Scalable and Robust Stateful Data Processing. In Chapter 4, we propose the *Triton join*, a new join algorithm that efficiently handles large-scale joins for which the join state exceeds the GPU memory capacity. Although fast interconnects provide faster random-access bandwidth than PCI-e, the join throughput experiences a sharp performance drop when the join state exceeds the GPU memory capacity. Thus, large joins face the challenges of limited scalability and robustness. Previous approaches avoid the transfer bottleneck by preprocessing data on the CPU and thereby reducing the transfer volume. In contrast, we show that fast interconnects enable GPUs to efficiently partition data out-of-core, and are thus able to scale to a large join state using exclusively the GPU. With our Triton join, the principle limitation posed by the GPU memory capacity is solveable with a throughput of more than 50% vs. a join with a small state. The content described in Chapter 4 chapter was published at SIGMOD 2022 [262].

Challenge 3: Scalable Iterative Algorithms. In Chapter 5, we demonstrate that iterative machine learning algorithms are able to process large data sets by examining k-means and proposing a scalable GPU execution strategy. State-of-the-art execution strategies accelerate k-means by extracting the compute-intensive parts and offloading these computations to the GPU. However, as execution is split into a CPU phase and a GPU phase, this execution strategy incurs data transfer overhead and misses optimization opportunities. On each iteration, the CPU and the GPU transfer the model over the interconnect, and separately pass over the data. These overheads exacerbate as the algorithm requires tens of iterations to converge [132]. As a result, the state-of-the-art strategy executes slower than a CPU-only baseline. In contrast, we avoid the model transfer overhead by proposing a new centroid update algorithm optimized for GPUs. In a second optimization step, this enables us to fuse all phases of k-means into a single data pass per iteration. Thus, our new GPU-only execution strategy efficiently scales k-means to large data volumes. The content described in Chapter 5 was published as a DaMoN 2018 short paper [260] and at Datenbanken Spektrum 2018 [259].

1.3 Contributions and Impact

During the course of our research, we have made the following contributions.

Conference Papers. We have published the main contributions of this thesis at top-tier national and international conference venues:

- Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl: *Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects*, in the ACM International Conference on Management of Data, June 14–19, 2020, Portland, OR, USA.

- Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl: *Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects*, in the ACM International Conference on Management of Data, June 12–17, 2022, Philadelphia, PA, USA.

- Clemens Lutz, Sebastian Breß, Tilmann Rabl, Steffen Zeuch, and Volker Markl: *Efficient and Scalable* k*-means on GPUs*, in Datenbank-Spektrum 18(3): 157–169 (2018).

- Clemens Lutz, Sebastian Breß, Tilmann Rabl, Steffen Zeuch, and Volker Markl: *Efficient* k*-means on GPUs*, in the 14th ACM Int. Workshop on Data Management on New Hardware (DaMoN ’18), colocated with SIGMOD/PODS, Houston, TX, USA, June 11th, 2018.

Follow-up Research. Our main contributions have inspired follow-up research on fast interconnects which are not part of this thesis:

- Alexander Kumaigorodski, Clemens Lutz, Volker Markl: *Fast CSV Loading Using GPUs and RDMA for In-Memory Data Processing*, in Database Systems for Business, Technology and Web, April 19th–June 21st, 2021, Dresden, Germany.
- Josef Schmeißer, Clemens Lutz, Volker Markl: *Indexing Data to Scale Processing on GPUs with Fast Interconnects*, ongoing research.

Research Awards. Our works have been honored by the research community with two awards and two badges. *Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects* was awarded **Best Paper at SIGMOD 2020** and *Fast CSV Loading Using GPUs and RDMA for In-Memory Data Processing* was awarded **Best Paper at BTW 2021**. These two works also received the **Reproducibility Badges** of their publication venues.

Teaching. Starting from the winter semester 20/21, our paper on *Fast CSV Loading Using GPUs and RDMA for In-Memory Data Processing* is featured in the course *Reproducibility Engineering* at the University of Passau and the Regensburg University of Applied Sciences. Students reproduce our research results as part of their graded coursework.

Open Source Contributions. We have released all projects that are part of this thesis under an open source license:

- `https://github.com/TU-Berlin-DIMA/CL-kmeans`. This repository contains our OpenCL implementation of k-means [259, 260]. The project consists of two single-processor execution strategies that are both able to run on either a CPU or a GPU, and one hybrid CPU-GPU execution strategy. CL-kmeans is licensed under the Mozilla Public Licence 2.0.
- `https://github.com/TU-Berlin-DIMA/fast-interconnects`. This repository contains all sub-projects related to fast interconnects [261, 262]. These include our Triton join, our hybrid hash table, no-partitioning joins and TPC-H Query 6 for the CPU and the GPU, and interconnect microbenchmarks. Alongside the code, we have documented our implementation techniques in detail. This project is licensed under the Apache License 2.0.

Additional Contributions. The author of this thesis has published works on stream processing in collaboration with other DIMA members. These papers are not related to fast interconnects and not part of this thesis:

- Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, Volker Markl: *Analyzing Efficient Stream Processing on Modern Hardware*, in The Proceedings of the VLDB Endowment, Vol. 12, No. 5, Los Angeles, CA, USA, August 26th–30th, 2019.

- Adrian Michalke, Philipp Marian Grulich, Clemens Lutz, Steffen Zeuch, Volker Markl: *An Energy-Efficient Stream Join for the Internet of Things*, in the 17th ACM Int. Workshop on Data Management on New Hardware (DaMoN '21), held online with SIGMOD/PODS, June 21st, 2021.

1.4 Thesis Outline

The remainder of this thesis is structured as follows.

Chapter 2. We first provide background information on fast interconnects and GPU-enabled DBMSs, which form the common ground for the subsequent chapters.

Chapter 3. Next, we investigate a fast interconnect. We present our insights on how DBMS implementers should interact with the hardware, as well as the opportunities and challenges facing DBMS designers. We additionally describe chapter-specific background and related work in this and in the next two chapters.

Chapter 4. In this chapter, we contribute our Triton join algorithm. We detail the random access characteristics of a fast interconnect. Based on these observations, we design two new out-of-core radix partitioning algorithms. Then, we build our Triton join on top of these partitioning algorithms.

Chapter 5. After that, we present our efficient k-means execution strategy. We show our new centroid update algorithm for the GPU. On this basis, we optimize k-means for out-of-core execution with a single data pass.

Chapter 6. In the final chapter, we draw conclusions from our findings and state our perspective on how this thesis will contribute to future work.

2 Background

Computer architecture and data management systems are fascinating topics with an astounding depth. In this chapter, we summarize the scientific foundation of our insights on these topics. In doing so, we focus on the aspects which we believe the data management community are less familiar with. We begin with a brief overview of how GPUs are programmed in Section 2.1, before examining GPU architectures in Section 2.2. In Section 2.3, we cover interconnect architecture with an emphasis on fast interconnects. We end our background by recapping GPU-enabled database management systems in Section 2.4.

2.1 GPU Programming Frameworks

Programmers write code for GPUs using a GPU computing framework. GPU computing frameworks consist of a programming language which compiles down to GPU assembly language and a runtime that executes the code on the GPU.

Overview. Most GPU programming frameworks are vendor-specific and target only GPUs. Nvidia provides the *CUDA* platform [304], AMD specifies the *ROCm* platform [29] and *HIP* language [24], while Intel supplies the *oneAPI* platform [201]. In contrast, *OpenCL* specifies a vendor-neutral language and runtime standard [224]. In addition to GPUs, vendors have implemented OpenCL for *central processing units (CPUs)*, *field-programmable gate arrays (FPGAs)*, and other processor types.

GPU Kernels. CUDA is based on C++ [304], whereas OpenCL C (i.e., OpenCL's language) extends the C language [224]. Both frameworks refer to functions that run on

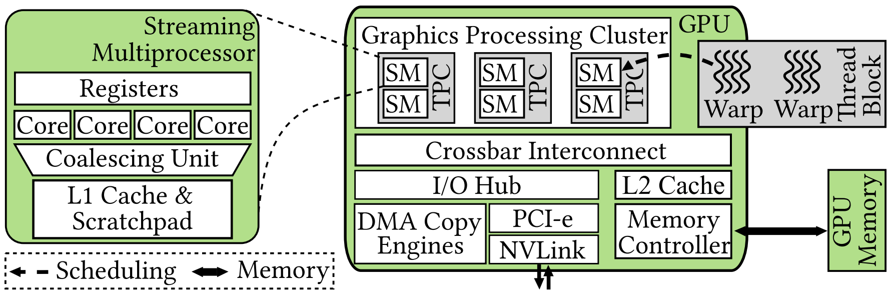


Figure 2.1: Hardware architecture of an Nvidia Volta GPU.

the GPU as *kernels*. When a program calls a kernel, the runtime *launches* the kernel on the GPU. Inside of kernels, programmers have access to built-in functions that expose GPU hardware functionality, e.g., thread synchronization and communication functions. We highlight the relevant functionality in Section 2.2.4.

Compilation. In a first step, CUDA and OpenCL are compiled to *PTX* [304]. PTX describes a low-level, device-independent intermediate representation of Nvidia's compiler [307]. Programmers can directly embed PTX instructions into CUDA programs as inline assembly to access advanced GPU functionality not exposed by the programming language [305]. In a final step, the compiler transforms PTX into a binary native to the specific GPU architecture [304].

Summary. GPU programming frameworks provide a high-level interface to the GPU hardware by means of programming languages that are familiar to systems programmers. Although we program the software artifacts accompanying this thesis using the CUDA and OpenCL frameworks, the concepts extend to other frameworks as well.

2.2 GPU Architecture

The hardware architectures of GPUs are designed for high-throughput processing. In Section 2.2.1, we give an overview of GPU architectures and their features. After that, we describe GPU thread execution in Section 2.2.2 and why control flow can limit performance in Section 2.2.3. In Section 2.2.4, we outline the hardware functionality responsible for thread synchronization and communication.

2.2.1 Overview

Modern GPU architectures all follow similar high-level design patterns. Recent examples are the Nvidia Volta [99, 297], Ampere [302], and Hopper [308], AMD CDNA [18] and CDNA 2 [22], and Intel Ponte Vecchio [68] architectures. We focus on the Nvidia Volta

GPU architecture, as most of our experiments in Chapters 3–5 are performed on the Nvidia Tesla V100 GPU model [297]. In cases where significant differences exist, we contrast Volta to the other GPU architectures.

In Figure 2.1, we show the architecture of an Nvidia Volta GPU. Volta GPUs consist of up to 84 *streaming multiprocessors (SMs)* [297]. SMs are paired in *texture processing cluster (TPC)*, and seven TPCs are grouped into a *graphics processing cluster (GPC)* for a total of six GPCs [297]. The GPU executes threads in parallel on all SMs. Each SM schedules threads in hardware [254] to hide memory latencies of up to 2 µs [141]. To efficiently schedule threads, the SM stores thread contexts in 65 thousand registers [297]. SMs physically execute 32 threads together as a *warp* [99]. The threads of a warp typically share a program counter [13, 254], but in Volta and newer GPUs each thread has its own program counter [297] (see Section 2.2.3). The AMD CDNA and RDNA architectures reduce the native warp size to 32 threads [17, 18, 22], i.e., the same size as Nvidia GPUs, instead of the 64 threads of previous AMD architectures [13, 15]. GPU programming languages abstract up to 32 warps (i.e., 1024 threads) as a *thread block* [21, 304]. Each thread block has access to *scratchpad memory*, which is a fast software-managed cache inside the SM [99]. The GPU caches memory accesses in its *L1 and L2 caches* [99]. A *crossbar interconnect* connects the SMs to GPU memory and an *I/O hub* [296]. The I/O hub has PCI-e and NVLink controllers that link the GPU to the host system and peer GPUs [296]. The I/O hub also contains multiple *DMA copy engines*[1] [296], to which GPU programs can offload asynchronous data transfers [303].

2.2.2 Thread Execution

In order to sustain a high computational throughput, GPUs are designed to execute thousands of threads in parallel [246, 254]. Specializing the hardware architecture for parallel processing instead of single thread performance increases efficiency, as speculative out-of-order cores consume five times more energy and die area compared to simple cores [67, 371].

Instructions take several cycles to execute (e.g., 4 cycles for an addition [34]), and memory accesses can take hundreds of cycles [34, 208]. These instruction and memory latencies can cause thread execution to stall until the dependencies of the next instruction are ready [254, 408]. To hide these latencies, the GPU schedules a different thread and continues execution [254, 408]. When a GPU kernel begins execution, a global thread block scheduler distributes thread blocks among the SMs [293]. The thread

[1]CUDA shows three DMA engines for NVLink models, and seven DMA engines for PCI-e models.

block scheduler uses a *most room policy*, which takes available hardware resources into account, e.g., scratchpad memory, registers, and threads [155]. Each SM then statically assigns warps to one of four sub-cores [208]. Each sub-core has a warp scheduler that fetches warp instructions from the L0 instruction cache and assigns each instruction to a dispatch unit [99]. The scheduling policy of the warp schedulers in Nvidia GPUs is not publicly known [204, 221], but a *greedy-then-oldest policy* [354] is commonly assumed by simulators [3, 187]. In contrast, AMD RDNA [17] and Intel X^e MAX [199] documentation specifies the greedy-then-oldest policy. Greedy-then-oldest continuously schedules a single warp until it stalls, and schedules the oldest (by instantiation time) ready warp next [354]. Volta GPUs have four types of dispatch units: a branch unit, a math unit, a memory and I/O unit, and a tensor unit [99]. Each dispatch unit dispatches the instruction to multiple execution units specialized for the data type [99], similar to a SIMD core [254]. In contrast to the other dispatch units, the memory and I/O unit is shared by the four sub-cores [99]. The memory and I/O unit *coalesces* (i.e., groups) the memory accesses of 8 threads into a single *memory transcation*, which it dispatches to the L1 data cache [221]. Coalescing memory accesses improves memory transfer efficiency [118, 291] and reduces the memory address translation request rate [347]. Even though the scheduler and dispatch units are single-issue [99], a sub-core can execute an integer and a floating-point instruction simultaneously [297, 311].

2.2.3 Control Flow and Warp Divergence

GPUs adhere to a *single instruction, multiple threads (SIMT)* model [254]. The SIMT model provides programmers with a multithreading abstraction for executing on SIMD hardware. SIMT differs from *single instruction, multiple data (SIMD)* in that SIMT threads, in contrast to SIMD lanes, are able to perform control flow (i.e., nested branch and loop statements) [304]. In this section, we first describe the current hardware implementations, and then give advice on effective GPU programming and summarize our findings.

Control Flow Reconvergence Schemes

In the SIMT model, branches are physically executed by masking execution units [133]. A thread-active mask marks each thread of a warp as active or inactive, and the warp executes all taken branches. A warp *diverges* when its threads take different branches and *reconverges* afterwards.

Hardware vendors have implemented control flow using two different reconvergence schemes: stack-based reconvergence [106, 145, 414] and thread frontiers [125, 126]. We briefly describe these schemes in the following.

Stack-Based Reconvergence. Control flow is statically handled by the compiler for AMD GPUs [19, 25] and Nvidia GPUs up until Pascal [297]. The compiler emits instructions to build a stack of branch targets, which include a thread-active mask for each branch. By popping branch targets from the stack, the GPU executes all possible paths through the control flow graph, and reconverges the warp at a predetermined reconvergence point. In this scheme, the warp always executes in lock-step, as the warp shares a program counter.

Thread Frontiers. Starting with the Volta and Sandy Bridge architectures, Nvidia [307] and Intel GPUs [125] dynamically reconverge warps at runtime (Nvidia calls this scheme *independent thread scheduling* [297]). The compiler emits branch and reconvergence instructions at all potential divergence and reconvergence points, but does not enumerate paths. Instead, the GPU evaluates which paths are taken during execution, and executes all taken paths. In contrast to stack-based reconvergence, each thread has its own program counter. Divergent branches cause the program counters to contain different addresses, and threads reconverge if their program counters point to the same instruction. Thus, a subset of the warp may opportunistically reconverge and execute a common instruction block.

Effective Control Flow Programming

In both reconvergence schemes, GPU programmers should limit divergent control flow to warps instead of branching in individual threads. Warp divergence leads to inefficient execution in both reconvergence schemes, as the sub-cores execute divergent paths sequentially [297]. During divergent paths, the execution units are partially idle. In contrast, divergent paths taken by different warps do not cause warp divergence [55, 56]. As a result, per-warp control flow usually performs better than per-thread control flow.

The thread frontiers scheme enables advanced data structures that are not possible to implement with stack-based reconvergence. In the stack-based reconvergence scheme, if a thread blocks, e.g., due to spinning on a lock, then the warp does not make progress [99]. Furthermore, warp barriers are subject to complex rules regarding undefined behavior [307]. This results in deadlocks when implementing, e.g., spin-locks [297, 310]. In contrast, thread frontiers (as implemented by Volta GPUs) guarantee forward progress for each thread and the soundness of thread barriers at sub-warp granularity [99]. In his Bachelor's thesis, Phillip Grote showed that these guarantees are sufficient to implement lock-based data structures on GPUs [162]. In some cases, lock-based synchronization strategies reduce the implementation complexity of thread-safe algorithms while retaining a high performance [135].

Summary

Overall, algorithms should control their flow at the granularity of warps instead of threads to avoid warp divergence. If per-thread control flow is unavoidable, modern GPU architectures handle the resuling warp divergence robustly.

2.2.4 Thread Synchronization and Communication

GPUs were initially designed to execute in a bulk-synchronous-parallel programming model [254]. However, recent parallel algorithms employ complex parallel programming patterns such as parallel aggregations [219, 357], the producer-consumer pattern [275, 323], thread-safe linear memory allocators [146, 395], and lock-free data structures [8, 232].

These techniques rely on GPU architectures that execute threads using the shared memory programming model [258] and support inter-thread synchronization and communication [295]. The execution of memory operations adheres to the semantics defined by a formal *memory consistency model* [258, 307]. Furthermore, the hardware exposes primitive operations (i.e,. *primitives*) that synchronize and communicate between threads [304, 307]. We categorize the primitives by their functionality into memory fences, thread barriers, atomic memory operations, and warp-level primitives. In the following, we describe the memory consistency model as well as the communication and synchronization primitives.

Memory Consistency Model. Correctly programming GPUs is challenging due to their weak memory consistency model [9, 390]. Lustig et al. specify a formal memory consistency model [258] based on the natural language specification provided by Nvidia for Volta and newer GPUs [307]. We briefly summarize the model.

Generally, memory consistency models are classified as either *strong* or *weak* [284]. A strong model retains the order of memory operations performed by the same thread, whereas a weak model relaxes this constraint to allow more reordering optimizations [284]. Nvidia defines a weak memory consistency model for PTX [258].

As depicted in Figure 2.2, CUDA and PTX bound inter-thread synchronization and communication to one of three *scopes*: thread block, GPU, and system [304, 307]. Scopes reduce the number of threads which are able to interact with each other's memory operations [307]. In particular, read-modify-write operators (e.g., atomic compare-and-swap) are atomic only in their defined scope [307]. In contrast, all loads and stores are

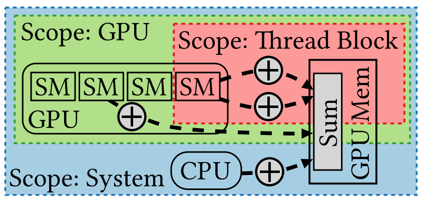


Figure 2.2: Memory model supports scoping of atomic memory operations such as `atomicAdd`.

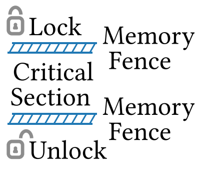


Figure 2.3: The memory fence ensures that the critical section is executed after acquiring and before releasing the lock.

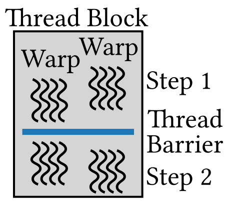


Figure 2.4: A thread barrier synchronizes threads before permitting them to start the next computation step.

atomic by way of the "no thin air" axiom — a loaded value must have been written by a store instead of speculatively resulting from an execution [258, 307]. However, CUDA exposes *weak* and *volatile* loads and stores, which are not thread-safe in regard to atomic read-modify-write operators [307].

Memory Fences. *Memory fences* establish an order between regular memory operations, as reordering operations across the fence is forbidden [304].

For example in Figure 2.3, when implementing a spinlock, the lock must be acquired before entering the critical section and released after exiting the critical section. In this case, imposing an order on memory operations is necessary to guarantee the correctness of the lock. However, the critical section (or parts thereof) could be reordered, e.g., to occur before the lock is acquired. Thus, a memory fence with an appropriate scope is required to separate the (un-)locking critical section.

Thread Barriers. A *thread barrier* synchronizes a group of threads, and is typically placed at the end of a parallel computation step (see Figure 2.4). Explicit thread barriers are defined for the warp and the thread block granularities [304]. A GPU-wide barrier is implicitly set at the kernel completion boundary [304]. Programmers are also able to explicitly synchronize SMs on Volta GPUs [297]. Thread barriers guarantee an ordering of memory operations for the threads participating in the barrier, i.e., a memory fence [304].

Atomic Memory Operations. The shared memory programming model allows

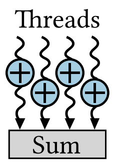


Figure 2.5: With atomic memory operations such as `atomicAdd`, multiple threads can safely access the same memory location.

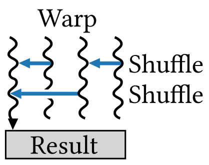


Figure 2.6: Warp-level primitives, e.g., `__shfl_sync`, enable a warp to shuffle values between threads inside of GPU registers.

threads to mutate shared data using atomic memory operations (*atomics*) [304]. As shown in Figure 2.5, an atomic memory operation modifies a primitive data type (e.g., integers) by reading its current value from memory, performing a modification, and writing back the new value to memory (i.e,. *read-modify-write*) without interference from other threads [304]. The targeted data value may be stored either in GPU memory or in scratchpad memory [307]. Although the first GPU architectures implement atomics in software, Pascal and newer GPUs efficiently execute atomics in hardware [207, 296].

Read-modify-write refers to a set of arithmetic and bitwise functions [304]. The most important atomics for this work are `atomicAdd`, `atomicCAS`, and `atomicExch`. `atomicAdd` atomically adds a value to a sum and returns the previous sum. We use `atomicAdd` to allocate array slots to threads, and to aggregate values computed by different threads. `atomicExch` atomically swaps two values. We use `atomicExch` to prepend elements to a linked list in closed addressing hash tables. `atomicCAS` (i.e., compare-and-swap) atomically swaps two values if an equality predicate is true. For example, we use `atomicCAS` to insert elements into a thread-safe hash table (see Section 2.4.2).

Atomic memory operations are supported only for particular data types, predominantly 32-bit and 64-bit unsigned integers [304]. In contrast to recent CPUs [188, 193], Maxwell and newer GPUs natively support atomic addition of floating point numbers [296], but do not support 128-bit `atomicCAS` and signed integer additions [304]. These limitations are relevant for data management, as SQL integers are signed [266] and swapping 64-bit pointers with 64-bit `atomicCAS` leads to the ABA problem [119]. The ABA problem occurs because a given pointer A references a value but does not define the value: when A is swapped for another pointer B and then back to A, the value referenced by A might have changed.

Warp-level Primitives. The threads of a warp are able to communicate within registers via *warp-level primitives*, illustrated in Figure 2.6. GPUs have primitives for data shuffling, matching, and reduction, as well as predicate voting [304]. Warp-level primitives are able to replace atomic memory operations if communication takes place at the warp granularity instead of at the thread block granularity [295].

For this thesis, the relevant functions are `__any_sync`, `__ballot_sync`, and `__shfl_sync`. Of these, `__any_sync` and `__ballot_sync` are warp vote functions, which evaluate a predicate per thread. The former returns a non-zero value if any predicate is true, whereas the latter returns a bitset containing all predicates. The predicate bitset is useful for computing prefix sums and electing a leader thread [253]. With `__shfl_sync`, threads retrieve a value provided by a different thread.

Summary. To efficiently manage thousands of threads, programmers should limit the scope of communication and synchronization operations. Hardware executes narrowly-scoped primitives with a higher performance than widely-scoped primitives. Designing algorithms to take advantage of these hardware features often speeds-up execution by 1.2–5.3× [147, 321, 322, 357].

2.3 Interconnect Architecture

Processors communicate with each other via interconnects. Despite their central role in the system, few works in the DBMS literature study interconnects in depth. Thus, we provide background information on interconnect technologies.

In Section 2.3.1, we first review how hardware and software interact to transfer data over the current PCI-e interconnects. Fast interconnects offer new, performance-enhancing features, which leads us to define fast interconnects as a distinct class of hardware in Section 2.3.2. Based on our definition, we provide an overview of the recent and future fast interconnect technologies offered and planned by different hardware vendors in Section 2.3.3. We then provide a detailed description of the interconnect architecture: data transmission (Section 2.3.4), address translation (Section 2.3.5), and cache-coherence (Section 2.3.6). We summarize in Section 2.3.8 by showing the overall architecture of NVLink 2.0, which is the fast interconnect we focus on in this thesis.

2.3.1 Data Transfer

State-of-the-art systems connect GPUs with a PCI-e 3.0 [7] or 4.0 [326] interconnect. *Peripheral component interconnect express (PCI-e)* implements transfer primitives in hardware on which programming frameworks build software APIs to transfer data.

Transfer Primitives. PCI-e provides two data transfer primitives: memory-mapped I/O (*MMIO*) and direct memory access (*DMA*). MMIO maps GPU memory into the

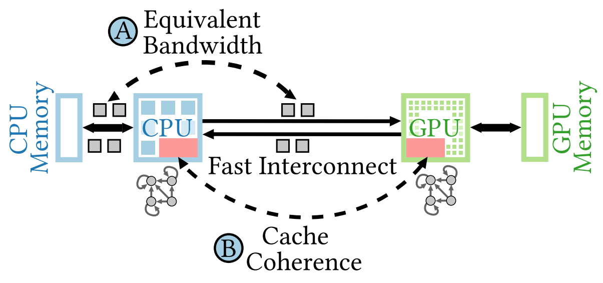


Figure 2.7: The distinguishing properties of fast interconnects.

CPU's address space. The CPU then initiates a PCI-e transfer by accessing the mapped GPU memory with normal load and store instructions. In contrast, DMA allows the GPU to directly access CPU memory. The key difference to MMIO is that DMA only allows access to a predetermined range of *pinned memory*. Memory is "pinned" by locking the physical location of pages, which prevents the OS from moving them. DMA operations can thus be offloaded to data copy engines. These are dedicated hardware units that facilitate both bidirectional transfers and overlapping transfers with computation.

Software APIs. CUDA [304] exposes these two primitives through three abstractions. `cudaMemcpyAsync` copies pageable (i.e., non-pinned) memory through MMIO, but copies pinned memory using DMA copy engines. In contrast, *Unified Virtual Addressing* exposes "zero-copy" pinned memory to the GPU via DMA. Finally, *Unified Memory* transparently moves CPU pages to GPU memory. The page migration is triggered by a memory access to a page not present in GPU memory. The operating system receives a page fault, and moves the requested page from CPU memory to GPU memory [432]. To avoid the page fault's latency, pages can be explicitly prefetched using `cudaMemPrefetchAsync`. Although Unified Memory is built on the aforementioned transfer primitives, CUDA hides the type of memory used internally. We derive transfer methods based on these software APIs in Section 3.3, and evaluate their performance in Section 3.6.2.

In summary, PCI-e data transfers are orchestrated in software and thus processors do not interact directly in hardware.

2.3.2 Fast Interconnects: A Definition

PCI-e interconnects lack hardware features which are important to efficiently manage data on GPUs. We provide a definition of *fast interconnects* based on a set of properties, and reason why these properties are necessary for data management.

Definition. We define a "fast interconnect" to have two distinguishing properties which we illustrate in Figure 2.7: Ⓐ the aggregate bidirectional interconnect bandwidth approximately matches the per-socket CPU memory bandwidth and Ⓑ the interconnect is cache-coherent. In our definition, cache-coherence means that the hardware natively supports a system-wide address space, data-dependent accesses to pageable memory, and system-wide atomic memory operations.

Rationale. We justify why our definition requires high bandwidth and the four cache-coherence sub-properties. One, sufficiently high interconnect bandwidth is necessary to level the playing field between the GPU and the CPU. Without high bandwidth, the GPU cannot efficiently access CPU memory (and vice-versa). Two, cache-coherence (excluding the subsumed properties) is necessary because CPUs transparently cache data. Without cache-coherence, programmers must manage caches manually to avoid accessing stale data. Three, without a system-wide address space, programmers must manually translate pointers in order to move them from one processor's address space to the next. Four, without pageable memory access, memory accessed from a different processor must be pinned beforehand. Five, without atomic memory operations, processors cannot share and mutate data at a fine granularity. Overall, these properties work together to make memory accesses and memory management faster and more convenient.

Practical Limitations. Processors are free to take advantage of only a subset of the interconnect's features. For example, IBM POWER9 CPUs do not achieve the peak bandwidth of NVLink 2.0 [191], and the L1 caches of current GPUs do not implement cache-coherence in hardware [99, 279]. Consequently, the programmer must deal with these shortcomings, e.g., by managing the cache-coherence of the GPU's L1 cache in software. However, the interconnect must support all features to avoid restricting processors to the intersection of their feature sets.

Conclusion. In our fast interconnect definition, we specify properties that complement each other and form a basis on which DBMSs can tightly integrate co-processors. Thus, a fast interconnect provides DBMSs with the means to resolve the data transfer bottleneck from a hardware perspective.

2.3.3 Fast Interconnect Technologies

The class of fast interconnects includes technologies from multiple hardware vendors. At present, these include Nvidia NVLink [130, 297, 302, 308], AMD Infinity Fabric [18, 22], and Intel Compute Express Link (CXL) [109, 110] for GPUs. Fast interconnects are also available for FPGAs with the IBM Open Coherent Accelerator Processor Interface (OpenCAPI) [314, 315], Arm Cache-Coherent Interconnect for Accelerators (CCIX) [30, 92, 330, 417], and CXL [148, 417]. Other fast interconnects such as Gen-Z and Intel UPI

Table 2.1: Fast Interconnect Technologies.

Interconnect	Processors	Vendors	Release	Address translation	Cache-coherence	Atomics	CPU-GPU	GPU-GPU
PCI-e 3.0	any	multiple	2013 [294]	optional	no	optional [16]	16 GB/s	16 GB/s tree
PCI-e 4.0	any	multiple	2020 [18, 302]	optional	no	optional	32 GB/s	32 GB/s tree
PCI-e 5.0	any	multiple	2022 [308]	optional	no	optional	64 GB/s	64 GB/s tree
NVLink 2.0	CPU, GPU	Nvidia [297], IBM [191]	2017 [297]	yes	yes	yes	75 GB/s	50 GB/s mesh, 150 GB/s switched
NVLink 3.0	GPU	Nvidia [302]	2020 [302]	yes [309]	yes	yes	no	100 GB/s mesh, 300 GB/s switched [404]
NVLink 4.0 & C2C	CPU, GPU	Nvidia [130, 308]	2023 [81]	unknown	yes	unknown	450 GB/s [81]	150 GB/s mesh, 450 GB/s switched
Infinity Fabric 3	CPU, GPU	AMD [22]	2021 [22]	yes [93]	MI250X [26]	unknown	100 GB/s	100–150 GB/s mesh
CXL 1.1	CPU, GPU, FPGA	Intel [61, 148, 197], AMD [160], Arm [330], Xilinx [417]	2019 [195]	CXL.cache	CXL.cache	CXL.cache	64 GB/s	support unclear
OpenCAPI 3.0	CPU, FPGA	IBM [396], Alpha Data [10]	2018 [83]	yes	checkout mode	yes	52 GB/s [10]	no [396]
CCIX 1.1	CPU, FPGA	Arm [331], Xilinx [417]	2020 [150]	yes	yes	yes	64 GB/s	64 GB/s mesh

will be superseded by CXL [154, 196].

Comparison. In Table 2.1, we compare fast interconnects with the current PCI-e versions. We list the year by the availability of the first GPU or FPGA product, and note the interconnect bandwidth per transfer direction. We focus on CPU-GPU connections, but some interconnects can also connect multiple GPUs in mesh or switched topologies.

Fast interconnects provide high bandwidth to CPU memory in the case of NVLink and Infinity Fabric. NVLink 4.0 and NVLink-C2C are related by their capabilities, although they serve different purposes: NVLink 4.0 connects multiple GPUs [308], whereas NVLink-C2C integrates CPUs with GPUs [130]. CXL offers the same bandwidth as PCI-e 5.0, as these technologies share the physical layer. In contrast to the PCI-e standards, fast interconnects feature address translation services and cache-coherence in hardware. Although address translation and atomics can optionally be added to PCI-e [7, 325, 326, 327], these features are not supported by GPUs in practice [24, 304]. Although CXL and OpenCAPI compliance is possible without cache-coherence, these interconnects classify GPUs (and FPGAs) as cache-coherent accelerators [109, 396].

Summary. Although fast interconnects are not yet mainstream, GPU and FPGA vendors are actively developing new fast interconnects. Standardization efforts such as CCIX and CXL are ongoing and could result in more wide-spread adoption in the hardware industry.

2.3.4 Data Transmission

The high performance of fast interconnects results from data transmission optimizations. We categorize the optimizations by the performance metric they primarily affect, i.e., bandwidth or latency.

Bandwidth Optimizations

Hardware designers improve the bandwidth by increasing the rate at which data are physically transmitted over the wires and by reducing interconnect protocol overheads. These approaches complement each other, and we show examples for both cases.

Signaling Rate. Fast interconnects operate at high clock frequencies to increase the *signaling rate*. For example, NVLink 2.0 operates at 25 GHz to transmit data at 25 Gbit/s per wire pair [297] and NVLink 3.0 transmits data at 50 Gbit/s per wire pair [100], which are 1.6× and 3.2× faster than PCI-e 4.0 [326], respectively. However, the high signaling rates of NVLink rely on substrate materials with tight electrical tolerances [105].

Interconnect Width. Widening the interconnect by adding more wires increases the data rate. PCI-e specifies a *lane* as two differential wire pairs with one pair per transfer direction, i.e., four wires in total [326]. In contrast, NVLink 1.0 specifies a bidirectional *link* as 32 wires [141], which is reduced to sixteen wires in NVLink 3.0 [302, 308] and to only eight wires in NVLink 4.0 [308]. Bit sequences are split into *flits* (flow control digits), each of which is striped over the wires of a lane (or link) [272]. Flits are multiplexed over all lanes (or links) [110, 316, 326], e.g., 16 PCI-e lanes are typically bundled to connect GPUs. Moreover, NVLink generations continuously increase the number of links from four in NVLink 1.0 [141] to six in NVLink 2.0 [297], twelve in NVLink 3.0 [100], and eighteen in NVLink 4.0 [308].

Encoding Scheme. Sending more bits per clock cycle also results in a higher data rate. Interconnects transmit digital signals by encoding them into analog signals using *encoding schemes* such as non-return-to-zero (NRZ) and four-level pulse amplitude modulation (PAM-4). NRZ encodes a zero as a low-voltage signal, and a one as a high-voltage signal [334]. In contrast, PAM-4 encodes two bits into four different voltage levels, which doubles the data rate at the cost of a worse signal-to-noise ratio [194]. PCI-e 6.0 departs from NRZ and instead uses PAM-4 to avoid increasing the interconnect clock frequency [376].

Block Code. Encoding schemes work in conjunction with a *block code* to facilitate clock recovery [334]. Modern interconnects are self-clocked (i.e., *serial interconnects*), which means that the receiver infers the transmitter's clock signal via *clock recovery*, instead of relying on an explicit clock signal sent over a dedicated wire [272]. A block code, e.g., the 8b/10b code of PCI-e 1.0 and 2.0 [376, 411], breaks long runs of only zeroes or only ones by transcoding 8-bit sequences into 10-bit sequences of limited runlengths [192, 411]. As a 8b/10b code incurs 20% overhead, recent interconnects specify low-overhead 64b/66b and 128b/130b block codes instead [109, 376, 396].

Packet Header Size. Data are sent over interconnects in the form of packets [92, 109, 141, 272, 396]. In PCI-e, packets consist of a payload of up to 512 bytes and a 20–28 byte header [287]. In contrast, fast interconnects reduce the overhead incurred by the header. For example, NVLink packets consist of a 16 byte header for 1–256 payload bytes [141].

Latency Optimizations

Higher bandwidth directly reduces latency, as serializing each digital packet into an analog signal takes less time [113]. Despite this, neglecting latency in the interconnect design results in a high latency [280], such as for PCI-e [287]. Thus, hardware vendors aim to the reduce latency of fast interconnects by design.

Flits. Fast interconnects transmit data as fixed-length flits instead of variable-length frames [109, 141, 396]. A *flit* (i.e., flow control digit) defines the minimum transmission unit as a fixed-length bit sequence [113]. For example, flits in OpenCAPI are 64 bytes long and aligned at a 16 byte offset [396]. In contrast, PCI-e 1.0–5.0 transmits data as *frames*, which denote their start and end with special *framing tokens* [326, 376]. Frames require complex receiver logic to find frames in the transmitted bit stream, and to rotate the bytes for the decoder [396]. Thus, flits lower the interconnect latency in comparison to frames [376, 396]. Shortening the flit size further reduces latency, as transmitting a small flit takes less time [377]. For example, NVLink 1.0 [141] has 16× smaller flits than PCI-e 6.0 [376] (16 bytes vs. 256 bytes).

Forward Error Correction. Recent interconnects correct errors by retransmitting faulty packets, along with all subsequent packets following a fault packet [98, 109, 141, 316, 376]. Thus, retransmission adds latency because it requires an additional roundtrip over the interconnect and blocks the data stream from proceeding [114]. Instead, PCI-e 6.0 performs *forward error correction (FEC)* [376]. The receiver corrects most, but not all, errors by applying an error correction code (ECC), which is appended to each flit [376]. Forward error correction is necessary for interconnects with a high bit error rate due to, e.g., a PAM-4 encoding scheme [100, 376], which may become more prevalent in future interconnects [377].

Summary

Fast interconnects optimize bandwidth and latency to achieve high performance. This performance is the result of rapidly integrating technological advancements on the physical and protocol layers to improve the data rate and reduce protocol overheads. In contrast to PCI-e, fast interconnects specialize for performance over interoperability, e.g., by sacrificing backward compatibility, adding design constraints, and tightening manufacturing tolerances.

2.3.5 Address Translation

Hardware and the operating system work together to *translate addresses* from a program's *virtual address space* to the *physical address space* provided by the hardware [180]. GPUs support address translation [13, 254], which enables a unified virtual address space consisting of CPU memory and GPU memory [296, 304]. However, GPUs cannot directly access pageable CPU memory, and instead require pinned memory (see Section 2.3.1).

We briefly provide an overview of address translation, and how fast interconnects

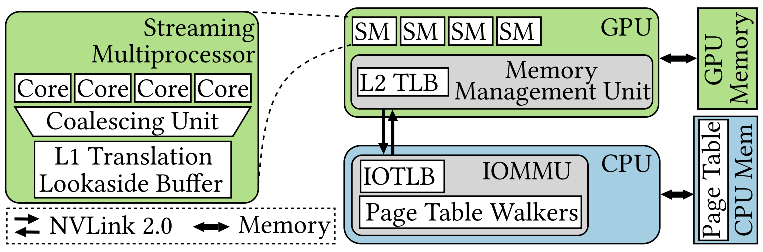


Figure 2.8: Address translation architecture of a system with a fast interconnect.

support it. After that, we describe the challenges of fully integrating GPUs into the system-wide address space, and the solutions employed by recent hardware.

Overview

As an example, we show the translation architecture of an IBM AC922 system [290] with an IBM POWER9 CPU and an Nvidia V100 GPU in Figure 2.8.

A *memory management unit (MMU)* is responsible for address translation [60]. The MMU looks up translations by *walking* a per-process *page table*, which is stored in memory [60]. *Page table walkers* accelerate address translations by performing multiple walks in parallel and operating asynchronously to the processor cores [60]. Translations are cached in *translation lookaside buffers (TLBs)* [60]. Modern processors have multiple TLB levels (e.g., an *L1 TLB* and an *L2 TLB*), and each CPU core (or GPU SM) has a private L1 TLB [60, 217]. CPUs may additionally contain an *input/output memory management unit (IOMMU)* and an *input/output translation lookaside buffer (IOTLB)* [58]. The IOMMU and IOTLB translate DMA accesses issued by PCI-e devices (and other interconnects) to CPU memory addresses [58].

Address Translation Support in Fast Interconnects

The PCI-e *address translation services (ATS)* specification [325, 326, 327] serves as a basis for CCIX and CXL [91, 109]. The ATS specification defines a protocol for co-processors to request translations from an IOMMU. The optional *page request services* extend the protocol to handle page faults.

For cache-coherent co-processors, CXL lifts the PCI-e ATS from an optional extension to a mandatory feature in its CXL.cache sub-protocol [109]. In contrast, CCIX always requires address translation, either via PCI-e ATS or a co-processor MMU which natively walks the host page table [91].

Instead of relying on PCI-e ATS, OpenCAPI and NVLink 2.0 specify their own address translation protocols. In OpenCAPI 3.0, the IOMMU translates all addresses [314]. OpenCAPI 4.0 additionally permits the co-processor to contain a TLB [315]. In contrast, Nvidia Volta GPUs, which are the only co-processors capable of NVLink 2.0, have a TLB that requests translations from the IOMMU [99].

However, no details have been released to the public on how Infinity Fabric and NVLink 4.0 translate addresses.

MMU and TLB Design Challenges

Efficiently implementing address translation in hardware presents many challenges. We discuss five challenges which are relevant for fast interconnects:

Parallel Translation Requests. GPUs are able to issue thousands of memory requests in parallel, which can incur hundreds of TLB lookups [241] and thousands of concurrent page table walks [347]. The standard TLB designs employed by CPUs cannot handle this high TLB request rate, which leads to a bottleneck [337].

High TLB Miss Rate. In contrast to CPU applications, GPU programs exhibit poor temporal locality and therefore experience a high TLB miss rate [205, 347, 381].

High TLB Miss Latency. Resolving a GPU TLB miss in GPU memory incurs a high latency of several hundred cycles [217, 239, 273]. For integrated GPUs (i.e., accelerated processing units), the GPU miss latency can be an order-of-magnitude higher than for the CPU due to the IOTLB [406].

Efficient Page Fault Support. Accessing pageable memory (instead of pinned memory) incurs *page faults* when the accessed pages are not mapped in the GPU's page table [296]. Nvidia Pascal and newer GPUs support page faults, but these are handled in software [296]. According to the specification, PCI-e is able to process page faults in hardware [153, 400, 406] if the hardware supports the optional ATS and page request interface extensions [325, 326, 327]. However, recent GPUs do not support pageable memory access via PCI-e [24, 304]. Thus, the GPU relies on the CPU to handle its page faults, which results in poor performance [151, 432].

IOTLB Management Overhead. The operating system sets up a dedicated page table for the IOMMU, which is managed separately from the system page table accessed by the CPUs [58]. Handling GPU page faults requires the GPU to interrupt the CPU, which resolves the page fault in software [406]. Mapping and unmapping pages causes the CPU to invalidate stale TLB entries by requesting an IOTLB *shootdown*, which must be relayed to the GPU TLB [6, 406]. Therefore, managing the IOTLB incurs overhead [6].

MMU and TLB Design Solutions

Recent GPUs and IOMMUs address these challenges as follows:

Post-Coalescer MMU. GPUs coalesce memory accesses to reduce the number of memory accesses (see Section 2.2). By placing the TLB behind the coalescing unit, the number of translation requests are reduced as well [337, 347]. This design allows GPU MMUs to cope with a high number of parallel translation requests.

Page Table Compression. Memory allocators often map adjacent virtual pages to adjacent physical pages [335, 336]. GPU MMUs exploit this behavior by opportunistically fetching a cacheline of page table entries (i.e., 128 bytes [208, 221] ≡ sixteen entries [299]) with one memory access and then compressing these page table entries into a single TLB entry if they are adjacent [121, 122, 335, 336]. Without compression, sixteen pages occupy 960 bits (120 bytes) in the TLB [121, 122]. The TLB compresses sixteen contiguous pages as a memory address and the number of pages to only 75 bits, and adjacent pages as a base address with 16 offsets to 411 bits [121, 122]. Thus, the TLB achieves compression ratios of up to 12.8×, thereby extending the TLB range and reducing the TLB miss rate.

Large Page Sizes. In addition to compressing page table entries, GPUs support large page sizes [299]. Large pages with, e.g., a size of 2 MiB lower the number of pages (and thus TLB entries) needed to cover a given memory range, and lessen the TLB miss rate [42, 347, 406].

Page Walk Cache. The TLB miss latency is affected by the number of memory accesses required to perform the page table walk [51]. GPU page tables are organized as radix trees up to five levels deep [299]. To translate an address, a page table walk might access up to five different memory locations, i.e., one per tree level [51]. To avoid traversing the full tree, modern CPUs feature a *page walk cache*, which cache the upper levels [51]. With a page walk cache, ideally only one memory access (i.e., to the leaf node) is sufficient to walk the page table. IOMMUs contain a page walk cache [27, 37, 190, 202], and research has evaluated the benefit for GPUs [347]. In contrast, it is unclear if commercial GPUs contain a page walk cache.

IOMMU Page Table Walkers. Recent IOMMUs include multiple hardware page table walkers to efficiently resolve page faults [37, 64, 65, 190, 191]. Handling the page faults in hardware enables the GPU to access pageable memory [99]. Furthermore, parallelizing the page table walks increases translation throughput of the IOMMU, which avoids stalling GPU threads [337, 347, 381, 382].

Single Page Table. Modern IOMMUs are capable of directly accessing the standard page table used by the CPUs [27, 37, 191]. Instead of maintaining a dedicated page table for the IOMMU, the operating system manages a single, system-wide page table. Thus, new mappings added to the page table are immediately available to the GPU [191].

Summary

In conclusion, fast interconnects upgrade address translation from a software feature to an integral part of the hardware. By performing translations using an IOMMU and caching them in the GPU TLB hierarchy, GPUs become capable of accessing memory system-wide. In return, CPUs are also able to directly access GPU memory. In effect, address translation places GPUs and CPUs on an equal level regarding memory management.

2.3.6 Cache-Coherence

CPUs come with cache-coherence out-of-the-box. In contrast, GPUs do not maintain coherence among their caches.

We motivate why cache-coherence is important for systems with CPUs and GPUs. After that, we describe the challenges of designing protocols to maintain coherence, and how fast interconnects currently overcome them.

Overview

Modern multi-processor systems have a hierarchy of caches, in which the first level is private and the last level is shared [181]. Cache-coherence hides the complexity of this cache subsystem from programmers. Without cache-coherence, processors might read stale data from their own cache or from memory if that memory location was updated in the cache of another processor [181]. Cache-coherence ensures that incoherent accesses to a single memory location do not occur [181]. At any given time, there can exist either a single writer, or multiple readers [284]. Thus, cache-coherence enforces a sequential store order per location. However, cache-coherence does not specify how accesses to different memory locations relate — this is defined by the memory consistency model [181].

In contrast to CPUs [181], the caches of current GPUs are not coherent [17, 307]. Graphics workloads infrequently synchronize and share data, and adding cache-coherence to GPUs would increase design complexity [284]. Instead, programmers must manually bypass the L1 cache to keep data consistent between threads [384]. However, new workloads such as data management and tighter integration of the CPU and the GPU challenge these assumptions.

Cache-Coherence Design Challenges

Coherent communication between heterogeneous processors with a fast interconnect requires a different coherence protocol than among homogeneous CPUs. We note five challenges that hardware designers face:

High Interconnect Latency. The latency of PCI-e is considered too high for fine-grained cache-coherence [151]. Although fast interconnects reduce latency, in Section 3.2 we show that it remains higher than the latency of CPU interconnects.

GPU-Induced Latency. Including a GPU in the system-wide cache-coherence domain can introduce additional latency for the CPU, as the CPU must issue coherence requests to the GPU (e.g., cacheline invalidation) [116]. This can occur even when the CPU accesses its local CPU memory, if the cacheline was previously accessed by the GPU. Thus, coherence protocols should avoid penalizing the CPU with additional latency.

High Degree of Parallelism. When executing a program on a GPU, thousands of memory requests are in flight at any given time [241]. Ensuring cache-coherence for these memory requests with the standard coherence protocols used by CPUs would cause bandwidth and storage overhead to issue and track coherence requests [384].

Decouple CPU and GPU Coherence Protocols. Due to the limitations of standard coherence protocols, new protocols seek to decouple CPUs and GPUs. One approach is to exclude the non-coherent GPU caches from the system-wide cache-coherence, but make GPU memory coherently cacheable by the CPU [5, 116]. Alternatively, the GPU can implement a different, GPU-optimized protocol than the CPU, which are combined via a coherence interface [11, 312]. In effect, vendors are able to implement their own coherence protocols and optimize for different processor types [5, 312].

Faulty Coherence Protocol Implementations. Verifying the correctness of coherence protocols is difficult [366]. An incorrect or malicious implementation could starve the CPU of memory resources [312]. To ensure system stability, the CPU should be protected from a misbehaving co-processor [312].

Cache-Coherence Design Solutions

Fast interconnects currently implement cache-coherence by either a selective caching protocol or a variant of the MESI and MOESI protocols. In the following we outline these two protocols.

Selective Caching. NVLink 2.0 relies on a two-state *invalid/valid (IV)* protocol with selective caching [191]. In the IV protocol, cachelines are either in a valid or an invalid state [284]. This means that a processor can only own a cacheline exclusively, as no shared state exists.

Selective caching on the GPU reduces protocol overhead by filtering coherence requests using, e.g., a cuckoo filter [5]. The filter probabilistically tracks which cachelines of GPU memory are owned by the CPU [5]. While a cacheline is captured by the filter, the GPU must fetch it from the CPU's cache instead of GPU memory [5]. The filter is periodically pruned when its fill state exceeds a high water mark [5]. However, as the GPU L1 caches are software-managed, the CPU does not send invalidation requests to the GPU [191]. Thus, the protocol employs asymmetry in order to reduce complexity for the GPU and avoid protocol overhead for the CPU [116].

M(O)ESI. More elaborate MESI-based protocols are used by CCIX [91], CXL [109, 110], OpenCAPI 4.0 [315] and NVLink-C2C [130].

In contrast to IV, MESI has four states: $\underline{\text{M}}$odified, $\underline{\text{E}}$xlusive, $\underline{\text{S}}$hared, and $\underline{\text{I}}$nvalid [284]. MESI has the advantage that multiple processors are able to retain the same cacheline in a read-only state [284]. On top of MESI, CXL adds an error state to protect the CPUs against a misbehaving co-processor [109].

In contrast, CCIX and NVLink-C2C use the Arm AMBA CHI protocol [92, 130], which extends MESI with an $\underline{\text{O}}$wned state, a partial owned state, and a partial exclusive state [36]. In contrast to MOESI, the ABMA CHI protocol allows processors to implement only a part of the state machine and includes additional optimizations [36]. Due to the Owned state, a cache can directly send a dirty cacheline to another cache without writing back the cacheline to memory [284]. The two partial states enable writing to a cacheline without first reading its contents [36]. *Snoop filters* [282] reduce the coherence requests sent across the interconnect in the CXL and AMBA CHI protocols [36, 109].

OpenCAPI 4.0 employs a MESI-based protocol with an Exlusive$_{\text{invalid}}$ state [315], which serves the same purpose as the partial states of AMBA CHI. We note that OpenCAPI 3.0 does not implement a cache-coherence protocol, and instead routes all coherent memory requests issued by the co-processor through the CPU [314].

Challenges Addressed. We find that the high interconnect latency is addressed by selective caching (NVLink 2.0) and snoop filtering (CXL, AMBA CHI). GPU-induced latency is entirely avoided by asymmetric coherence protocols (NVLink 2.0, CXL). The caches of GPUs are natively not coherent, and thus circumvent the challenges inherent to coherence under massive parallelism at the cost of increased programming complexity (all current GPUs). This also decouples the CPU coherence protocol from the GPU. Protocols either guard the CPU against faulty protocol implementations by simplifying the protocol (NVLink 2.0, AMBA CHI) or introducing explicit error states (CXL).

Summary

Overall, the coherence protocols of fast interconnects are rapidly evolving. Heterogeneous processors require different protocol features than CPUs. The current generation of fast interconnects, i.e., NVLink 2.0, have limitations that may reduce performance when accessing shared data. We expect that the upcoming coherence protocols, AMBA CHI in particular, will integrate CPUs and GPUs more tightly.

2.3.7 Atomic Memory Operations

Memory operations that enforce system-wide atomicity are a special case of the GPU atomic memory operations discussed in Section 2.2.4. In current GPU programming languages, atomic memory operations are not system-wide atomic by default. Instead, the programmer must explicitly select a global scope for the atomic operation, e.g., `atomicAdd_system` [24, 304]. Interconnects implement system-wide atomic memory operations in two ways: cache-coherence protocols and atomic transactions.

Cache-Coherence Protocols. Cache-coherence enables fast interconnects to use the same method as CPUs [117]. Coherence protocols such as MESI [319] define a *Modified* state in which a processor (specifically a CPU core, memory controller, or I/O controller) retains exclusive, mutable access to a cacheline. By locking the cacheline in this modified state until the operation is complete, the core enforces atomicity [284]. Thus, to execute atomics on the same cacheline, processors move the cacheline back-and-forth [117].

Atomic Transactions. In contrast, an atomic transaction ships a modification to a memory location [91, 326, 327]. In the system, each memory location is associated with an interconnect controller. By extending the interconnect controller to execute atomic memory operations, processors are able to send atomics to the respective interconnect controller instead of running the instruction themselves. Thus, multiple atomic memory operations can be in flight to the same memory location simultaneously, which eliminates the interconnect round-trip latency of acquiring a cacheline [138, 326]. Furthermore, a processor does not have to be part of the cache-coherence domain to issue atomic memory operations over an interconnect [7]. Active memory operations take this concept one step further by pushing atomics into the memory controller [138, 188], which is supported by, e.g., IBM POWER9 CPUs [35]. However, cache-coherence can serve as a fallback for atomic operations not supported by the receiving processor [191].

Overall, system-wide atomic memory operations are the basis for fine-grained synchronization in DBMSs. In practice, all currently available fast interconnects support both coherence-based and transaction-based atomics [35, 91, 109, 191].

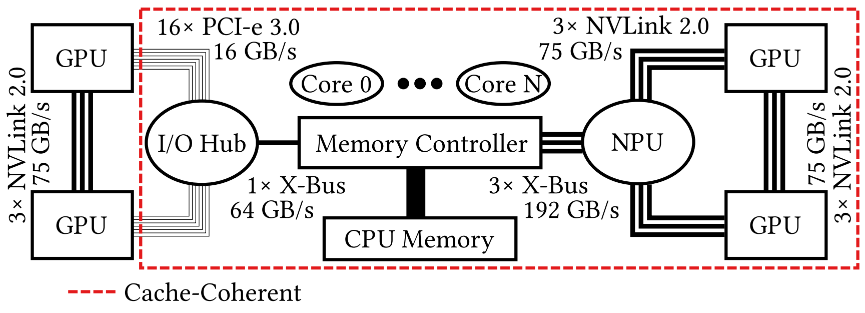


Figure 2.9: Architecture and cache-coherence domains of GPU interconnects, with their electrical bandwidths annotated.

2.3.8 Summary: NVLink 2.0

Fast interconnects depart from PCI-e in that they integrate GPUs into the system via address translation, cache-coherence, and system-wide atomic memory operations. The Nvidia Volta GPUs and IBM POWER9 CPUs that we use in this dissertation support NVLink 2.0 [84, 190, 191, 297]. Therefore, we summarize our discussion of fast interconnects with NVLink 2.0 as an example.

In Figure 2.9, we contrast the architectures of PCI-e 3.0 and NVLink 2.0.

Physical Layer. NVLink 2.0 connects up to one CPU and six GPUs in a point-to-point mesh topology, which has the advantage of higher aggregate bandwidth compared to a tree. Connections consists of multiple full-duplex links that communicate at 25 GB/s per direction. A device has up to six links. Of these, up to three links can be bundled for a total of 75 GB/s. Thus, two GPUs can saturate CPU memory bandwidth, but adding a third reduces the per-GPU bandwidth by ⅓. Like PCI-e, NVLink transmits packets. However, packet headers incur less overhead for small packets, with a 16 byte header for up to 256 bytes of payload.

Transfer Primitives. Data transfers from CPU memory can use MMIO and DMA copy engines. However, in contrast to PCI-e, NVLink gives the GPU direct access to pageable CPU memory. GPU load, store, and atomic operations are translated into CPU interconnect commands (i.e., X-bus on POWER9) by the *NVLink Processing Unit (NPU)*. The NPU is connected by three X-Bus links, each capable of 64 GB/s.

Address Translation. The GPU is integrated into a system-wide address space. If a TLB miss occurs on a GPU access to CPU memory, the NVLink Processing Unit acts as an IOMMU and provides the address translation by walking the CPU's page table. Thus, in contrast to Unified Virtual Addressing and Unified Memory, address translations do not require OS intervention.

Cache-Coherence. Memory accesses are cache-coherent on 128-byte cache-line boundaries. The CPU can thus cache GPU memory in its cache hierarchy, and the GPU can cache CPU memory in its L1 caches. Cache-coherence guarantees that writes performed by one processor are visible by any other processor. The observable order of memory operations depends on the memory consistency model. Intel CPUs guarantee that aligned reads and writes are atomic, and that writes are (nearly) sequentially consistent [193, vol.3A, §8.2]. In contrast, IBM CPUs and Nvidia GPUs have weaker memory consistency models [258].

Summary. Overall, NVLink 2.0 presents an opportunity to explore the new design space offered by fast interconnects for DBMSs. We discuss these opportunities and challenges in Section 3.2. Furthermore, we investigate the translation architecture of NVLink 2.0, its TLB miss latency and random-access bandwidth in detail in Section 4.2.4.

2.4 GPU-enabled Database Management Systems

In this section, we give an overview of GPU-enabled DBMSs (Section 2.4.1) and the DBMS operators that we focus on in this thesis (Section 2.4.2).

2.4.1 DBMS Design

Throughout this dissertation, we assume a generalized model of how DBMSs integrate GPUs, which we present in Figure 2.10. Our model extracts the relevant design features of state-of-the-art GPU-enabled DBMSs that manage data out-of-core [78, 102, 146, 244, 274, 349]. *Out-of-core data* reside outside of the GPU, and the GPU accesses these data over an interconnect. Our work focuses on data stored in CPU memory.

We refer the interested reader to the recent survey conducted by Rosenfeld et al. [356] for an in-depth discussion of GPU-enabled DBMS designs.

GPU Use Cases. In principle, various tasks performed by a DBMS can be offloaded to a GPU, e.g., query execution [158], query optimization [175], transaction processing [72, 172], stream processing [231], and data loading [235]. However, OLAP and machine learning queries are the most established use-cases for GPUs in DBMSs [350, 356].

OLAP and ML. *Online analytical processing (OLAP)* and machine learning queries are formulated ad hoc by the user to explore a data set [71, 94]. They are complex, i.e., consisting of scans, joins, and aggregations in the case of OLAP, and additionally linear algebra operators and iterative control flow in the case of machine learning. Although these workloads are read-heavy, the queries work on the entire data set. As a human is

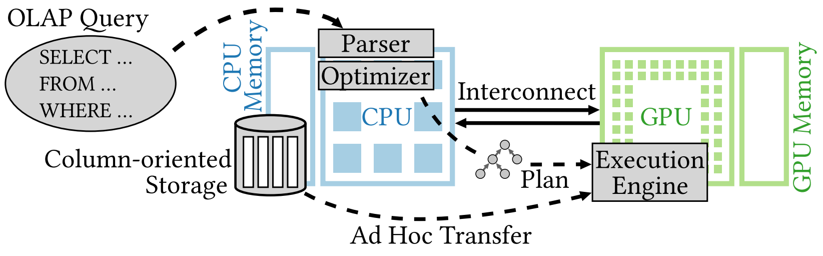


Figure 2.10: Design of a GPU-enabled DBMS.

in the loop, response times should be short. Thus, the goal of GPU-enabled DBMSs is to achieve the high data rates necessary to query large data volumes in a timely fashion.

DBMS Design. From a design perspective, GPU-enabled DBMSs are column-oriented, in-memory DBMSs [77, 170, 176, 424]. This DBMS design stores data in CPU memory and is optimized for analytical query processing [73, 220]. Storing large data sets in CPU memory is possible because modern servers have terabytes of memory capacity (i.e., *DRAM — dynamic random access memory*) [23, 200, 393], and recent non-volatile memory technologies such as Intel Optane are capable of increasing the capacity by an order-of-magnitude [33, 375]. Column-oriented in-memory DBMSs increase the data access performance relative to row-oriented data layouts [77, 170].

Query Execution. Within the DBMS, GPUs extend the query execution engine [78, 102, 146, 244, 274, 349]. The DBMS gains a runtime with operators specialized for GPU execution. During execution, the DBMS transfers the data from CPU memory to the GPU ad hoc and executes the query [78, 102, 146, 244, 274, 349]. The other DBMS components required to execute a query, e.g., SQL parsing and query planning and optimization, are retained on the CPU [77, 103, 176].

Single Data Pass. Different execution paradigms exist to construct query execution engines. Some engines generate code just-in-time to emit operator pipelines [79, 102, 146, 322], others vectorize [274] or tile [323, 374] the execution, and operator-at-a-time execution is yet another variant [77, 176, 244]. The processing paradigm is immaterial to our thesis, but we must consider that only operator pipelines and vectorized/tiled engines are capable of processing queries in a single data pass [146, 374], which avoids transferring data items over the interconnect multiple times.

Summary. Overall, modern GPU-enabled DBMSs achieve a high query execution performance. Advances in the query execution paradigms continue to increase execution efficiency. Furthermore, the computational performance of GPUs is also rapidly increasing with each GPU generation [397, 398]. Thus, in order to sustain their query performance, GPU DBMSs require fast access to data.

2.4.2 DBMS Operators

We present the state-of-the-art knowledge for executing DBMS operators on GPUs. We cover the no-partitioning hash join and partitioned hash join operators, as well as k-means and its two-phase execution pattern.

No-Partitioning Hash Join

In Chapter 3, we investigate the no-partitioning hash join algorithm as proposed by Blanas et al. [63]. The no-partitioning hash join algorithm is a parallel version of the canonical hash join [48]. We focus on this algorithm because it is simple and well-understood. Loading base relations from CPU memory requires high bandwidth, scaling the hash table beyond GPU memory requires low latency, and sharing the hash table between multiple processors requires cache-coherence. Thus, the no-partitioning hash join is a useful instrument to investigate fast GPU interconnects.

The anatomy of a no-partitioning hash join consists of two phases, the build and the probe phase. The build phase takes as input the smaller of the two join relations, which we denote as the inner relation R. In the build phase, we populate the hash table with all tuples in R. After the build phase is complete, we run the probe phase. The probe phase reads the second, larger input relation as input. We name this relation the outer relation S. For each tuple in S, we probe the hash table to find matching tuples from R. When executing the hash join in parallel on a system with p cores, its time complexity observes $O({}^{1}/_{p}(|R| + |S|))$.

Partitioned Hash Join

We extend the parallel radix-partitioned hash join algorithm as introduced by Kim et al. [226] in Chapter 4. The core idea of partitioned join algorithms is to increase data locality, such that we are able to store the hash table in the processor cache [378]. The low access latency of the cache improves the performance of random accesses to the hash table. This technique also applies to other hash-based relational operators, such as group-based aggregations [124, 378, 423] and duplicate elimination [124]. In contrast to other partitioning methods, radix partitioning has the advantage that it helps to reduce TLB misses [267].

Radix-partitioned hash joins have two constraints. First, the hash tables must fit into the cache. As the cache has a constant size, larger data volumes require a higher *fanout* (i.e., number of partitions) to keep the size of each partition constant. Second, a high fanout incurs frequent TLB misses if the fanout is higher than the number of available

TLB entries. TLB misses are expensive, because resolving a miss involves between 1–6 memory accesses [51].

For large data sets, there exists a fundamental tension between these two constraints. On the one hand, a high fanout is necessary to efficiently look up hash table entries. On the other hand, a high fanout increases the cost of partitioning. In the following, we describe how database literature addresses this trade-off by optimizing high-fanout partitioning on CPUs and GPUs.

On CPUs, we reduce TLB misses through *software write-combining (SWWC)* [364]. SWWC reduces TLB misses by intermediately buffering tuples in the processor cache. Tuples are then written to their final positions in batches. Thus, a batch size of N reduces the amount of TLB misses by a factor of N [367]. Flushing buffers can be optimized with *non-temporal stores*, that avoid polluting the cache [410]. Finally, storing partition offsets in a *micro-row layout* reduces cache misses and requires less cache space [46, 368].

Optimization techniques for GPUs differ from those for CPUs. Scattered writes can be coalesced by partially sorting tuples in scratchpad memory [363]. A thread block works together to sort tuples and flush them to memory. In contrast to SWWC on CPUs, all tuples are flushed at the same time. If the batch size is larger than the fanout, it follows that each memory transaction must write out multiple tuples per partition. However, although writes introduce coalescing opportunities, misalignment can still prevent coalescing, thereby reducing efficiency.

On recent GPUs that support efficient atomic additions [207], partitioning can be improved by replacing prefix scan with a linear allocator for a single data pass within the scratchpad [361, 395]. A linear allocator tracks free array slots using an atomically incrementing counter. We refer to this approach as the *linear allocator software write-combining (Linear)* partitioning algorithm.

k-Means

k-Means [256, 263] represents an iterative algorithm for cluster analysis in machine learning applications, which we analyze in Chapter 5. It partitions N observations into k clusters such that each observation belongs to the cluster with the nearest mean. The input to k-means is a set of *points* in an $\mathbb{R}^d$ space spanned by d *features*, and a parameter k that specifies the number of clusters. k-Means produces a *centroid* per cluster and a *label* per point as output. A centroid defines the mean of the points forming a cluster, while a label identifies the cluster to which a point belongs.

The actual processing of k-means is performed in two phases [39]. First, during the *point assignment phase*, a point is assigned to the cluster of its nearest centroid. The nearest centroid is determined by the Euclidean distance. Second, during the *centroid update phase*, the means are recalculated for each cluster. Both phases together form one k-means iteration that is repeated either until the mean squared error of the old and new centroids converges below an ϵ or a defined iteration limit is exceeded.

The time complexity of k-means is $O(Ndk)$ per iteration, i.e., the number of data points N multiplied by the features per point d and the clusters k. However, as the values of d and k are predetermined, the runtime of an iteration is linear in regard to the data size [39, 233, 317]. Thus, k-means is well-suited for processing large data sets.

Summary

In summary, the two hash join algorithms and k-means cover a spectrum of DBMS design challenges: data intensity, large state, and iterations. Focusing on well-understood operators gives us the freedom to explore these design challenges on a new hardware platform while effectively communicating our insights.

3 Scalable Data-intensive Query Processing

In this chapter, we investigate the scalability limitations of GPU co-processing and analyze how a faster interconnect helps us to overcome them by providing high bandwidth and low latency. In Figure 3.1, we show that fast interconnects enable the GPU to access CPU memory with the full memory bandwidth. Furthermore, we propose a new co-processing strategy that takes advantage of the cache-coherence provided by fast interconnects for fine-grained CPU-GPU cooperation. Overall, fast interconnects integrate GPUs tightly with CPUs and significantly reduce the data transfer bottleneck.

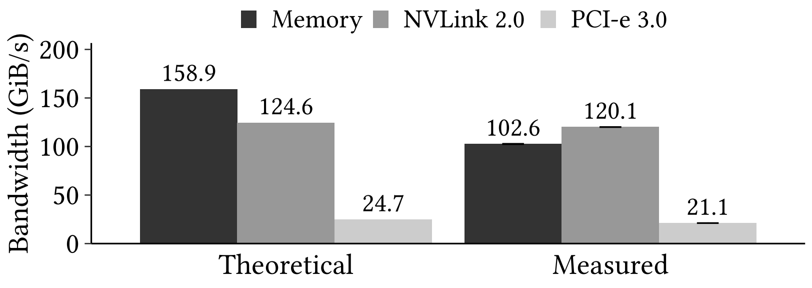


Figure 3.1: NVLink 2.0 eliminates the GPU's main-memory access disadvantage compared to the CPU.

3.1 Introduction

The data transfer bottleneck is responsible for the limited scalability of GPU-enabled DBMSs [45, 146, 161, 274, 349, 374, 424]. This bottleneck exists because current GPU interconnects such as PCI-e 3.0 [7] and PCI-e 4.0 [326] provide significantly lower bandwidth than CPU memory. We break down the transfer bottleneck into three fundamental limitations for GPU-enabled data processing:

L1: Low interconnect bandwidth. When the DBMS decides to use the GPU for query processing, it must transfer data ad hoc from CPU memory to the GPU. With current interconnects, this transfer is slower than processing the data on the CPU [78, 161, 170]. Consequently, we can only speed up data processing on GPUs by increasing the interconnect bandwidth [102, 146, 219, 385, 413]. Although data compression [136, 359] and approximation [340] can reduce transfer volume, their effectiveness varies with the data and query.

L2: Small GPU memory capacity. To avoid transferring data, GPU-enabled DBMSs cache data in GPU memory [78, 176, 218, 355]. However, GPUs have limited on-board *GPU memory* capacity (up to 32 GiB). In general, large data sets cannot be stored in GPU memory. The capacity limitation is intensified by DBMS operators that need additional space for intermediate state, e.g., hash tables or sorted arrays. In sum, GPU co-processing does not scale to large data volumes.

L3: Coarse-grained cooperation of CPU and GPU. Using only a single processor for query execution leaves available resources unused [102]. However, co-processing on multiple, heterogeneous processors inherently leads to execution skew [129, 163], and can even cause slower execution than on a single processor [78]. Thus, CPU and GPU must cooperate to ensure that the CPU's execution time is the lower bound. Cooperation requires efficient synchronization between processors on shared data structures such as hash tables or B-trees, that is not possible with current interconnects [32].

We structure our investigation based on these limitations. Our contributions are as follows:

1. We analyze NVLink 2.0 to understand its performance and new functionality in the context of data management (Section 3.2). NVLink 2.0 is one representative of the new generation of fast interconnects.

2. We investigate how fast interconnects allow us to perform efficient ad hoc data transfers (**L1**). We experimentally determine the best transfer strategy (Section 3.3).

3. We scale queries to large data volumes while considering the new trade-offs of fast

interconnects (**L2**). We use a no-partitioning hash join as an example (Section 3.4).

4. We propose a new cooperative and robust co-processing approach that enables CPU-GPU scale-up on a shared, mutable data structure (**L3**, Section 3.5).
5. We evaluate joins as well as a selection-aggregation query using a fast interconnect (Section 3.6).

The remainder of this chapter is organized as follows. In Sections 3–6 we present our contributions. Then, we present our experimental results in Section 3.6 and discuss our insights in Section 3.7. Finally, we review related work in Section 3.8 and conclude the chapter in Section 3.9.

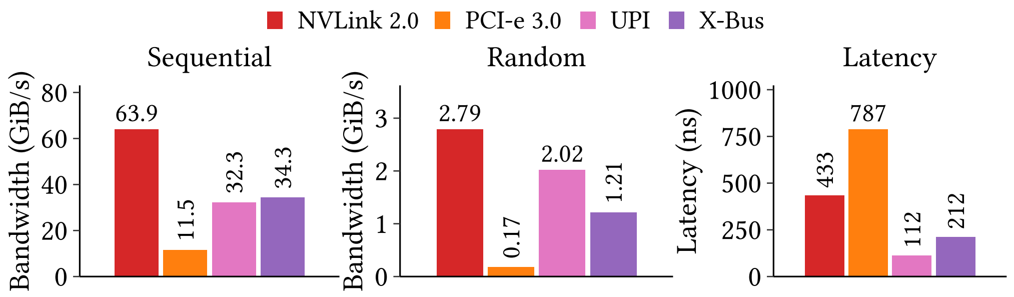
3.2 Analysis of a Fast Interconnect

In this section, we analyze the class of fast interconnects by example of NVLink 2.0 to understand their performance and new functionality in the context of data management. The main improvements of fast interconnects compared to PCI-e 3.0 are higher bandwidth, lower latency, and cache-coherence. We investigate these properties and examine the benefits and challenges for scaling co-processing.

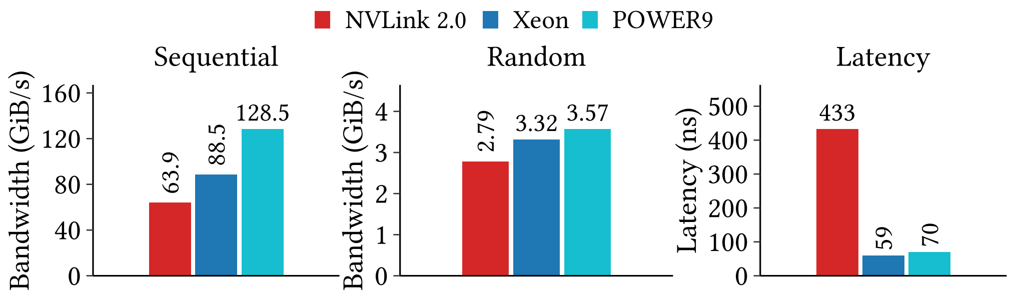
Bandwidth & Latency. We start by quantifying how much NVLink 2.0 improves the GPU's interconnect performance. We compare NVLink 2.0's ② performance to GPU (PCI-e 3.0 ①) and CPU interconnects (Intel Xeon Ultra Path Interconnect (UPI) ③, IBM POWER9 X-Bus ④), CPU memory (Intel Xeon ⑤, IBM POWER9 ⑥), and GPU memory (Nvidia V100 ⑦). We visualize these data access paths in Figure 3.3. In all our measurements we show 4-byte read accesses on 1 GiB of data. We defer giving further details on our measurement setup and methodology to Section 3.6.1.

We first compare NVLink 2.0 to the other GPU and CPU interconnects in Figure 3.2(a). Our measurements show that NVLink 2.0 has 5.5× more sequential bandwidth than PCI-e 3.0, and twice as much as UPI and X-Bus. Random accesses patterns are 16× faster than PCI-e 3.0, and 38% faster than UPI. However, while the latency of NVLink 2.0 is 45% lower than PCI-e 3.0, it is 3.8× higher than UPI and 2× higher than X-Bus. Overall, NVLink 2.0 is significantly faster than PCI-e 3.0, and more bandwidth-oriented than the CPU interconnects.

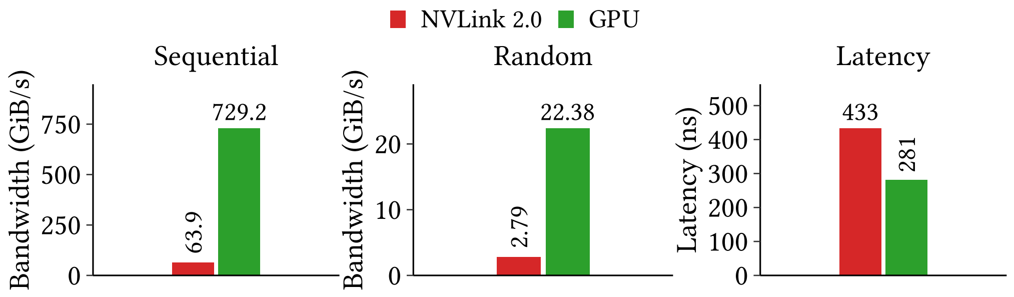
Next, we show the NVLink 2.0 vs. CPU memory in Figure 3.2(b). We note that the IBM CPU has 8 DDR4-2666 memory channels, while the Intel Xeon only has 6 channels of the same memory type. We see that for sequential accesses, the Intel Xeon and IBM POWER9 have 38% and 2× higher bandwidth than NVLink 2.0, respectively. For random



(a) NVLink 2.0 vs. CPU & GPU Interconnects.

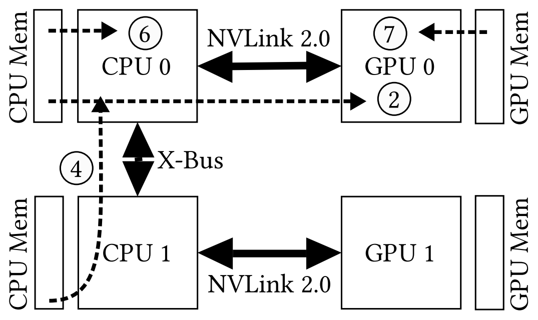


(b) NVLink 2.0 vs. CPU memory.

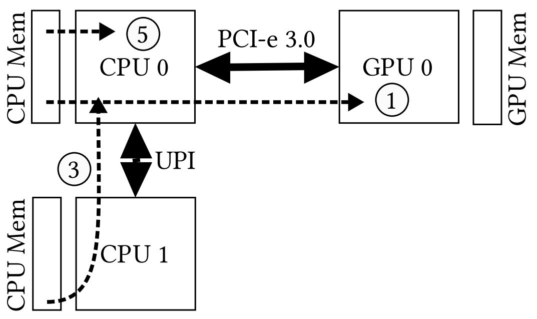


(c) NVLink 2.0 vs. GPU memory.

Figure 3.2: Bandwidth and latency of memory reads on IBM and Intel systems with Nvidia GPUs. Compare to data access paths shown in Figure 3.3.



(a) 2× IBM POWER9 with 2× Nvidia V100-SXM2.



(b) 2× Intel Xeon with 1× Nvidia V100-PCIE.

Figure 3.3: Data access paths on IBM and Intel systems.

accesses, NVLink 2.0 is 16% slower than the Intel Xeon and 22% slower than the IBM POWER9. The latency of NVLink 2.0 is 6–7× higher than the latency of CPU memory. We take away that, although NVLink 2.0 puts the GPU within a factor of two of the CPUs' bandwidth, it adds significant latency.

Finally, in Figure 3.2(c), we compare GPU accesses to CPU memory over NVLink 2.0 with GPU memory. We observe that both access patterns have an order-of-magnitude higher bandwidth in GPU memory, but that latency over NVLink 2.0 is only 54% higher. As GPUs are designed to handle such high-latency memory accesses [183, 408], they are well-equipped to cope with the additional latency of NVLink 2.0.

Cache-coherence. Cache-coherence simplifies the practical use of NVLink 2.0 for data processing. The advantages are three-fold. First, the GPU can directly access any location in CPU memory, therefore pinning memory becomes unnecessary. Second, allocating pageable memory is faster than allocating pinned memory [142, 278, 403]. Third, the operating system and database are able to perform background tasks that are important for long-running processes, such as memory defragmentation [107] and optimizing NUMA locality through page migration [251].

In contrast, the non-cache-coherence of PCI-e has two main drawbacks. First, data consistency must be managed in software instead of in hardware. The programmer either manually flushes the caches [304], or the OS migrates pages [296]. Second, system-wide atomics are unsupported. Instead, a work-around is provided by first migrating Unified Memory pages to GPU memory, and then performing the atomic operation in GPU memory [300].

Research shows that adding fine-grained cache-coherence to PCI-e is not feasible due to its high latency [151]. However, NVLink 2.0 removes these limitations [191] and thus is better-suited for data processing.

Benefits. We demonstrate three benefits of NVLink 2.0 for data processing with a no-partitioning hash join. First, we are able to scale the probe-side relation to arbitrary data volumes due to NVLink 2.0's high sequential bandwidth. With the hash table stored in GPU memory, we retain the GPU's performance advantage compared to a CPU join. Second, we provide build-side scalability to arbitrary data volumes using NVLink 2.0's low latency and high random access bandwidth. Thus, we are able to spill the hash table from GPU to CPU memory. Third, we employ the cache-coherence and system-wide atomics of NVLink 2.0 to share the hash table between a CPU and a GPU and scale-up data processing.

Challenges. Despite the benefits of NVLink 2.0 for data processing, translating high interconnect performance into high-performance query processing will require addressing the following challenges.

First, an out-of-core GPU join operator must perform both data access and computation efficiently. Early GPU join approaches cannot saturate the interconnect [170, 171]. More recent algorithms saturate the interconnect, and are optimized to access data over a low-bandwidth interconnect [213, 385]. This can involve additional partitioning steps on the CPU [385]. We investigate how a GPU join operator can take full advantage of the higher interconnect performance.

Second, scaling the build-side volume beyond the capacity of GPU memory in a NP-HJ requires spilling the hash table to CPU memory. However, spilling to CPU memory implies that the GPU performs irregular accesses to CPU memory, as, by design, hash functions map keys to uniformly distributed memory locations. Such irregular accesses are inefficient over high-latency interconnects. For this reason, previous approaches either cannot scale beyond GPU memory [170, 213], or are restricted to partitioning-based joins [385]. Higher interconnect performance requires us to reconsider how well a NP-HJ that spills to CPU memory performs on GPUs.

Third, fully exploiting a heterogeneous system consisting of CPUs and GPUs requires them to cooperatively process the join. We must take into account data locality, synchronization costs, and the differences in hardware architectures to achieve efficiency.

3.3 Efficient Data Transfer between CPU and GPU

In order to process data, the GPU needs to read input data from CPU memory. Since the GPU memory is limited to tens of gigabytes, we cannot store a large amount of data on the GPU. As a consequence, any involvement of the GPU in data processing requires ad

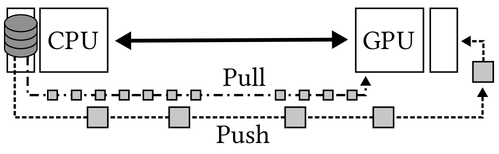


Figure 3.4: Push- vs. pull-based data transfer methods.

Table 3.1: An overview of GPU transfer methods.

Method	Semantics	Level	Granularity	Memory
Pageable Copy	Push	SW	Chunk	Pageable
Staged Copy				
Dynamic Pinning				
Pinned Copy				Pinned
UM Prefetch				Unified
UM Migration	Pull	OS	Page	Unified
Zero-Copy		HW	Byte	Pinned
Coherence				Pageable

hoc data transfer, which makes interconnect bandwidth the most critical resource (L1).

We can choose between different strategies to initiate data transfers between CPU and GPU. Each strategy shows different performance on the same interconnect. In this section, we discuss these data transfer strategies to identify the most efficient way for data transfer. We build on these insights in the following sections.

Recent versions of CUDA provide a rich set of APIs that abstract the MMIO and DMA transfer primitives described in Section 2.3.1. From these APIs, we derive eight transfer methods that we list in Table 3.1. We divide these methods into two categories based on their semantics, *push-based* and *pull-based*. On a high level, push-based methods perform course-grained transfers to GPU memory, whereas in pull-based methods the GPU directly accesses CPU memory. We depict these differences in Figure 3.4. We first describe push-based methods, and then pull-based methods.

3.3.1 Push-based Transfer Methods

In order to transfer data, push-based methods rely on a pipeline to hide transfer latency. The pipeline is implemented in software and executed by the CPU. We describe the pipeline stages of each method and contrast their differences.

Pageable Copy. Pageable Copy is the most basic method to copy data to the GPU. It is exposed in the API via the `cudaMemcpyAsync` function, and transfers data directly from any location in pageable memory. As the API is called on the CPU, data are *pushed* to the GPU. Before we setup the pipeline, we split the data into chunks. Subsequently, we setup a two-stage pipeline by first transferring each chunk to the GPU, and then processing the chunk on the GPU. As both steps can be performed in parallel, the computation overlaps with the transfer.

Pinned Copy. As Nvidia recommends using pinned memory instead of pageable memory for data transfer [303], we apply the same technique as in Pageable Copy to pinned memory. Thus, the hardware can perform DMA using the copy engines instead of using a CPU thread to copy via memory-mapped I/O. Therefore, Pinned Copy is typically faster than Pageable Copy, but requires the database to store all data that is accessed by the GPU in pinned memory.

Staged Copy. However, storing all data in pinned memory violates Nvidia's recommendation to consider pinned memory as a scarce resource [303], and pinning large amounts of memory complicates memory management. Therefore, we setup a pinned staging buffer for the copy. In the pipeline, we first copy a chunk of data from pageable memory into the pinned memory buffer. Then, we perform the transfer and compute stages. We thus pipeline the transfer at the expense of an additional copy operation within CPU memory.

Dynamic Pinning. CUDA supports pinning pages of preexisting pageable memory. This allows us to pin pages ad hoc before we transfer data to the GPU, avoiding an additional copy operation in CPU memory. After that, we execute the copy and compute stages.

Unified Memory Prefetch. If we use Unified Memory and know the data access pattern beforehand, we can explicitly prefetch a region of unified memory to the GPU before the access takes place. This avoids a drawback of the Unified Memory Migration method that we describe next, namely that migrating pages on-demand has high latency and stalls the GPU [432]. We execute the transfer in a two-stage pipeline that consists of prefetching a chunk of data to the GPU, and then running the computation. Thus, prefetching requires a software pipeline in addition to using Unified Memory.

3.3.2 Pull-based Transfer Methods

Many database operators access memory irregularly, especially operators based on hashing. Hashed accesses are irregular, because hash functions are designed to generate uniform and randomly distributed output. These accesses are also data-dependent, as the hash function's input are attributes of a relation (e.g., the primary key).

Push-based transfer methods cannot handle these types of memory access. The CPU decides which data are transferred to the GPU. Thus, the GPU has no control over which data it processes, and cannot satisfy data-dependencies.

In contrast, pull-based methods are able to handle data-dependencies, as they intrinsically request data. In the following, we introduce three pull-based transfer methods.

Unified Memory Migration. Instead of dealing with pageable and pinned memory inside the database, Unified Memory allows us to delegate data transfer to the operating system. Internally, memory pages are migrated to the GPU on a page access [432] (4 KiB on Intel CPUs, 64 KiB on IBM CPUs [286]). Therefore, the GPU *pulls* data, and pipelining in software is unnecessary. However, the database must explicitly allocate Unified Memory to store data.

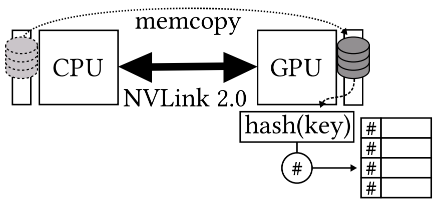
Zero-Copy. The previous approaches involve software or the operating system to manage transferring data. In contrast, we can use Unified Virtual Addressing to directly access data in CPU memory during GPU execution. We are able to load data with byte-wise granularity, but are restricted to accessing pinned memory. As Zero-Copy is managed entirely in hardware, software or operating system are not involved.

Coherence. NVLink 2.0 offers a new transfer method that is unavailable with previous interconnects. Using the hardware address translation services and cache-coherence, the GPU can directly access any CPU memory during execution. In contrast to Unified Memory Migration, NVLink 2.0 accesses memory with byte-wise granularity. In contrast to Unified Virtual Addressing, NVLink 2.0 does not require pinned memory. Instead, the GPU is able to directly access pageable CPU memory. Thus, NVLink 2.0 lifts previous constraints on the memory type and access granularity.

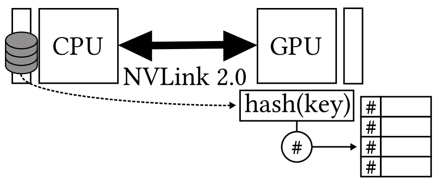
3.4 Scaling GPU Hash Joins to Arbitrary Data Sizes

Current algorithms and systems for data processing on GPUs are all limited to some degree by the capacity of GPU memory (L2). Being limited by GPU memory capacity is the most fundamental problem in adopting GPU acceleration for data management in practice. In this section, we study how fast interconnects enable us to efficiently scale up data processing to arbitrary database sizes.

We study the impact of fast interconnects on the example of a no-partitioning hash join because of its unique requirements: (1) The build phase performs random memory accesses and thus requires either a low-latency interconnect to access the hash table in CPU memory, or enough GPU memory to store the hash table. The latter is a common scalability constraint. (2) The probe phase puts high demands on the interconnect's bandwidth. We discuss how we can scale up the probe side (Section 3.4.1) and the build



(a) Data and hash table in GPU memory.



(b) Data in CPU memory and hash table in GPU memory.

Figure 3.5: Scaling the probe side to any data size.

side (Section 3.4.2), respectively, and propose our hybrid hash table approach to improve performance (Section 3.4.3).

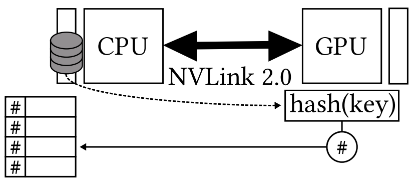
3.4.1 Scaling the Probe Side to Any Data Size

Transferring the inner and outer relations on-the-fly allows us to scale the relations' cardinalities regardless of GPU memory capacity. We begin by describing a simple, *baseline join* [158, 422] that is non-scalable. After that, we remove the probe-side cardinality limit by comparing the baseline to the *Zero-Copy pull-based join* introduced by Kaldewey et al. [213]. Based on the Zero-Copy join, we contribute our *Coherence join* that uses the Coherence transfer method. To simplify the discussion, we focus on pull-based methods. However, at the cost of additional complexity, we could instead use push-based pipelines to achieve probe-side scalability [170, 171].

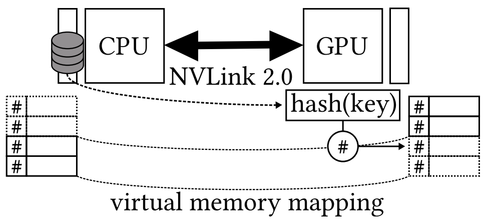
First, in the baseline approach that we show in Figure 3.5a, we first copy the entire build-side relation R to GPU memory. When the copy is complete, we build the hash table in GPU memory. Following that, we evict R and copy the probe-side relation S to GPU memory. We probe the hash table and emit the join result (i.e., an aggregate or a materialization). The benefit of this approach is that it only requires the hardware to support synchronous copying. However, this baseline doesn't scale to arbitrary data sizes, as it is limited by the GPU's memory capacity.

Next, in Figure 3.5b, we illustrate our probe-side scalable join. By using a pull-based transfer method, we are able to remove the scalability limitation. Zero-Copy and Coherence enable us to access CPU memory directly from the GPU (i.e., by dereferencing a pointer). Therefore, we build the hash table on the GPU by pulling R tuples on-demand from CPU memory. Behind the scenes, the hardware manages the data transfer. After we finish building the hash table, we pull S tuples on-demand and probe the hash table.

Finally, we replace the Zero-Copy transfer method with the Coherence transfer method in the Zero-Copy join. The Zero-Copy method requires the base relations to be



(a) Data and hash table in CPU memory.



(b) Data in CPU memory and hash table spills from GPU memory into CPU memory.

Figure 3.6: Scaling the build side to any data size.

stored in pinned memory. However, databases typically store data in pageable memory. We enable the GPU to access any memory location in pageable memory by replacing Zero-Copy with Coherence, which simplifies GPU data processing.

3.4.2 Scaling the Build Side to Any Data Size

We assume that the hash table is small enough to fit into GPU memory in Section 3.4.1. This limits the cardinality of R. We now lift this limitation and consider large hash tables.

We show our build-side scalable join in Figure 3.6a. The join is based on our probe-side scalable join, that we introduce in Section 3.4.1. However, in contrast to our probe-side scalable join, we store the hash table in CPU memory. By storing the hash table in CPU memory instead of in GPU memory, we are no longer constrained by the GPU's memory capacity.

In contrast to reading in base relations, hash table operations (insert and lookup) are data-dependent and have an irregular memory access pattern. Pull-based transfer methods (e.g., Coherence) enable us to perform these operations on the GPU in CPU memory. As we typically allocate memory specifically to build the hash table, we can allocate pinned memory or Unified Memory for use with the Zero-Copy or Unified Memory Migration methods. This flexibility allows us to choose the optimal transfer method for our hardware.

3.4.3 Optimizing the Hash Table Placement

Although the Coherence transfer method enables the GPU to access any CPU memory location, access performance is non-uniform and varies with memory locality. CPU memory is an order-of-magnitude slower than GPU memory for random accesses (see Section 3.2). We combine the advantages of both memory types in our new *hybrid hash*

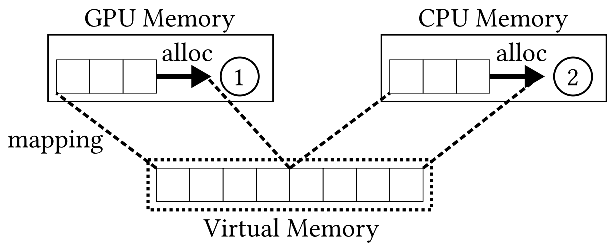


Figure 3.7: Allocating the hybrid hash table.

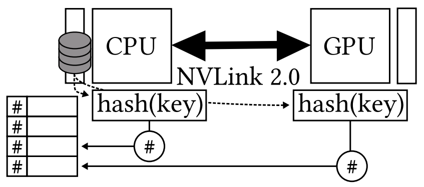
table. We design the hybrid hash table such that access performance degrades gracefully when the hash table's size is increased.

In Figure 3.6b, we show that our hybrid hash table can replace a hash table in CPU memory without any modifications to the join algorithm. This is possible because the hybrid hash table uses virtual memory to abstract the physical location of memory pages. We use virtual memory to combine GPU pages and CPU pages into a single, contiguous array. Virtual memory has been available previously on GPUs [217]. However, fast interconnects integrate the GPU into a system-wide address space, which enables us to map physical CPU pages next to GPU pages in the address space.

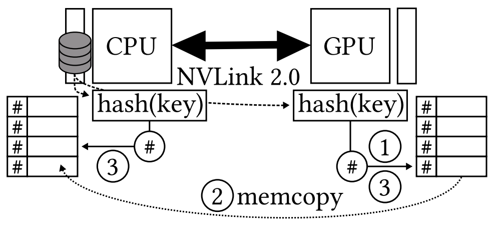
We allocate the hybrid hash table using a greedy algorithm, that we depict in Figure 3.7. By default, ① we allocate GPU memory. If the hash table is small enough, we allocate the entire hash table in GPU memory. Otherwise, ② if not enough GPU memory is available, we allocate memory on the CPU that is nearest to the GPU. Therefore, we spill the hash table to CPU memory. If that CPU has insufficient memory, we recursively search the next-nearest CPUs of a multi-socket NUMA system until we have allocated sufficient memory for the hash table. Overall, we allocate part of the hash table in GPU memory, and part in CPU memory.

The hybrid hash table is optimized for handling the worst case of a uniform join key distribution. We model this case as follows. We assume that the hash table consists of G_{mem} and C_{mem} bytes of GPU and CPU memory. We then expect that $A_{GPU} = \frac{G_{mem}}{G_{mem}+C_{mem}}$ of all accesses are to GPU memory. We estimate hash join throughput to be $J_{tput} = A_{GPU}G_{tput} + (1 - A_{GPU})C_{tput}$, where G_{tput} and C_{tput} are the hash join throughputs when the hash table resides in GPU and CPU memory, respectively. Overall, throughput is determined by the proportion of accesses to a given processor.

There are two additional benefits to our hybrid hash table that cannot be replicated without hardware support. First, the contiguous array underlying the hybrid hash table comes at zero additional cost, because processors perform virtual-to-physical address translation regardless of memory location. We could simulate a hybrid hash



(a) Cooperatively process join on CPU and GPU with hash table in CPU memory.



(b) Build hash table on GPU, copy the hash table to processor-local memories, and then cooperatively probe on CPU and GPU.

Figure 3.8: Scaling-up using CPU and GPU.

table on hardware without a system-wide address space by mapping together two non-contiguous arrays in software. However, the software indirection would add extra cycles on each access. Second, besides a change to the allocation logic, we leave the hash join algorithm unmodified. Thus, our hybrid hash table can easily be integrated into existing databases.

3.5 Scaling-up using CPU and GPU

The third fundamental limitation (L3) of GPU co-processing is single processor execution. Without a way that enables CPUs and GPUs to collaborate in query processing, we leave available processing resources unused and cannot take full advantage of a heterogeneous CPU+GPU system.

In this section, our goal is to increase throughput by utilizing all available processors cooperatively, i.e., combining CPUs and GPUs. The main challenge is to guarantee that performance always improves when we schedule work on a GPU, even for the first query that is executed on the GPU. For this, the scheduling approach must be highly robust with respect to execution skew. As a consequence, truly scalable co-processing has the following three requirements. (a) We must process chunks of input data such that we can exploit data parallelism to use CPU and GPU for the same query. (b) At the same time, the task scheduling approach needs to avoid load imbalances. (c) The approach must avoid resource contention (e.g., of memory bandwidth) to prevent slowing down the overall execution time.

We first propose a heterogeneous task scheduling scheme. Following that, we optimize our hash table placement strategy for co-processing. Finally, we describe scaling up on multiple GPUs that are connected with a fast interconnect.

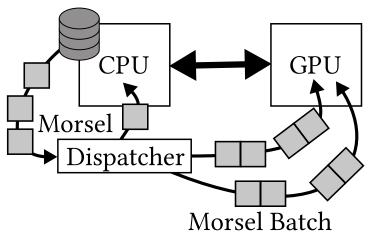


Figure 3.9: Dynamically scheduling tasks to CPU and GPU processors.

3.5.1 Task Scheduling

Load imbalances inherently occur on heterogeneous architectures due to the relative throughput differences of processors. As the throughput of a processor depends on many variable parameters that change over time (e.g., query, data, processor clock speeds), we cannot know the relative differences upfront. A task scheduler ensures that all processors deliver their highest possible throughput.

We adapt the CPU-oriented, morsel-driven approach [63, 247] for GPUs. In the CPU-oriented approach, all cores work concurrently on the same data and, in the case of joins, the same hash table. Cores balance load by requesting fixed-sized chunks of data (i.e., *morsels*) from a central dispatcher, that is implemented as a read cursor. Each core advances at its own processing rate.

In Figure 3.9, we show our heterogeneous scheduling approach. In contrast to the CPU-oriented approach, we give each processor the right amount of work to minimize execution skew by considering the increased latency of scheduling work on a GPU, and the higher processing rate of the GPU. Instead of dispatching one morsel at-a-time, we dispatch batches of morsels to the GPU concurrently. Batching morsels amortizes the latency of launching a GPU kernel over more data. Concurrently scheduling batches on two separate work queues hides scheduling latency and ensures that the GPU is always occupied. We empirically tune the batch size to our hardware.

3.5.2 Heterogeneous Hash Table Placement

Processors are fastest when accessing their local memories. Consequently, our hybrid hash table (Section 3.7) prefers data in GPU memory, and spills to CPU memory only when necessary. In our hybrid hash table, however, we consider only a single processor. In this section, we optimize for multiple, heterogeneous processors accessing the hash table via a fast interconnect. We consider two cases: one globally shared hash table, and

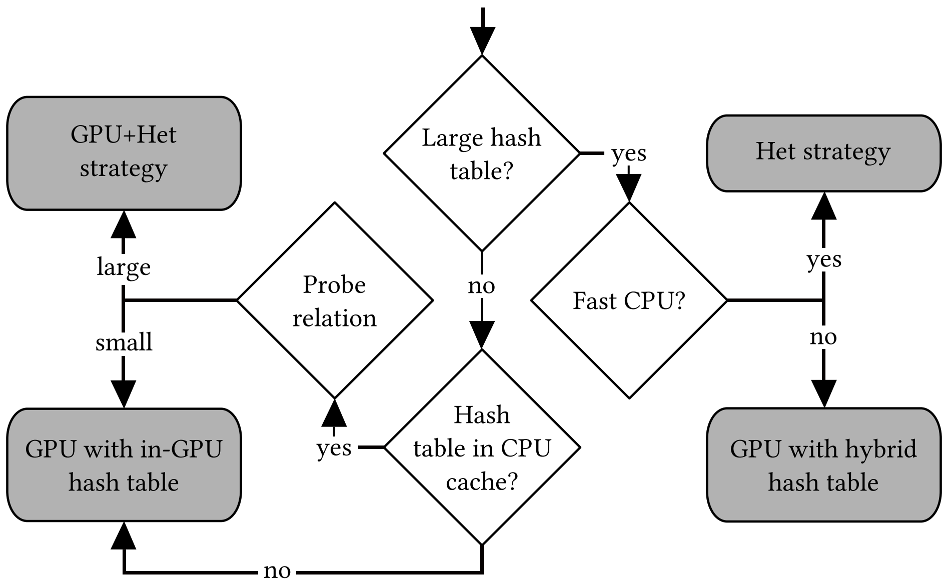


Figure 3.10: Hash table placement decision.

multiple per-processor hash tables. We summarize the placement decision process in Figure 3.10.

In Figure 3.8a, we show the CPU and GPU processing a join using a globally shared hash table (*Het* strategy). Globally sharing a hash table retains the build-side scaling behavior that we achieve in Section 3.4.2. However, we avoid our hybrid hash table optimization and store the hash table in CPU memory. We choose this design because we aim to always speed up processing when using a co-processor. Therefore, we avoid slowing down CPU processing through remote GPU memory accesses. In addition, the CPU has significantly lower performance when accessing GPU memory than the GPU accessing CPU memory, due to the CPU coping worse than the GPU with the high latencies of GPU memory and the interconnect [191].

We handle the special case of small build-side relations separately (*GPU + Het* strategy), because processors face contention when building the hash table. Furthermore, small hash tables allow us to optimize hash table locality. We show our small table optimization in Figure 3.8b. In a first step, ① one processor (e.g., the GPU) builds the hash table in processor-local memory (in this case, GPU memory). Following that, ② we copy the finished hash table to all other processors. By storing a local copy of the hash table on each processor, we ensure that all processors have high random-access bandwidth to the hash table. Finally, ③ we execute the probe phase on all processors using our heterogeneous scheduling strategy. Our strategy could be extended to multi-way joins

Table 3.2: Workload Overview.

Property	A (from [63])	B	C (from [226])
key / payload	8 / 8 bytes	8 / 8 bytes	4 / 4 bytes
cardinality of R	2^{27} tuples	2^{18} tuples	$1024 \cdot 10^6$ tuples
cardinality of S	2^{31} tuples	2^{31} tuples	$1024 \cdot 10^6$ tuples
total size of R	2 GiB	4 MiB	7.6 GiB
total size of S	32 GiB	32 GiB	7.6 GiB

(e.g., for a star schema) by building hash tables on a different processor in parallel, and then copying all hash tables to all processors. However, we focus on investigating fast interconnects using a single join.

3.5.3 Multi-GPU Hash Table Placement

Systems with multiple GPUs are connected in a mesh topology similar to multi-socket CPU systems. For small hash tables, we can use the GPU+Het execution strategy with multiple GPUs. However, for large hash tables, multi-GPU systems can distribute the hash table over multiple GPUs, as GPUs are latency insensitive [183, 408]. We distribute the table by interleaving the pages over all GPUs. This strategy is used in NUMA systems [247]. Fast interconnects enable us to use the strategy in multi-GPU systems.

In contrast to CPU+GPU execution, distributing computation over multiple GPUs provides three distinct advantages. First, using only GPUs avoids computational skew. Second, distributing large hash tables within GPU memory frees CPU memory bandwidth for loading the base relations. Finally, interleaving the hash table over multiple GPUs utilizes the full bi-directional bandwidth of fast interconnects, as opposed to the mostly uni-directional traffic of the Het strategy.

3.6 Evaluation

In this section, we evaluate the impact of NVLink 2.0 on data processing. We describe our setup in Section 3.6.1. After that, we present our results in Section 3.6.2.

3.6.1 Setup and Configuration

We first introduce our methodology and experimental setup. Then, we describe the data sets that we use in our evaluation. Finally, we introduce our experiments.

Environment. We evaluate our experiments on one GPU and two CPU architectures. We conduct our GPU measurements using an Nvidia Tesla V100-SXM2 and a V100-PCIE ("Volta"), on IBM and Intel systems, respectively. Both GPUs have 16 GB memory. We conduct our CPU measurements on a dual-socket IBM POWER9 at 3.8 GHz with 2×16 cores and 256 GB memory, and on a dual-socket Intel Xeon Gold 6126 ("Skylake-SP") at 2.6 GHz with 2×12 cores and 1.5 TB memory. The Intel system runs Ubuntu 16.04, and the IBM POWER9 system runs Ubuntu 18.04. We implement our experiments in C++ and CUDA. We use CUDA 10.2 and GCC 8.4.0 on all systems, and compile all code with "-O3" and native optimization flags.

Methodology. We measure throughput of the end-to-end join. We define join throughput as the sum of input tuples divided by the total runtime (i.e., $\frac{|R|+|S|}{\text{runtime}}$) [367, 385]. For each experiment, we report the mean and standard error over 10 runs. We note that our measurements are stable with a standard error less than 5% from the mean.

Workloads. In Table 3.2, we give an overview of our workloads. We specify workloads A and C similar to related work [48, 63, 226]. We scale these workloads 8× to create an out-of-core scenario. We define workload B as a modified workload A with a relation R that fits into the CPU L3 and GPU L2 caches and represents small dimension tables. All workloads assume narrow 8- or 16-byte <key, value> tuples. We generate tuples assuming a uniform distribution, and a foreign-key relationship between R and S. Unless noted otherwise, each tuple in S has exactly one match in R. We store the relations in a column-oriented storage model.

Settings. In the following experiments, we use the Coherence transfer method for NVLink 2.0 and the Zero Copy method for PCI-e 3.0, unless noted otherwise. We set up our no-partitioning hash join with perfect hashing, i.e., we assume no hash conflicts occur due to the uniqueness of primary keys. Our join is equivalent to the NOPA join described by Schuh et al. [367]. We allocate memory as 2 MiB huge pages on the NUMA node closest to the GPU, and preallocate the pages at boot time [285] to avoid page fragmentation. On CPUs, we explicitly bind threads to CPU cores. On the POWER9 CPU, we tune memory reads by disabling stride-N prefetching (`DSCR` = 0) [188], as we observed that stride-N prefetching reduces sequential bandwidth. However, sequential prefetching remains enabled.

Baseline. As a CPU baseline, we use the radix partitioned, multi-core hash join implementation ("PRO") provided by Barthels et al. [53]. We modify the baseline to use our perfect hash function, thus transforming the PRO join into a PRA join [367]. Furthermore, we tune our baseline to use the best radix bits (12 bits), page size (huge pages), SMT (enabled), software write-combine buffers (enabled) and NUMA locality parameters for our hardware. As our experiments run on one GPU, we run the baseline on one CPU.

Experiments. We conduct ten experiments. First, we evaluate the impact of transfer

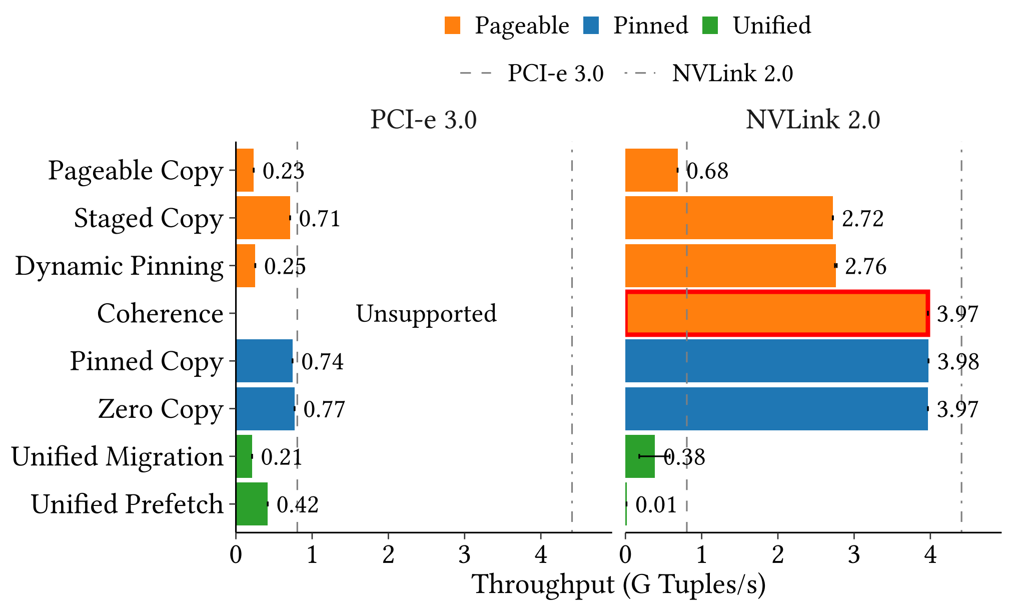


Figure 3.11: No-partitioning hash join using different transfer methods for PCI-e 3.0 and NVLink 2.0.

methods on data processing when using PCI-e 3.0 and NVLink 2.0. Then, we show the impact of NUMA locality considering the base relations and the hash table. Next, we explore out-of-core scalability when exceeding the GPU memory capacity with TPC-H query 6, the probe-side relation S, and the build-side relation R. Furthermore, we investigate the performance impact of different build-to-probe ratios, as well as skewed data and varying the join selectivity. Lastly, we investigate heterogeneous cooperation between a CPU and a GPU that share a hash table.

3.6.2 Experiments

In this section, we present our experimental results and describe our observations.

GPU Transfer Methods

In Figure 3.11, we show the join throughput of each transfer method with PCI-e 3.0 and NVLink 2.0 for workload A (2 GiB $\bowtie$ 32 GiB). The outer relation is thus larger than GPU memory. We load both relations from CPU memory, and build the hash table in GPU memory.

PCI-e 3.0. We observe that pinning the memory is necessary to reach the peak transfer bandwidth of 11.4 GB/s. The Staged Copy method is within 8% of Zero Copy, despite

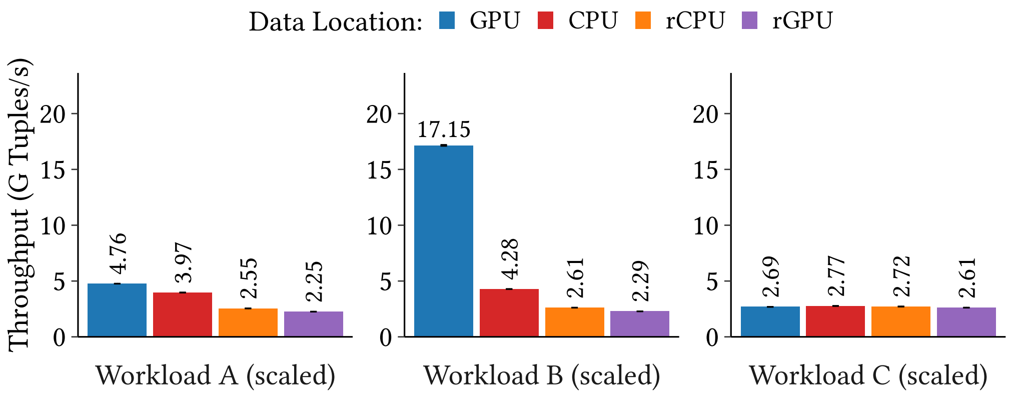


Figure 3.12: Join performance of the GPU when the base relations are located on different processors, increasing the number of interconnect hops from 0 to 3.

copying from pageable memory. The hidden cost of using pageable memory is that we utilize 8 CPU cores to stage the data into pinned buffers. In contrast, Unified Migration and Unified Prefetch are 72% and 45% slower than Zero Copy. Although prefetching avoids the cost of demand-paging, we observe the overheads of evicting cached pages and mapping new pages in GPU memory. Pageable Copy and Dynamic Pinning are both significantly slower than Zero Copy. We note that the Coherence method is unsupported by PCI-e 3.0, due to PCI-e being non-cache-coherent.

NVLink 2.0. In contrast to PCI-e 3.0, NVLink 2.0 achieves up to 5× higher bandwidth. The Coherence method is within 8% of the maximum possible throughput. The throughput of Zero Copy and Pinned Copy match that of Coherence, despite using pinned memory instead of pageable memory. Transfers from pageable memory without using cache-coherence (i.e., Pageable Copy, Staged Copy, Dynamic Pinning) all achieve less throughput than Coherence. NVLink underperforms PCI-e in only two cases, when using either Unified Memory method[1]. Overall, the Coherence and Zero Copy methods are fastest, and NVLink 2.0 shows significantly higher throughput than PCI-e 3.0.

Data Locality

We measure the impact of base relation locality in Figure 3.12. We process the workloads from Table 3.2, and scale them down to fit into GPU memory (13 GiB, 12 GiB, and 10 GiB). We store R and S in GPU memory, CPU memory, remote CPU memory, and remote GPU

[1]We speculate that this is due to the POWER9 driver implementation receiving less optimization than on x86-64.

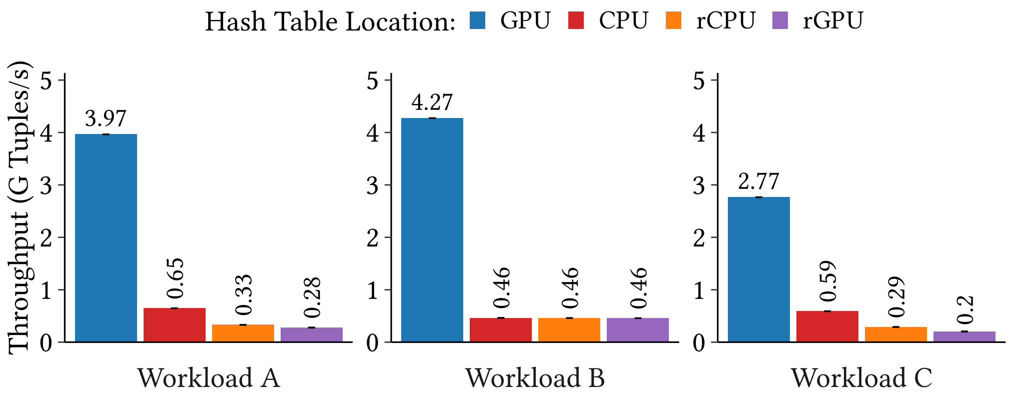


Figure 3.13: Join performance of the GPU when the hash table is located on different processors, increasing the number of interconnect hops from 0 to 3.

memory (compare to Figure 3.3(a)). Each step increases the number of interconnect hops to load the data. In all measurements, we store the hash table in GPU memory.

Workload A. We observe that join throughput decreases by 16–53% as we increase the number of hops. We see that going from 1 to 2 hops has a larger effect than from 2 to 3 hops, because the X-Bus interconnect has lower throughput than NVLink 2.0 (cf. Figure 3.2(a)).

Workload B. We notice that storing the memory in GPU memory has 4× higher throughput than a single hop over NVLink 2.0. The reason is that the hash table is cached in the GPU's L2 cache, which has higher random access bandwidth than GPU memory. For this workload, there is a 46% penalty for traversing three interconnects instead of one.

Workload C. In contrast to the best-case scenario represented by B, C is a worst-case scenario, as the relations have equal cardinalities (i.e., $|R| = |S|$). As a result, random memory accesses to GPU memory dominate the workload.

Summary. Overall, NVLink 2.0 is not the bottleneck for hash joins that randomly access GPU memory. In addition, increasing the number of hops is mainly limited by the X-Bus' bandwidth.

Hash Table Locality

In Figure 3.13, we measure the influence of hash table locality on join performance. We process workloads A–C that have up to 34 GiB of data, and increase the interconnect

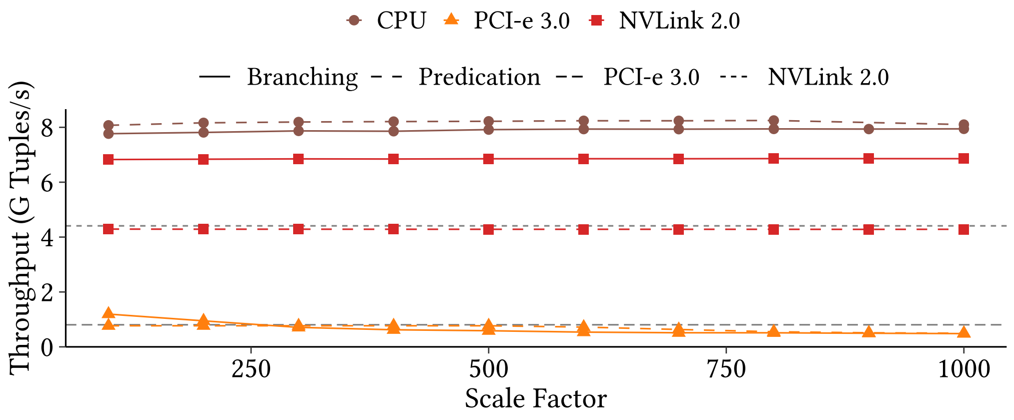


Figure 3.14: Scaling the data size of TPC-H query 6.

hops to the hash table. In all measurements, we store the base relations in local CPU memory that is one hop away over NVLink 2.0.

Workloads A and C. We see that a single NVLink 2.0 hop causes an 78–83% throughput decrease. Adding a second hop and third hop effects another 50% and 15–31%, respectively.

Workload B. We observe that, in contrast to GPU memory, the small hash table is not cached in the GPU's L2 cache for NVLink 2.0. The L2 cache is memory-side [431], and cannot cache remote data. We conclude that reducing random access bandwidth and increasing latency has a significant impact on join throughput.

Selection and Aggregation Scaling

We scale TPC-H query 6 from scale factor 100 to 1000 in Figure 3.14. This constitutes a working set of 8.9–89.4 GiB. We assume that no data are cached in GPU memory, thus all data are read from CPU memory. We run branching and predicated variants. The CPU is an IBM POWER9, for which we set 4-way SMT in the branching variant and ensure that predication uses SIMD instructions.

Interconnects. The CPU achieves the highest throughput, and outperforms NVLink 2.0 by up to 20% and PCI-e 3.0 by up to 16.2×. However, NVLink 2.0 achieves a speedup of up to 13.8× over PCI-e 3.0, thus considerably closing the gap between the GPU and the CPU.

Branching vs. Predication. Counterintuitively, branching performs better than predication on the GPU with NVLink 2.0. This is caused by the query's low selectivity of only 1.3%, that enables us to skip transferring parts of the input data. In contrast,

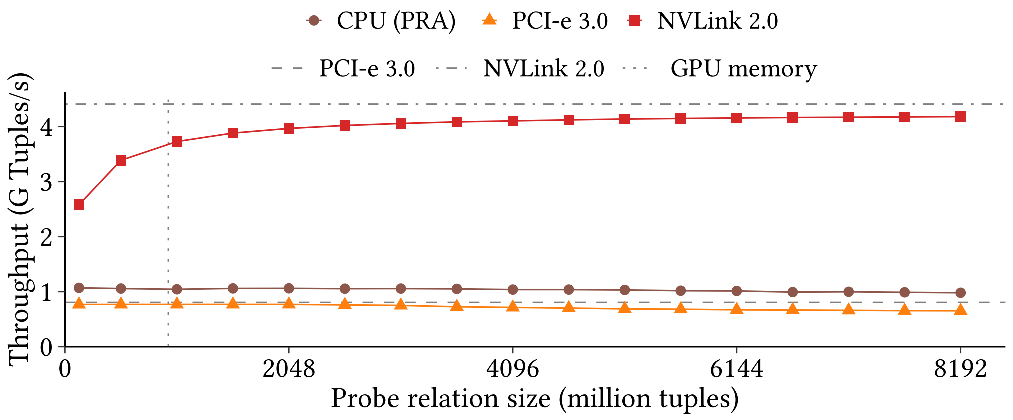


Figure 3.15: Scaling the probe-side relation.

predication loads all data and is thus bounded by the interconnect bandwidth.

Overall, NVLink 2.0 significantly narrows the gap between the CPU and the GPU for computationally light workloads, and enables the GPU to process large data volumes.

Probe-side Scaling

We analyze the effect of scaling the probe-side relation on join throughput in Figure 3.15. We use workload C with 16-byte tuples, and increase the probe-side's cardinality from 128–8196 million tuples (1.9–122 GiB). We store the base relations in CPU memory, and the hash table in GPU memory.

Observations. We notice that the throughput of NVLink 2.0 is 3.3–6.4× faster than PCI-e 3.0 and 2.4–4.2× faster than the CPU baseline. Throughput improves with larger data due to the changing build-to-probe ratio, that we investigate in detail in Section 3.6.2. In contrast, the throughput of PCI-e 3.0 remains constant, because of the transfer bottleneck. Thus, PCI-e 3.0 cannot outperform the CPU baseline.

Overall, we are able to process data volumes larger than the GPU's memory capacity at a faster rate than the CPU.

Build-side Scaling

In Figure 3.16, we scale the hash table size up to 2× the GPU memory capacity. The total data size reaches up to 91.5 GiB, counting both base relations plus the hash table. While scaling, we examine the effect of hash table placement strategies (see Section 3.4.3). We use workload C with 16-bytes tuples and increase the cardinality of both base relations.

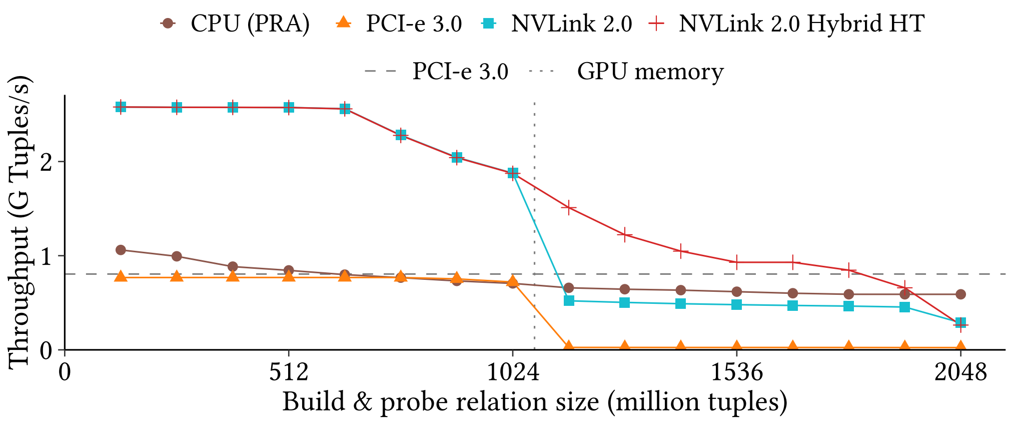


Figure 3.16: Scaling the build-side relation.

PCI-e 3.0. We note that throughput reaches 0.77 G Tuples/s as long as the hash table can be stored in GPU memory, which is 0–27% slower than the CPU baseline. For hash tables that are larger, throughput declines by 97% to 0.02 G Tuples/s, which is 22× slower than the CPU baseline.

NVLink 2.0. Our first observation is that throughput is 3.3× higher than PCI-e 3.0 and 2.4–3.2× higher than the CPU baseline for in-GPU hash tables. The throughput degrades above a hash table size of 8 GiB due to TLB misses (see Section 4.2.4). Although throughput declines by 85% for out-of-core hash tables, performance remains 12–20× higher than PCI-e 3.0. Although NVLink 2.0 is slower than the CPU baseline for the out-of-core hash table, NVLink 2.0 remains within 23% of the CPU.

NVLink 2.0 with Hybrid Hash Table. We notice that storing parts of the hash table in GPU memory achieves a speedup of 1–3× over only NVLink 2.0, despite facing a uniform foreign key distribution. We summarize that NVLink 2.0 helps to achieve higher out-of-core throughput than PCI-e 3.0, and that throughput degrades gracefully, instead of riding over a performance cliff when the hash table is larger than the GPU's memory capacity.

Build-to-probe Ratios

In Figure 3.17, we quantify the impact of different build-to-probe ratios on join throughput. We use workload C with 16-byte tuples, and increase S with a $|R|$-to-$|S|$ ratio from 1:1 up to 1:16 (up to 2 GiB $\bowtie$ 30.5 GiB). We store the base relations in CPU memory, and the hash table in GPU memory.

Observations. The build phase takes 70% of the time, and is thus 56% slower than

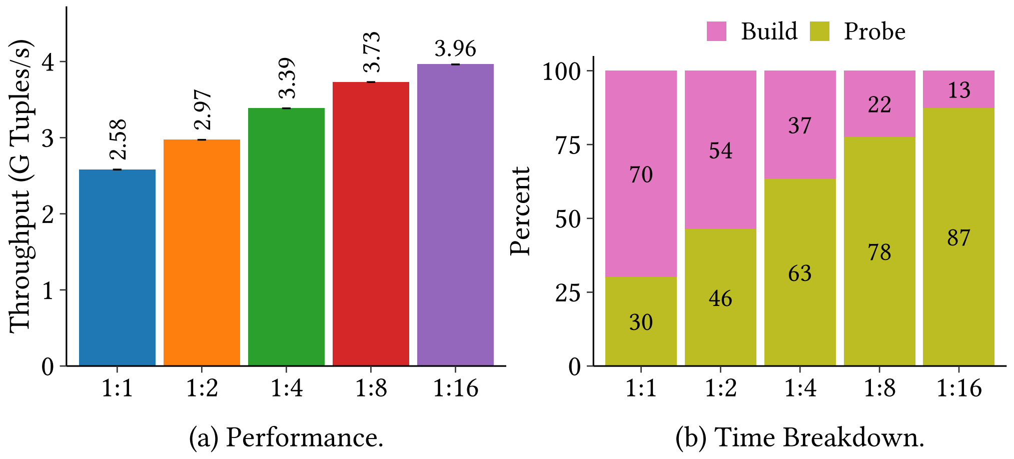


Figure 3.17: Different build-to-probe ratios on NVLink.

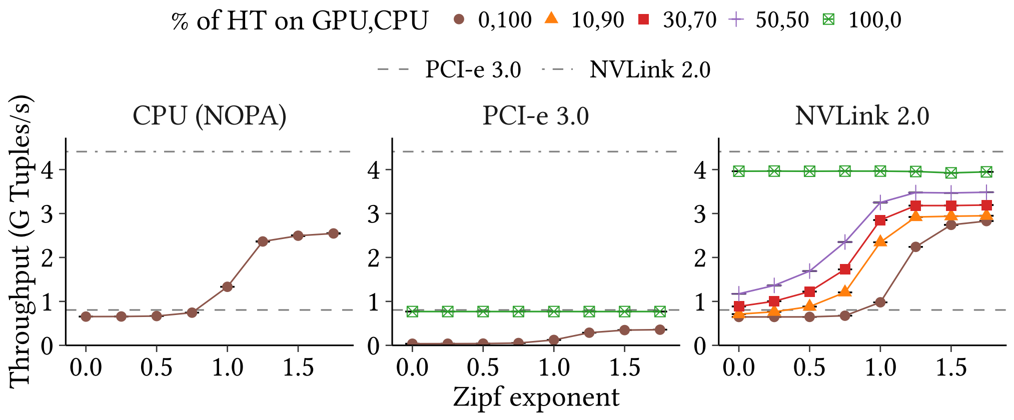


Figure 3.18: Join performance when the probe relation follows a Zipf distribution.

the probe phase. The impact is most visible for 1:1 ratios. For larger ratios, the build-side takes up a smaller proportion of time, which makes the join faster. We are able to observe these differences because NVLink 2.0 eliminates the transfer bottleneck for this use-case.

Data Skew

We explore a join on data skewed with a Zipf distribution in Figure 3.18. We use workload A (34 GiB), but skew S with Zipf exponents between 0–1.75. With an exponent of 1.5, there is a 97.5% chance of hitting one of the top-1000 tuples. Thus, increasing data skew tends to increase cache hits. To show the effect of caching, we place the hash table in CPU memory, in GPU memory, and in a hybrid hash table with a varying CPU-to-GPU

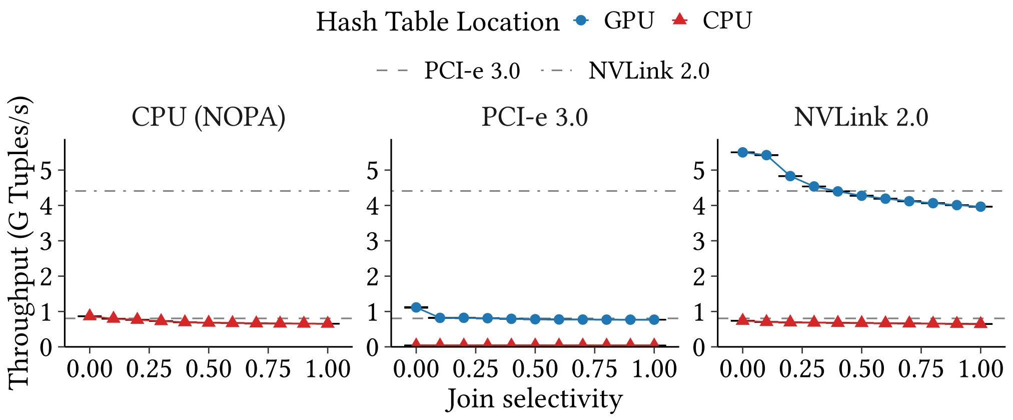


Figure 3.19: The effect of join selectivity on throughput.

memory ratio.

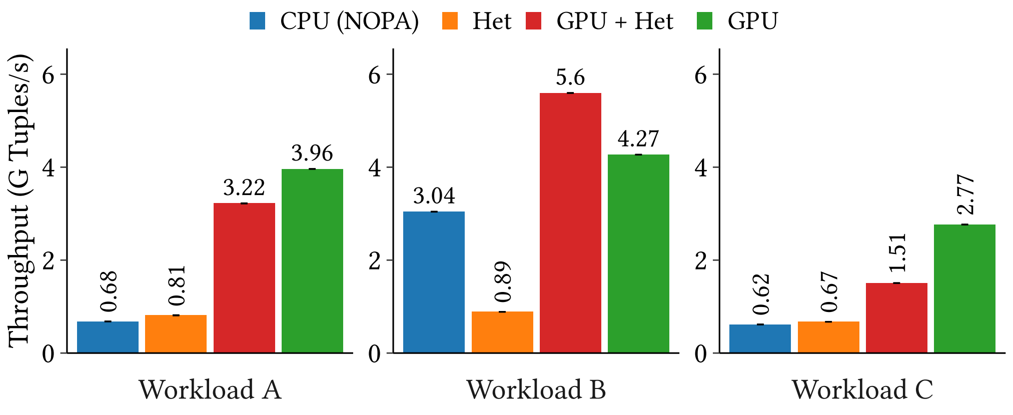
Observations. We observe that higher skew leads to a higher throughput of 3.9×, 4.4×, and 9.4× for the CPU, NVLink 2.0, and PCI-e 3.0, respectively. This effect is not present for hash tables in GPU memory, as transferring the base relations from CPU memory is the bottleneck. Thus, we see throughput increase with the hybrid hash table.

Join Selectivity

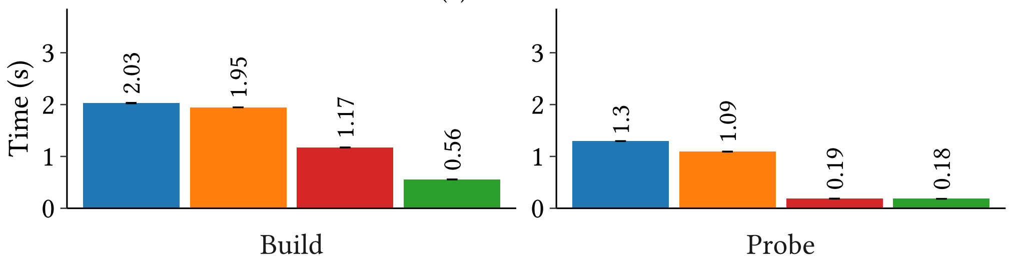
We evaluate the effect of join selectivity on join throughput in Figure 3.19. We vary the selectivity of Workload A (34 GiB) from 0–100% by randomly selecting a subset of *R*. We show the performance of in-GPU and out-of-core hash table placement, and compare the GPU against an IBM POWER9 CPU running the same NOPA join variant.

Observations. Our measurements show that join throughput decreases with higher selectivity. The decrease is largest at 28% for NVLink 2.0 with a GPU memory hash table. In contrast, PCI-e 3.0 slows down by only 4% with a hash table in CPU memory. We notice that both interconnects achieve throughput higher than the calculated bandwidth would suggest. This is because only the join key is necessary to establish a match. If there is no match, the value is not accessed. However, if there are one or more matches, the whole cache line is loaded. In effect, at 10% selectivity, 81.5% of values are loaded, resulting in a throughput drop.

Overall, NVLink 2.0 with an out-of-core hash table achieves similar performance to the CPU, and 6.1–7.4 × better performance with a GPU-local hash table.



(a) Performance.



(b) Time per join phase in workload C.

Figure 3.20: Cooperative CPU and GPU join. *Het* uses a shared hash table in CPU memory, whereas *GPU* + *Het* uses private hash tables in processor-local memory.

CPU/GPU Co-processing Scale-up

In Figure 3.20(a), we show the join throughput when scaling up the join to a CPU and a GPU using the cooperative Het and GPU + Het strategies described in Section 3.5.2. We use the workloads in Table 3.2, that have a size up to 34 GiB. We drill down into the individual join phases of workload C in Figure 3.20(b) to gain more insights. As the CPU we use an IBM POWER9, and execute the same NOPA algorithm that we run on the GPU. We store the hash table in CPU memory for the CPU and Het execution strategies, and in GPU memory for the GPU and GPU + Het strategy. In GPU + Het, we copy the hash table to CPU memory for the probe phase. The Het and GPU + Het strategies use 32 MiB CPU and 128 MiB GPU morsels, whereas data are statically assigned to threads in the CPU-only and GPU-only strategies. We store the base relations in CPU memory for all strategies.

Workload A. We observe that the Het, Het + GPU, and GPU execution strategies run faster than the CPU strategy by 19%, 4.7×, and 5.8×, respectively. Adding a GPU always increases throughput, and the GPU without the CPU achieves the highest throughput. The GPU-only strategy is faster than both heterogeneous strategies.

Workload B. We see that Het is 70% slower than CPU-only, whereas Het + GPU and GPU achieve speed-ups of 1.8× and 1.4×. As the CPU is able to cache the small hash table, storing the hash table in GPU memory is necessary for the GPU to increase throughput over the CPU. The cooperative GPU + Het strategy outperforms the GPU-only strategy by 31%.

Workload C. We notice that all strategies which use a GPU achieve higher throughput than the CPU-only strategy: Het by 8%, GPU + Het by 2.4×, and GPU by 4.6×.

Time per Join Phase in Workload C. To understand why the GPU-only strategy often outperforms the heterogeneous strategies, we investigate the join phases individually.

In both phases, we notice that adding a GPU to the CPU increases performance, but that the GPU by itself is fastest. We observe that a processor-local hash table increases throughput (Het vs. GPU + Het), and that transitioning from a CPU-only to a CPU/GPU solution (Het and GPU + Het) decreases processing time.

Overall, using a GPU achieves the same or better throughput than the CPU-only strategy for most of our workloads. However, caching the hash table in GPU memory results in the best performance.

3.7 Discussion

In this section, we discuss the key insights that we obtained from our fast interconnects characterization (Section 3.2) and data processing evaluation (Section 3.6).

(1) GPUs have high-bandwidth access to CPU memory. We observed that GPUs can load data from CPU memory with bandwidth similar to the CPU. Thus, offloading data processing on GPUs becomes viable even when the data is stored in CPU memory.

(2) GPUs can efficiently process large, out-of-core data. A direct consequence of (1) is that transfer is no longer a bottleneck for complex operators. We have shown speedups of up to 6× over PCI-e 3.0 for hash joins operating on a data structure in GPU memory. In these cases, performance is limited by other factors, e.g., computation or GPU memory.

(3) GPUs are able to operate on out-of-core data structures, but should use GPU memory if possible. In our evaluation, we showed up to 20× higher throughput with NVLink 2.0 than with PCI-e 3.0. Despite this speedup, operating within GPU memory is still 6.5× faster compared to transferring data over NVLink 2.0. However, for hash tables up to 1.8× larger than GPU memory, we achieved competitive or better performance than an optimized CPU radix join by caching parts of the hash table in GPU memory.

(4) Scaling-up co-processors with CPU + GPU makes performance more robust. A cache-coherent interconnect enables processors to work together efficiently. Processors that cooperate avoid worst-case performance, thus making the overall performance more robust.

(5) Due to cache-coherence, memory pinning is no longer necessary to achieve high transfer bandwidth. We evaluated eight transfer methods, and discovered that fast interconnects enable convenient access to pageable memory without any performance penalty. The benefit is that memory management becomes much simpler because we no longer need staging areas in pinned memory.

(6) Fair performance comparisons between GPUs vs. CPUs have become practical. As a final point, in this chapter, we have studied the performance of a GPU and a CPU that load data from the same location (i.e., CPU memory). Fast interconnects enabled us to observe speedups without caching data in GPU memory, although the CPU remains faster in some cases.

Summary. With fast interconnects, GPU acceleration becomes an attractive scale-up alternative that promises large speedups for DBMSs.

3.8 Related Work

We contrast our contributions to related work in this section.

Transfer Bottleneck. The realization that growing data sets do not fit into the co-processor's memory [161, 424] has led recent works to take data transfer costs into account. GPU-enabled DBMSs such as GDB [170], Ocelot [176], CoGaDB/Hawk/HorseQC [78, 79, 146], and HAPE [102, 103], as well as accelerated machine learning frameworks such as SystemML [40] and DAnA [264], are all capable of streaming data from CPU memory onto the co-processor. HippogriffDB [250] and Karnagel et al. [219] take out-of-core processing one step further by loading data from SSDs. The effect of data transfers has also been researched for individual relational operators on GPUs [159, 213, 219, 259, 260, 357, 385, 395]. All of these works observe that transferring data over a PCI-e interconnect is a significant bottleneck when processing data out-of-core. In this chapter, we investigate how out-of-core data processing can be accelerated using a faster interconnect.

Transfer Optimization. The success of previous attempts to resolve the transfer bottleneck in software heavily depends on the data and query. Caching data in the co-processor's memory [78, 176, 218] assumes that data are reused, and is most effective for small data sets or skewed access distributions. Data compression schemes [136, 359] must match the data to be effective [115, 131], and trade off computation vs. transfer time. Approximate-and-refine [340] and join pruning using Bloom filters [163] depend on the query's selectivity, and process most of the query pipeline on the CPU. In contrast to these approaches, we show that fast, cache-coherent interconnects enable new acceleration opportunities by improving bandwidth, latency, as well as synchronization cost.

Transfer Avoidance. Another approach is to avoid the transfer bottleneck altogether by using a hybrid CPU-GPU or CPU-FPGA architecture [173, 174, 214, 255, 318]. Hybrid architectures integrate the CPU cores and accelerator into a single chip or package, whereby the accelerator has direct access to CPU memory over the on-chip interconnect [76, 165]. In contrast to these works, we consider systems with discrete GPUs, because discrete co-processors provide more computational power and feature high-bandwidth, on-board memory.

Out-of-core GPU Data Structures. Hash tables [74, 223], B-trees [43, 212, 372, 421], log-structured merge trees [41], and binary trees [225] have been proposed to efficiently access data using GPUs. In contrast, we investigate hash tables in the data management context. We demonstrate concurrent CPU and GPU writes to a shared data structure, and perform locality optimizations. In addition, our approach is more space-efficient than previous shared hash tables [74].

Fast Interconnects. NVLink 1.0 and 2.0 have been investigated previously in microbenchmarks [208, 248, 249, 328, 329] and for deep learning [230, 399, 419]. In contrast to these works, we investigate fast interconnects in the data management context. To the best of our knowledge, we are the first to evaluate CPU memory latency and random CPU memory accesses via NVLink 1.0 or 2.0. Raza et al. [349] study lazy transfers and scan sharing for HTAP with NVLink 2.0. In contrast, we conduct an in-depth analysis of fast interconnects.

3.9 Conclusion

We conclude that, on the one hand, fast interconnects enable new use-cases that were previously not worthwhile to accelerate on GPUs. On the other hand, currently NVLink 2.0 represents a specialized technology that has yet to arrive in commodity hardware. Overall, in this chapter we have made the case that future DBMS research should consider fast interconnects for accelerating workloads on co-processors.

4 Scalable and Robust Stateful Data Processing

GPUs cannot scale joins to large data volumes due to two limiting factors: (1) large state does not fit into the GPU memory, and (2) spilling state to CPU memory is constrained by the interconnect bandwidth. In this chapter, we propose a new join algorithm that scales to large data volumes by taking advantage of fast interconnects. Fast interconnects such as NVLink 2.0 connect the GPU to CPU memory at a high bandwidth, and thus enable us to design our join to efficiently spill its state. As a result, GPU-enabled DBMSs are able to scale the join state beyond the GPU memory capacity.

4.1 Introduction

GPU-enabled DBMSs obtain a performance advantage from GPU join and group-by aggregation queries with an in-GPU state [78, 146, 161, 170, 374, 424]. However, based on our insights obtained in Chapter 3, GPUs cannot efficiently scale to a large, out-of-core state due to the data transfer bottleneck. This hardware limitation leads to a narrow scope where DBMSs benefit from GPUs.

Nevertheless, as we illustrate in Figure 4.1, higher interconnect bandwidth is necessary, but not sufficient for high scalability. Even if the GPU is given a faster interconnect, the CPU outperforms the GPU when joining two large data sets. Therefore, we identify three fundamental challenges that need to be addressed to widen the applicability of GPUs:

Scalability. GPU joins store their state in GPU memory to increase throughput [171, 213, 288, 321, 374]. Due to the limited capacity of GPU memory, GPU joins cannot

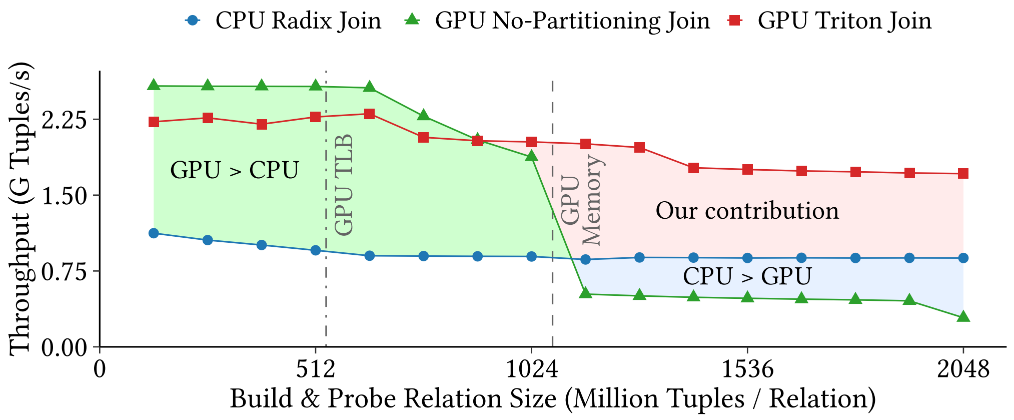


Figure 4.1: Out-of-core state results in a performance cliff and a slow-down, despite using a fast interconnect. In contrast, our Triton join gracefully scales to joins with a large state.

efficiently scale to a large state [261]. In contrast, CPUs [23, 200, 393] have two orders-of-magnitude higher memory capacity than GPUs [22, 297, 302]. Thus, we must adapt GPU joins to spill their state to CPU memory in order to achieve scalability.

Robustness. Spilling the join state to CPU memory results in a performance cliff [261]. These sharp performance drops are difficult to account for in query optimizers, because cardinality estimates can be significantly wrong [101, 270]. Thus, GPU-enabled DBMSs must gracefully scale to large data sizes for a consistent user experience.

Efficiency. State-of-the-art approaches reduce interconnect transfers by shifting computations from the GPU to the CPU [159, 163, 340, 385, 395]. However, both interconnect bandwidth and CPU cycles are scarce resources. DBMSs should use the GPU to offload computations from the CPU, while maximizing performance.

Fast interconnects have the potential to help us address the above challenges and improve join throughput by providing high-bandwidth, cache-coherent access to CPU memory. Our goal is to enable GPUs to process joins with a state that exceeds the GPU memory capacity. Thus, we consider joins smaller and larger than the GPU memory. For large joins, we partition data out-of-core in CPU memory using the fast interconnect to achieve data locality during the join. In contrast, small joins require us to cache all intermediate results in GPU memory to avoid unnecessary data transfers. We combine the GPU-based partitioning and the caching in our new, hierarchical hybrid hash join algorithm: $^3H^+$ ≡ the *Triton join*.

Overall, our contributions are as follows:

1. We investigate the requirements of an out-of-core GPU join in regard to fast

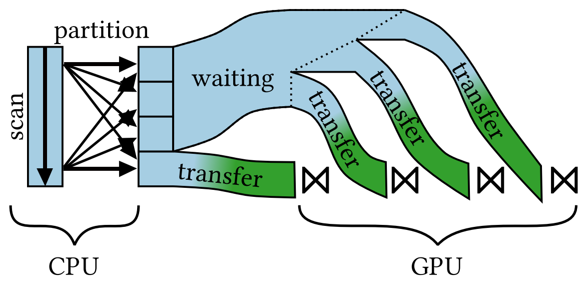


Figure 4.2: The CPU-partitioned join strategy splits state into small partitions before starting to process data on the GPU.

interconnects, and identify hardware bottlenecks that limit scalability (Section 4.2).

2. We propose a new GPU radix partitioning algorithm that takes advantage of fast interconnects to achieve a high bandwidth and scale to large data volumes (Section 4.3).

3. We present our new Triton join algorithm, a scalable radix-partitioned GPU hash join that partitions data using the GPU and caches partitioned data in GPU memory (Section 4.4).

The further structure of this chapter is as follows. We motivate our approach by revisiting out-of-core GPU joins in Section 4.2. After that, we demonstrate our out-of-core radix partitioning approach in Section 4.3, and then overcome these challenges with our Triton join in Section 4.4. In Section 4.5, we show our evaluation and discuss our insights. Finally, we review related work in Section 4.6 and conclude in Section 4.7.

4.2 Revisiting Out-of-Core GPU Joins

Existing out-of-core join algorithms assume to varying degrees that interconnect bandwidth is the bottleneck, which fundamentally shapes the design strategy underpinning the algorithm. In this section, we examine how fast interconnects change this assumption, and study the impact of the interconnect on the join strategy.

We first discuss the CPU-partitioned join strategy (Section 4.2.1), as we find it an enlightening point in the design space due to its focus on the transfer volume. Then, we show that fast interconnects enable the GPU to process out-of-core data (Section 4.2.2). Based on this insight, we argue that fast interconnects open the door for a new high-level design, the GPU-partitioned join strategy (Section 4.2.3). Finally, we analyze the hardware capabilities and limitations to inform our detailed design choices (Section 4.2.4).

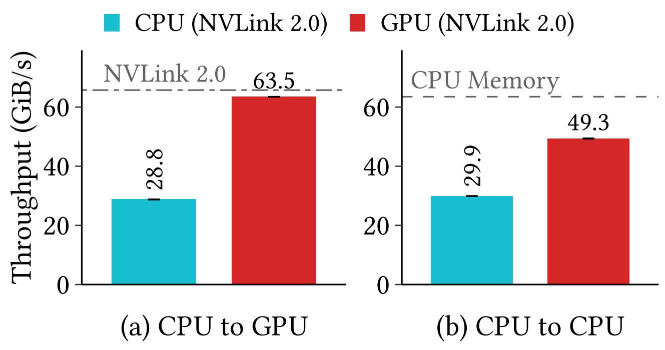


Figure 4.3: Data partitioning throughput of a CPU and a GPU for different source and destination locations.

4.2.1 The CPU-Partitioned Join Strategy

A recent *CPU-partitioned join strategy* proposes to partition the data on the CPU before transferring it to the GPU [385]. The goals are to minimize data transfers across the interconnect, and to access the join's state efficiently in GPU memory.

We outline this join strategy in Figure 4.2. It consists of three phases. First, the CPU partitions the data into working sets that individually fit into GPU memory. Then, the strategy transfers a working set to the GPU. Third, the GPU joins the relations within the working set. Steps two and three are repeatedly executed in a pipeline to hide the transfer latency. Although the partitioning and transfer may overlap, at least one relation must be completely partitioned before starting the join.

The CPU can initially transfer only a fraction of the data to the GPU, as only one working set completely fits into GPU memory at a time. Let this fraction be $\alpha := \frac{|\text{working set}|}{|\text{data}|}$. To saturate the interconnect bandwidth, the CPU must partition at a rate higher than $^1/_\alpha \times$ [transfer bandwidth]. For example, with a 12 GiB/s transfer rate and $\alpha = {}^1/_4$, the CPU must partition at $4 \cdot 12$ GiB/s = 48 GiB/s. However, the partitioning throughput must increase to 260 GiB/s in order to saturate a 65 GiB/s fast interconnect.

We argue that such a partitioning rate is unrealistically fast, as it would exceed the CPU memory bandwidth even when using multiple CPUs. As a result, the CPU-partitioned strategy underutilizes the GPU and the fast interconnect.

4.2.2 Fast Interconnects Outpace CPUs

Fast interconnects provide a new opportunity to utilize hardware resources efficiently by computing all join phases on the GPU. We show that if the join is optimized for a fast interconnect, then the GPU is able to outperform a CPU even for this data-intensive task.

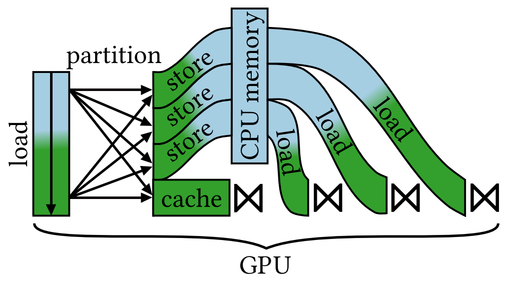


Figure 4.4: The GPU-partitioned join strategy processes both the partition and join phases on the GPU, spilling state to CPU memory if necessary.

We demonstrate our insight in Figure 4.3. We measure the partitioning throughput of a CPU and a GPU, and distinguish between the two extreme cases: (a) either all resulting partitions fit into GPU memory or (b) all partitions are stored to CPU memory. Both processors read the base relation from CPU memory, and split the data into 512 partitions. We observe that in both cases the GPU is faster than the CPU. Conversely, despite transferring all partitions at once ($\alpha = 1$), the CPU cannot saturate the fast interconnect.

Our take-away is that fast interconnects require a new approach for GPU joins to take full advantage of the hardware. The existing CPU-partitioned strategy underutilizes the GPU *and* the fast interconnect. Thus, a GPU-centric approach would be able to utilize the available hardware resources better.

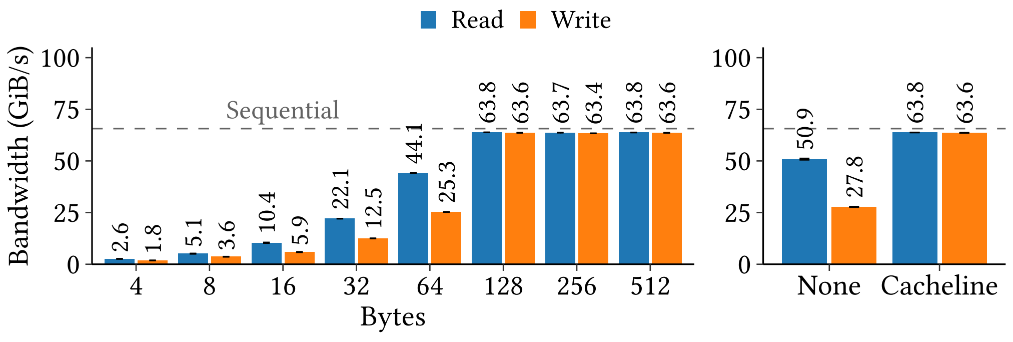
4.2.3 The GPU-Partitioned Join Strategy

Our goal is to compute the join end-to-end on the GPU. For this reason, we propose a new, *GPU-partitioned join strategy* that is optimized for GPUs with fast interconnects.

We highlight our GPU-partitioned join strategy in Figure 4.4. Our strategy works as follows. In the partitioning phase, the GPU loads the data from CPU memory, and caches the resulting partitions in GPU memory. If this state exceeds the GPU memory capacity, the GPU spills the remainder to CPU memory. In the join phase, the GPU loads the spilled state from CPU memory again.

We overlap transfers and computations using two methods. For phases that consist of a single GPU kernel, we rely on the hardware cache-coherence [261]. In contrast, for phases consisting of multiple kernels, we describe a new transfer method in Section 4.4.2.

Overall, the advantages of our strategy are that (1) computation is offloaded to the GPU and that (2) the join gracefully scales to out-of-core state. The trade-off is a 1–2× higher transfer volume, depending on how many data are cached vs. spilled to CPU



(a) Access Granularity (b) Alignment

Figure 4.5: GPU interconnect bandwidth of a random access pattern to CPU memory with varying access granularities.

memory.

4.2.4 Capabilities of Fast Interconnects

To efficiently implement our GPU-partitioned join strategy in practice, we require an in-depth understanding of the interconnect hardware. Crucially, if data is spilled during the partitioning phase, the GPU performs random writes to CPU memory [267]. Thus, we analyze the key metrics for random accesses: the interconnect bandwidth of fine-grained memory accesses, and the TLB miss latency.

Efficient Transfers with Fine Granularity.

Ideally, a join running on the GPU achieves the full interconnect bandwidth when accessing CPU memory. However, the bandwidth achieved in practice depends on the access granularity, as the GPU executes memory accesses in units of *memory transactions* [304]. Memory transactions have a hardware-specific size. If accesses are fine-grained, i.e., smaller than the memory transaction size, then each memory transaction only carries a partial payload. This leads to a reduced bandwidth utilization. Although memory transactions in GPU memory have been researched [204, 221, 386], prior work does not consider the effect on the interconnect bandwidth.

Setup. We experimentally determine the minimum required memory access granularity to achieve the full interconnect bandwidth in Figure 4.5(a). In the experiment, the GPU randomly accesses CPU memory on the nearest NUMA node. We first scale the access granularity from 4–16 bytes by increasing the integer type from 32–128 bits. Then, we continue to scale by coalescing 2–32 threads (i.e., up to a warp) for 32–512-byte accesses. We measure read and write accesses within a 1 GiB array, and efficiently generate the random access pattern via a linear congruential generator [229]. All accesses are aligned according to their granularity, i.e., a 512-byte access is aligned to 512 bytes.

Results. In the measurement, we observe that the interconnect bandwidth grows linearly with the access granularity. Small reads up to 64 bytes are 44–74% faster than writes. At 128 bytes, the bandwidth of random accesses equals the bandwidth of a coalesced sequential access pattern.

Furthermore, in Figure 4.5(b), we determine that misaligned accesses reduce the achieved interconnect bandwidth. We measure that misaligning a 512-byte memory access by 16 bytes reduces the bandwidth by 20% for reads and 56% for writes.

Analysis. From our results, we deduce that "Volta" GPUs coalesce CPU memory accesses via NVLink 2.0 into 128-byte memory transactions (or larger) instead of 32 bytes in GPU memory [221, 388]. These transactions are aligned to 128-byte cachelines. Our analysis is substantiated by vendor documentation on NVLink 2.0 [191, 304] and an investigation of PCI-e [281]. However, it remains unclear why small reads outperform small writes.

Our findings differ from GPU literature, which suggests that GPU random accesses to CPU memory are slower that sequential transfers [261], and that GPU programmers should coalesce memory accesses of warps with natural alignment on the data type [20, 304].

Overall, if accesses are *perfectly coalesced* as described above, GPUs are able to achieve the full interconnect bandwidth for random CPU memory accesses at a 128-byte granularity.

TLB Miss Cost with Fast Interconnects.

Fast interconnects give GPUs high-bandwidth access to terabytes of data in CPU memory. Due to the large data size, a join randomly accesses thousands of memory pages. As a result, virtual to physical memory address translations impact join throughput. We quantify the address translation cost for GPUs, and discover that TLB misses when using a fast interconnect are up to an order-of-magnitude more expensive than TLB misses in GPU memory.

Setup. In Figure 4.6, we compare the TLB miss costs of GPU accesses to GPU memory and to CPU memory. We measure the latency of individual memory accesses with fine-grained pointer chasing [273]. We perform 16, 32, and 64 MiB strides in a memory range of 6–10.7 GiB in GPU memory and a range of 1–87.5 GiB in CPU memory. We

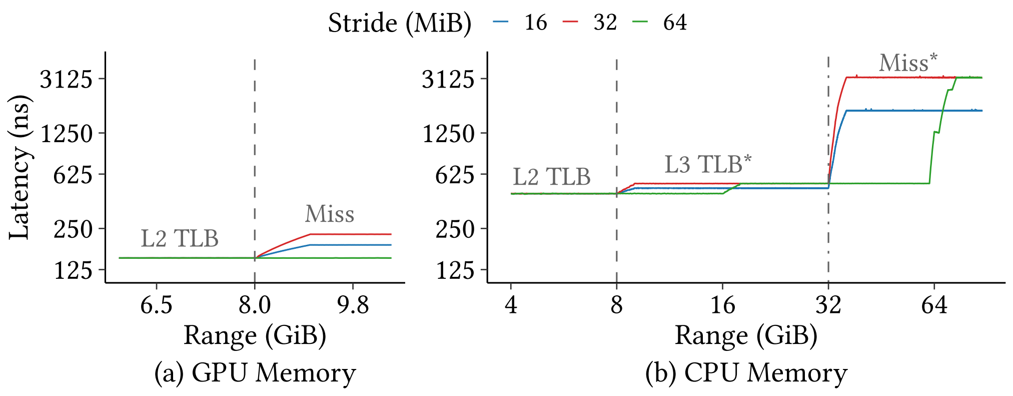


Figure 4.6: TLB miss latency for GPU memory, and for CPU memory via NVLink 2.0.

allocate 2 MiB huge pages in CPU memory on the NUMA node closest to the GPU. To avoid page fragmentation, we preallocate huge pages early at boot time [285]. To prevent the hardware from caching translations across runs, we flush the IOTLB before each run by calling the `mprotect` system call [157]. We observe that the GPU TLBs are flushed by the CUDA runtime before each kernel launch. As the L1 data cache is virtually tagged and thus does not incur address translations [208], we bypass the L1 cache with the `cg` PTX cache hint [307].

Results. In GPU memory, we observe that the GPU L2 TLB covers 8 GiB. We measure a L2 TLB hit latency of 151.9 ± 4.8 ns and a miss latency of 226.7 ± 4.8 ns. Our measurements match the results of Jia et al. [208], who state that "Volta" GPUs have a L1 TLB in addition to the L2 TLB.

In CPU memory, the L2 TLB also covers 8 GiB with a hit latency of 449.7 ± 32.4 ns. Beyond the L2 TLB, we notice two miss plateaus, one at 9.5–32 GiB and another above 37 GiB. For the first, we measure a latency of 532.9 ± 45.8 ns, and 3186.4 ± 154.0 ns for the second. We speculatively name the plateaus *L3 TLB** and *Miss**.

Analysis. We observe that the L2 TLB page size is 32 MiB not only in GPU memory [208, 217], but also in CPU memory. Thus, 16 physically adjacent 2 MiB pages are likely coalesced on a page table walk [121, 122, 191, 217, 335].

However, we lack evidence to fully explain the TLB misses that occur in CPU memory. The high miss latency (Miss*) indicates a GPU TLB miss, that results in an IOTLB or IOMMU lookup. In contrast, the L2 TLB miss penalty to the L3 TLB* is only 83 ns. On the one hand, this is likely too short to traverse the interconnect. On the other hand, the L2 TLB miss penalty in GPU memory is similar at 75 ns. As NVLink 2.0 enables

Table 4.1: Partitioning design goals.

Algorithm	Space Efficient	Perfect Coalescing	High Fanout
SWWC	✗	✗	✗
Linear	✓	✗	✗
Shared	✓	✓	✗
Hierarchical	✓	✓	✓

a system-wide page table [191], we assume that the GPU does not duplicate the table in GPU memory. Thus, our results might indicate that another translation caching layer exists [54, 59, 205], that is distinct from the IOTLB. However, we leave a deeper investigation to future work.[1]

In conclusion, TLB misses are a hard problem to mitigate for out-of-core algorithms. However, we find that if an algorithm carefully manages its TLB misses and access granularity, then the GPU can achieve a high interconnect bandwidth even for random accesses.

4.3 Efficiently Partitioning Data over a Fast Interconnect

In order to join large data efficiently, the GPU first needs to partition the data out-of-core. We transform our hardware insights from Section 4.2.4 into concrete design goals (Section 4.3.1), on which we base two new radix partitioning algorithms for GPUs. First, we increase the interconnect utilization of random writes in our *shared software write-combining (Shared)* algorithm (Section 4.3.2). In a next step, we reduce GPU TLB misses for high fanouts in our *hierarchical software write-combining (Hierarchical)* algorithm (Section 4.3.3).

4.3.1 Design Goals

Achieving high partitioning throughput requires us to consider both the GPU architecture and the fast interconnect. To this end we formulate three design goals. First, the algorithm should be space efficient, due to the small scratchpad capacity. As a thread block shares the scratchpad, all accesses to the scratchpad must be thread-safe. Second, random memory accesses over the interconnect should adhere to perfect coalescing (see Section 4.2.4). Finally, large data sets require a high fanout to reduce the size of each partition. This incurs TLB misses, which should be avoided (see Section 4.2.4).

State-of-the-art algorithms do not achieve these goals, as we summarize in Table 4.1.

[1]According to Nvidia, this information is currently not publicly available.

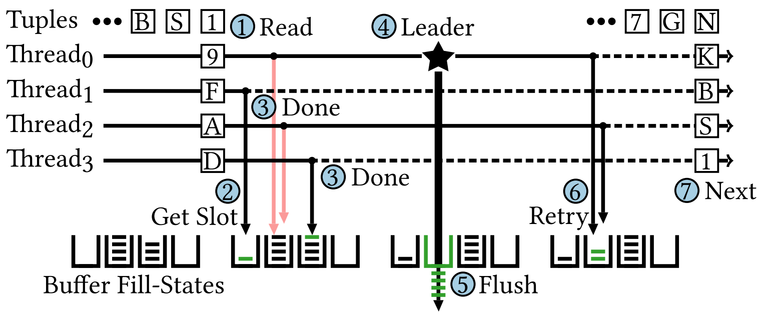


Figure 4.7: A thread block shares scratchpad buffers, and flushing is coalesced. Shown is one warp with four threads.

The SWWC algorithm allocates thread-private buffers, as CPUs have large caches. Linear is designed for in-GPU partitioning, and opportunistically coalesces writes by sorting batches of tuples. Thus, we devise a new partitioning approach optimized for out-of-core partitioning.

4.3.2 Shared: High-Throughput Partitioning

We design our *shared software write-combining (Shared)* algorithm for space-efficiency and perfect coalescing. We first provide an overview of Shared, and then examine the buffer and flush phases in detail. Finally, we discuss how Shared achieves our design goals.

Description. In Figure 4.7, we show the execution flow of our algorithm in seven steps. On a high level, Steps 1–3 make up the *fill phase*, and Steps 4–6 constitute the *flush phase*. Step 7 begins a new fill phase. Before execution begins, the input is divided into equally-sized chunks which are assigned to thread blocks.

Fill Phase. We display the fill phase in Listing 4.1. Execution proceeds in warps. In Step ①, each thread reads a tuple into a register and hashes the key. Then (②), each thread tries to acquire a free slot in the buffer indicated by the hash. Threads acquire slots atomically, as the buffers are shared among all warps. If a thread successfully acquires an empty slot ($\longrightarrow$) in Step ③, the thread stores its tuple into the buffer and marks itself "done". If all threads in a warp are "done", the warp proceeds to the next fill phase. Else, if any thread encountered a full buffer ($\longrightarrow$), the warp proceeds to the flush phase.

Flush Phase. In Listing 4.2, we specify the flush phase. The flush phase begins with Step ④. All active threads (i.e., not "done") of the warp participate in a leader ballot, and elect a thread as the warp leader. We define the first invalid slot (i.e., the buffer size) as a lock on the buffer. Thus, the full buffers are locked since Step two. All active threads except the leader immediately release their lock to enable parallel flushes by other warps.

```
// Fill tuples into buffers
for (i = threadIdx.x; i < tuples; i += blockDim.x) {
  // Step ①: Read a tuple and hash its key
  Tuple tuple = { keys[i], values[i] };
  p_index = hash(tuple.key) % FANOUT;

  slot = 0; done = false;
  do {
    // Try buffering a tuple
    if (not done) {
      // Step ②: Acquire a free slot
      slot = atomicAdd(&slots[p_index], 1);

      if (slot < BUFFER_SIZE) {
        // Step ③: Store tuple into buffer
        buffers[p_index][slot] = tuple;
        __threadfence_block(); // Wait for write
        atomicAdd(&fillstate[p_index], 1); // Mark written
        done = true;
      }
    }
    [..] // Flush phase, see Listing 4.2

  // Step ⑥: Repeat flush phase until each thread has buffered its tuple
  } while (__any_sync(WARP_MASK, not done));

} // Step ⑦: Start a new fill phase

[..] // Flush all buffers
```

Listing 4.1: Fill phase of Shared partitioning algorithm.

```
// Flush a full buffer

// Step ④: Elect a warp leader
is_candidate = (slot == BUFFER_SIZE); // Implicit lock
do_flush = __ballot_sync(WARP_MASK, is_candidate);

// Step ⑤: Flush leader's buffer
if (do_flush) {
  leader_id = __ffs(ballot) - 1;

  // Unlock, maybe another warp will flush
  if (is_candidate && leader_id != LANE_ID) {
    slots[p_index] = BUFFER_SIZE;
  }
  __syncwarp(); // Wait for unlock

  if (leader_id == LANE_ID) {
    // Wait until buffer is full
    while (fillstate[p_index] != BUFFER_SIZE);
    fillstate[p_index] = 0;
  }
  __syncwarp(); // Warp waits until leader is ready

  // Get leader's flush parameters
  leader_index = __shfl_sync(WARP_MASK, p_index, leader_id);
  dst = &p_output[p_offsets[leader_index]];
  src = &buffers[leader_index][0];

  // Flush buffer
  warp_memcpy(dst, src, BUFFER_SIZE);

  if (leader_id == LANE_ID) {
    // Update memory offset of partition
    p_offsets[leader_index] += BUFFER_SIZE;
    __threadfence_block();

    // Unlock
    slots[leader_index] = 0;
  }
}
// See Listing 4.1 for remaining steps
```

Listing 4.2: Flush phase of Shared partitioning algorithm.

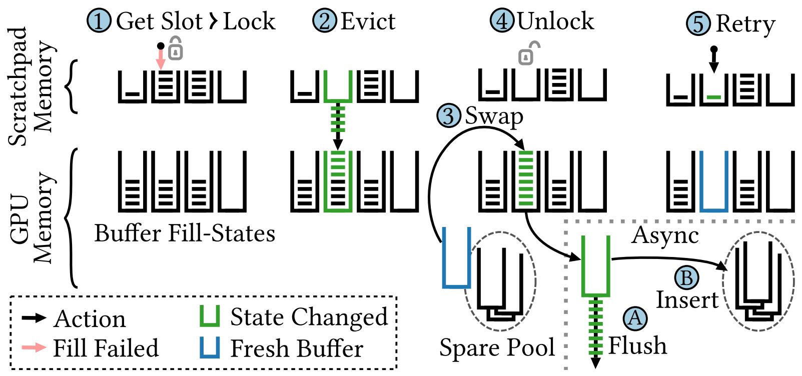


Figure 4.8: Buffer tuples in a two-level SWWC hierarchy for high fanouts. The 2nd level provides space for more buffers.

Next (⑤), the warp flushes the leader's buffer. Then, in Step ⑥, the active threads retry acquiring a slot. If at least one thread fails to acquire a slot, the warp repeats the flush phase until all threads have buffered their tuple and are marked "done". Finally, all threads start a new fill phase in Step ⑦.

Design Discussion. In our design, two aspects are important to efficiently share buffers and perfectly coalesce writes. First, filling the buffers is thread-safe but lock-free. Only flushing a buffer requires a lock, which we assign to a warp instead of spinning on the lock. Second, each flush is a multiple of the memory transaction size and also aligned to the transaction size. This ensures that optimally-sized writes are not split into two memory transactions.

4.3.3 Hierarchical: High-Fanout Partitioning

To reduce expensive GPU TLB misses (see Section 4.2.4), we introduce a new *hierarchical shared software write-combine (Hierarchical)* algorithm. Hierarchical extends the SWWC buffers in scratchpad memory with a second-level cache in GPU memory. By adding buffer capacity, Hierarchical incurs less TLB misses when writing to CPU memory, and we are thus able to increase the fanout.

Description. We derive the Hierarchical algorithm from Shared by extending the flush phase into a two-level hierarchy in Figure 4.8. The fill phase remains unchanged. The new flush phase consists of seven steps, that are executed by a warp. We provide a detailed description in Listing 4.3.

L1 Eviction. The flush begins when the warp encounters a full buffer and obtains a lock on that buffer in Step ①. The lock is enforced by the fill-state counter when the buffer is full. In Step ②, the warp evicts all tuples from the L1 buffer to its corresponding

```
// Flush a full buffer with buffer hierarchy

// Step ①: Obtain lock on buffer
int is_candidate = (pos == L1_BUFFER_SIZE);
if (__ballot_sync(WARP_MASK, is_candidate)) {
  [..] // Same as in Shared algorithm, see Listing 4.2

  // Get leader's L2 slot and active L2 buffer
  l2_slot = l2_slots[leader_index];
  active_l2_buffer = l2_buffer_map[leader_index];

  // Step ②: Evict tuples from L1 buffer to L2 buffer
  dst = &l2_buffers[active_l2_buffer][l2_slot];
  src = &buffers[leader_index][0];
  warp_memcpy(dst, src, L1_BUFFER_SIZE);
  __syncwarp(); // Warp waits until warp_memcpy is finished

  // Update L2 slot and check if need to flush L2 buffer
  do_l2_flush = (++l2_slot == slots_per_l2_buffer);

  // Step ③: Swap in an empty L2 buffer
  if (do_l2_flush) {
    l2_slot = 0;
    if (leader_id == LANE_ID) {
      l2_buffer_map[leader_index] = spare_pool;
      dst = &p_output[p_offsets[leader_index]];
      p_offsets[leader_index] += L2_BUFFER_SIZE;
   }
   __syncwarp(); // Warp waits until leader has swapped buffer
  }

  // Step ④: Unlock L1 buffer
  if (leader_id == LANE_ID) {
    l2_slots[leader_index] = l2_slot;
    __threadfence_block();
    slots[leader_index] = 0;
  }

  // Step Ⓐ: Asynchronously flush L2 buffer
  if (do_l2_flush) {
    dst = __shfl_sync(WARP_MASK, dst, leader_id);
    src = &l2_buffers[active_l2_buffer][0];
    warp_memcpy(dst, src, L2_BUFFER_SIZE);

    // Step Ⓑ: Add empty buffer to spare pool
    spare_pool = active_l2_buffer;
  }
} // Step ⑤: Start a new fill phase
```

Listing 4.3: Flush phase of Hierarchical partitioning algorithm.

L2 buffer. If free space remains in the L2 buffer, the warp proceeds to the next fill phase after the eviction completes. Otherwise, if the buffer is full, the warp transitions to the L2 flush.

L2 Flush. The warp flushes the L2 buffer asynchronously to the execution of other warps as follows. In Step ③, the warp first swaps the full buffer with an empty buffer from a spare buffer pool. The swap is non-blocking, as the spare pool contains one spare buffer per warp (i.e., *double-buffering*). Then, the warp releases its lock on the buffer in Step ④. This allows other warps to fill the fresh buffer in parallel to the flush. Thus, the next two steps occur asynchronously to the main control flow. In Step Ⓐ, the warp flushes the full buffer's contents to CPU memory, and inserts the emptied buffer into the spare pool in Step Ⓑ. Finally, the warp proceeds to a new fill phase (⑤).

Design Discussion. A key aspect of Hierarchical is that L2 buffers are flushed asynchronously. This shortens the critical section, as we move the high-latency writes to CPU memory outside of the lock. Crucially, releasing only the L1 buffer is not enough. Instead, the L2 buffer must also be released via double-buffering. Inside the critical section, the buffer swap consists of a pointer update followed by a scratchpad memory fence and has a low overhead.

Overall, our Hierarchical algorithm enables us to efficiently partition large, out-of-core data in CPU memory with a high fanout.

4.4 Scaling the State of a GPU Join

Join algorithms are all limited by the GPU memory capacity and the interconnect bandwidth. For example, no-partitioning joins have poor data locality when the hash table spills to CPU memory, whereas partitioned joins are bandwidth-intensive due to their multiple data passes. The challenge is to achieve good data locality while at the same time reducing interconnect transfers.

In this section, we introduce our *Triton join algorithm*, which is based on the hybrid hash join [124] and aims to balance these two constraints. We optimize our Triton join for GPUs by performing multi-pass radix partitioning [267] (Section 4.4.1), overlapping transfer and compute (Section 4.4.2), and a new caching scheme for in-memory data (Section 4.4.3). By using the GPU to partition data with our Hierarchical algorithm and caching a working set in GPU memory, the Triton join puts into practice our GPU-partitioned join strategy that we describe in Section 4.2.3.

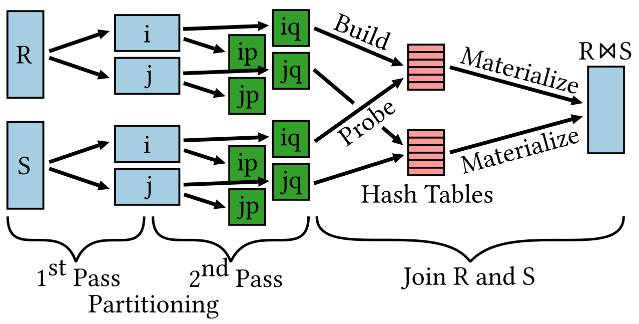


Figure 4.9: The Triton join is based on a parallel radix-partitioned hash join with three stages.

4.4.1 The Triton Join Algorithm

The Triton join algorithm joins an inner relation R and an outer relation S using an equality predicate (i.e., an equi-join). We define the cardinality of R to be smaller or equal to the cardinality of S. We explicitly make no assumptions about the data volume $|R|$ and $|S|$, apart from that the system has enough total memory capacity to store both relations; either relation may be smaller or larger than the GPU memory capacity C.

We illustrate our Triton join algorithm in Figure 4.9. The algorithm consists of three stages:

1st Pass. The first pass radix-partitions R and S by the lower B_1 bits of the hashed join key. We choose B_1 such that two corresponding partition pairs of R and S fit into, e.g., half of the GPU memory, i.e., $|R_i| + |S_i| + |R_j| + |S_j| < \frac{C}{2}$. For example, 1 TiB of data requires $B_1 = 9$ radix bits to store each partition into a 2 GiB memory buffer. Two pairs, i and j, are necessary to pipeline the next algorithm stages. The first partitioning pass uses only a part of the GPU memory's capacity, e.g., $\frac{C}{2}$ to leave space for the results of the 2nd partitioning pass. The partitioning is executed in parallel on the GPU. At the end of this stage, all threads wait at a barrier before the join continues to the second partitioning pass.

2nd Pass. The second pass partitions each R_i and S_i partition by their next higher radix bits. Our choice of B_2 ensures that the resulting R_{ip} partitions fit into the scratchpad memory. For example, a 2 GiB partition requires $B_2 = 15$ radix bits, given a 64 KiB scratchpad. Optionally, the second pass processes only a subset of B_2, and a third pass handles the remainder. The second pass reads data from CPU memory and writes its results to GPU memory. Thus, the third pass and the join phase operate within GPU memory.

Join R and S. The join phase processes each R_{ip} and S_{ip} pair together. The join first

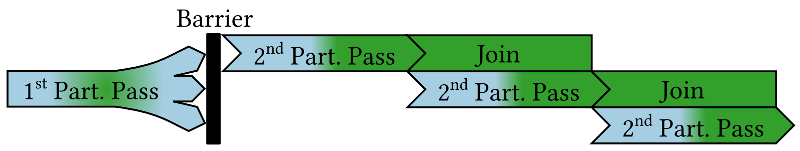


Figure 4.10: In the Triton join, the 2nd partitioning pass and the join are overlapped to optimize interconnect utilization.

builds a hash table in scratchpad memory with R_{ip}. Then, the join probes the hash table with S_{ip}. The join result is written to CPU memory, as, in the general case, the results are larger than the GPU memory capacity. The join requires only a single data pass to materialize results by using a linear allocator [146]. Alternatively, each thread aggregates values inside a register, and the total result of all threads is computed by, e.g., an atomic addition.

4.4.2 Overlapping Transfer and Compute

Pipeline parallelism is an integral part of our Triton join, as pipelining hides the data transfer time. In the partitioning stages, the GPU pulls data from pageable CPU memory on-demand using the cache-coherence [261]. This mechanism enables the hardware to transfer data implicitly and in parallel to computations. However, the Triton join requires multiple kernels to overlap with the data transfer (i.e., the second partitioning pass and the join). Multiple kernels can be overlapped with explicit transfers (e.g., `cudaMemcpyAsync`), but this would require pinned memory. Instead, we devise a new solution based on *concurrent kernel execution* [14, 293].

Concurrent kernel execution enables task parallelism on GPUs by running kernels on different SMs, and serves to increase GPU resource utilization [323]. In our Triton join, we configure each pipeline stage to occupy half of the available SMs and schedule the stages on multiple CUDA streams as shown in Figure 4.10. The GPU then executes the kernels in parallel. Thus, the transfer in the partitioning stage overlaps with the computation in the join stage.

4.4.3 Caching the Working Set in GPU Memory

We transform the partitioned hash join into a hybrid hash join by caching part of the state in GPU memory. Caching state reduces data transfers for small data sets, while providing robustness against performance cliffs when scaling the data size. However, achieving these benefits requires us to consider how caching impacts transfers.

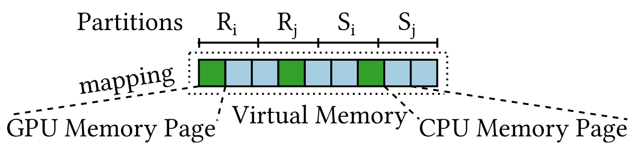


Figure 4.11: State is cached in GPU memory pages that are interleaved with CPU memory pages into a contiguous array.

The Triton join keeps the interconnect busy by distributing the cache space evenly over the intermediate state. We implement the cache by allocating pages that are physically in GPU and CPU memory, and then mapping these pages into a contiguous array in virtual memory, which we illustrate in Figure 4.11. The pages are interleaved in intervals in proportion to the physical allocation sizes, e.g., one GPU page after every two CPU pages. During execution, the GPU accesses multiple pages in parallel, and consistently utilizes the interconnect due to the evenly spaced CPU memory pages.

This is different than the standard hybrid hash join [124], which only caches the hash table of the first partition R_0. After partitioning the data, the hybrid hash join directly joins the partitions, e.g., $R_0 \bowtie S_0$. In contrast, the Triton join performs multiple partitioning passes. Hypothetically, if the Triton join were to cache R_0 and S_0 to speed up the second partitioning pass, the interconnect would be idle while the GPU partitions and joins R_0 and S_0 in GPU memory. Consequently, caching would reduce the transfer-compute overlap and leave performance on the table.

4.5 Evaluation

In this section, we evaluate how well our Triton join scales to large data volumes. We describe our experiment setup and configuration in Section 4.5.1, and then present our results in Section 4.5.2.

4.5.1 Setup and Configuration

We first detail our evaluation environment and methodology. Next, we give an overview of the data sets used in our evaluation. Finally, we outline our experiments.

Environment. We conduct our measurements on an IBM AC922 Power System 8335-GTH. The system consists of two IBM POWER9 (“Monza”) CPUs and two Nvidia Tesla V100-SXM2 (“Volta”) GPUs. Each GPU is connected to one CPU via NVLink 2.0. For PCI-e 3.0 measurements, we use an Nvidia V100-PCIE GPU. Each CPU has 16 cores clocked at 3.8 GHz, that support 4-way SMT and 128-bit VSX SIMD instructions. Each

GPU consists of 80 SMs running at 1.53 GHz. The system contains 128 GiB of CPU memory per socket, and each GPU has 16 GiB of GPU memory. The machine runs Ubuntu 18.04 with Linux 5.0.0-25. Our experiments are implemented in C++ and CUDA. We compile our code with GCC 8.4.0 and CUDA 10.2 with the flags: "-O3 -mcpu=native -mtune=native".

Methodology. We measure the join throughput in billions of tuples per second (*G tuples/s*). As in recent works [261, 367, 385], we define the join throughput as the total input cardinality divided by the total runtime (i.e., $\frac{|R|+|S|}{\text{runtime}}$). We report the mean and standard error over 10 runs for all experiments. We note that our measurements are stable with a standard error below 5%.

Workloads. We specify our default workload similar to related works [48, 226, 261, 385]. We use two base relations, R and S, each consisting of 16-byte <`key,record-id`> tuples. We scale their cardinalities to $|R| = |S| \in \{128, 512, 2048\}$ million tuples (*M tuples*) each. R contains primary keys, and S references the primary keys of R. We randomly shuffle the unique primary keys, generate the foreign keys following a uniform random distribution in the range $s \in [1, |R|]$, and fill the `record-ids` with random values. We store the relations in a column-oriented layout. In summary, we define in-GPU and out-of-core scenarios with up to 61 GiB of data.

Settings. Unless mentioned otherwise, our measurements are configured with the following settings and optimizations. All base relations are stored in pageable CPU memory (i.e., non-pinned). We allocate memory as 2 MiB huge pages [48, 367, 368] on the NUMA node closest to the GPU, and preallocate the pages at boot time [285] to avoid page fragmentation. The GPU directly accesses CPU memory using cache-coherence [261]. We use our Hierarchical partitioning algorithm with 6–10 radix bits for the first pass, and our Shared variant with 9 radix bits for the second pass. For the Triton and radix joins, we use a bucket-chaining hash table [170, 385] with 2048 entries [385]. On the GPU, we store the hash table in the scratchpad cache. For the no-partitioning join, we configure a linear probing scheme with a 50% load factor [219, 352, 357]. In both hashing schemes we use a multiply-shift hash function [127, 352].

Baselines. We measure a radix-partitioned, multi-core hash join implementation [53]. We port all optimizations used by Balkesen et al. [48] to the POWER ISA as described below (our own implementation adds SIMD loads). We extend the code with an array join [367] (i.e., perfect hashing), and partition with 12–14 radix bits in a single pass.

We optimize our CPU implementation (shown in Section 4.2) for the POWER9 architecture. We tune memory reads with SIMD instructions and by disabling stride-N prefetching (`DSCR` = 0) [188], as we observed that stride-N prefetching reduces sequential bandwidth. Note that sequential prefetching is enabled. We optimize the SWWC flush using SIMD stores to write 128-byte cachelines. We note that in contrast to x86_64, the POWER ISA does not support non-temporal stores that bypass the cache [128, 188]. We tested streaming store hints (`dcbtst` and `dcbz`), but these provided no speedup. In our prefix sum, each SIMD lane builds a private histogram to avoid read-after-write hazards [179]. We tune the SMT setting (16, 32, or 64 threads) for each data point.

Experiments. We conduct fourteen experiments. First, we evaluate how our Triton join speeds-up join throughput compared to a GPU no-partitioning join and a CPU radix-partitioned join. We then explain why the Triton join outperforms no-partitioning joins. After that, we profile the Triton join to account where time goes. As the partitioning phase has a large performance impact, we show how it is affected by the processor type (CPU vs. GPU). We analyze the GPU partitioning algorithms in-depth on out-of-core data, and evaluate the second partitioning pass in GPU memory. Based on these results, we tune the partitioning fanouts of the Triton join. Next, we measure the speedup gained by caching, and explore computing the prefix sum on the CPU vs. the GPU. Furthermore, we analyze build-to-probe ratios and wide tuples. Finally, we investigate how future hardware might affect the Triton join.

4.5.2 Experiments

We conduct our experimental evaluation in this section.

Scaling the Triton Join vs. Baselines

In Figure 4.12, we scale the base relations from 128–2048 million tuples per relation. The relations have the same size, and consist of 16-byte tuples. The total data size is thus 3.8–61 GiB, and is up to 122 GiB large when considering the partitioned copy. This is close to the CPU memory capacity of one 128 GiB NUMA node. We compare the throughput of the Triton join to state-of-the-art join strategies on a CPU and a GPU. In addition to the IBM POWER9, we include a Intel Xeon Gold 6126 (“Skylake-SP”) with 12 cores at 2.6 GHz. Note that Figure 4.1 is a simplified version of this experiment with only perfect hashing. Next, we compare the baselines.

CPU Radix Join. The performance of the POWER9 baseline declines by 22% from 1.1 G tuples/s to 0.9 G tuples/s, due to increasing the fanout from 2^{12} to 2^{14}. Perfect hashing is 6–16% faster than bucket chaining. In contrast, the Xeon is slower at 1.0–0.6 G tuples/s. Above 1408 M tuples, the SWWC buffers exceed the Xeon’s 1.25 MiB L3 cache capacity (the POWER9 has 5 MiB/core). Thus, the Xeon switches to two-pass partitioning and a

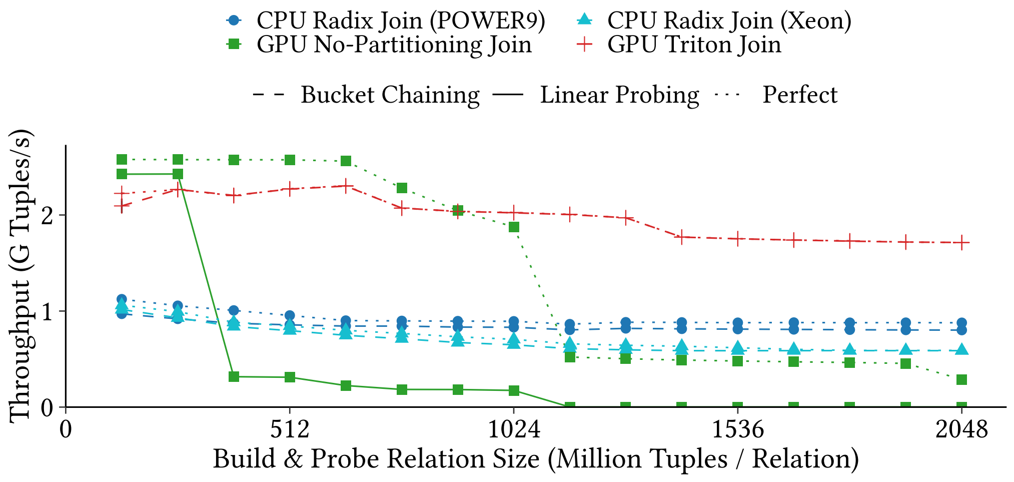


Figure 4.12: Scaling the build-side relation.

2^{18} fanout.

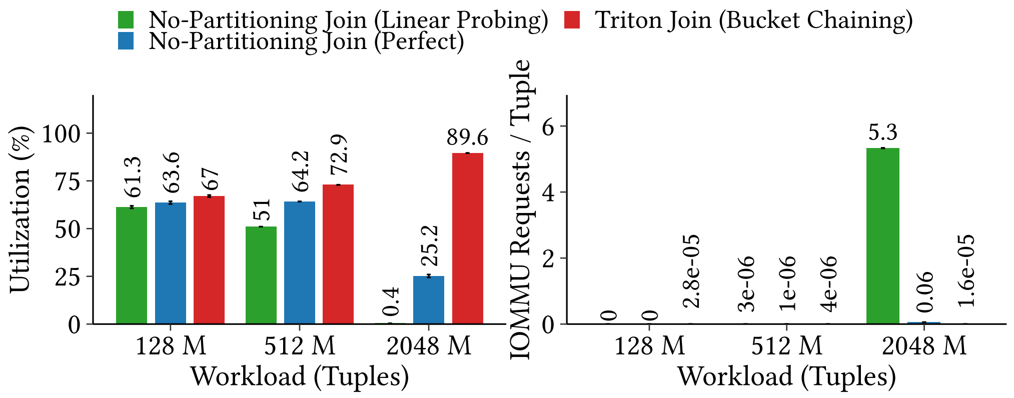
GPU No-partitioning Join. The GPU baseline using perfect hashing achieves 2.5 G tuples/s up to a relation size of 640 M tuples. For larger relations, the throughput decreases to 0.5 G tuples/s for sizes above 1024 M tuples. This performance degradation occurs due to exceeding the GPU memory capacity. In contrast, linear probing reaches only 1.1 M tuples/s for large inputs due to exceeding the GPU TLB range, which we analyze in detail in Section 4.5.2. As a result, perfect hashing is up to 400× faster than linear probing.

GPU Triton Join. The Triton join performs within 85% of the GPU baseline for relations up to 896 M tuples. Then, the Triton join gracefully degrades from 2.3 to 1.7 G tuples/s. It retains 74% of its peak throughput for 2048 M tuples of data. Thus, the Triton join is 1.9–2.6× faster than the POWER9 baseline, and up to 3.9× faster than the GPU baseline with perfect hashing. The performance of bucket chaining remains within 0–2% of perfect hashing.

Summary. We draw three conclusions. First, a no-partitioning join does not scale well on GPUs with fast interconnects. Second, the hashing scheme has a large impact on the no-partitioning join, but only a small impact on the partitioned joins. Third, in all cases, our Triton join outperforms the baselines beyond 1024 M tuples.

Why the Triton Join Outperforms No-partitioning Joins

To better understand the join performance, we analyze the interconnect utilization and GPU TLB misses using hardware performance counters in Figure 4.13. We calculate the



(a) Interconnect Utilization. (b) GPU TLB Misses

Figure 4.13: Interconnect usage of join algorithms.

utilization as the measured bandwidth divided by the theoretical limit. We measure the bandwidth of CPU to GPU transfers including protocol overhead, for which the theoretical limit is 75 GB/s. We use a GPU prefix sum to obtain a full GPU profile. In addition, we count GPU TLB misses as the number of address translation requests received by the CPU's IOMMU [64, 189]. Note that GPU vendors do not expose GPU TLB hardware performance counters [28, 301, 389].

Interconnect Utilization. With an increasing data size, the Triton join caches a smaller proportion of the data in GPU memory. This increases interconnect utilization, as the join phase reads data from CPU memory more often. Closer inspection shows that the prefix sum and partitioning phases are at 90–100% utilization, but the join phase varies between 9–78%. In contrast, the no-partitioning join utilizes the interconnect at up to 63.6% for hash tables in GPU memory, but drops to 25.2% for out-of-core hash tables. With linear probing, utilization drops further to 0.4%.

GPU TLB Misses. GPU TLB misses are the main reason why the no-partitioning join with linear probing has a low interconnect utilization. The 50% load factor doubles the hash table size of linear probing compared to perfect hashing (64 GiB vs. 30.5 GiB for 2048 M tuples), and is rounded up to a power of two. Thus, the hash table exceeds the GPU TLB range of 32 GiB by 2× (see Section 4.2.4). As a result, the GPU issues a translation request to the IOMMU on nearly every memory access, i.e., 5.3 requests per tuple. In contrast, the Triton join issues an IOMMU request once per 10^5 tuples.

Overall, partitioning is effective at reducing TLB misses. Spilling leads to intensive interconnect utilization. However, caching and interconnect utilization are challenging to balance.

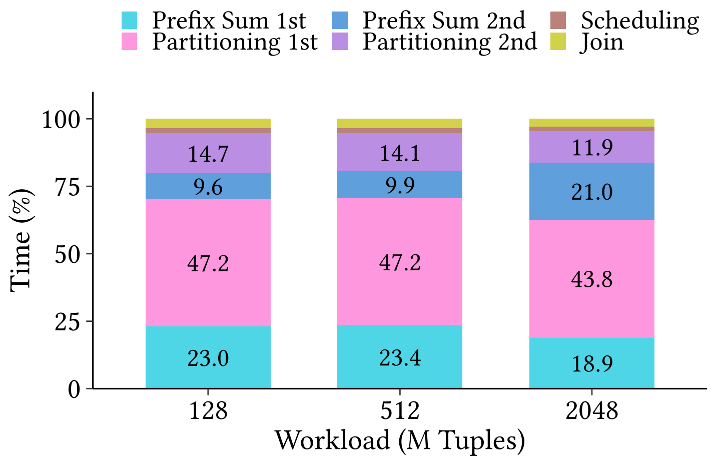


Figure 4.14: Time breakdown of the Triton join.

Time Accounting

As not all phases of the Triton join are interconnect bound, we account where time goes. We break down the execution time per kernel in Figure 4.14, and profile each kernel to find out whether the GPU is executing (instruction issued) or stalling (everything else) in Figure 4.15. We configure a GPU prefix sum instead of a CPU prefix sum to obtain a full GPU profile.

Time Breakdown. Most of the time is spent in the first partitioning pass, which always reads data from CPU memory. This partitioning pass takes 43.8–47.2% of the total time, and the first prefix sum takes 18.9–23.4%. In contrast, the join phase reads data from GPU memory unless data is spilled to CPU memory. In our implementation, the join phase consists of four kernels: a prefix sum and a partitioner for the second pass, a join task scheduler, and the join. Spilling increases the time spent in the second prefix sum, as it copies the data into GPU memory to avoid redundant transfers by subsequent kernels.

Profiling. Both prefix sum passes and the first partitioning pass are mostly interconnect bound due to memory dependencies. In contrast, the second partitioning pass is mostly compute bound due to issuing instructions, as it runs in GPU memory. Only first partitioning pass and second prefix sum pass change with the workload, due to spilling and reloading data. Counterintuitively, large data sizes reduce memory stalls for partitioning, as high fanouts cause additional work.

Our take away is that interconnect bandwidth limits the partitioning phase, but compute power limits the join phase. As bandwidth outweighs computation, we focus

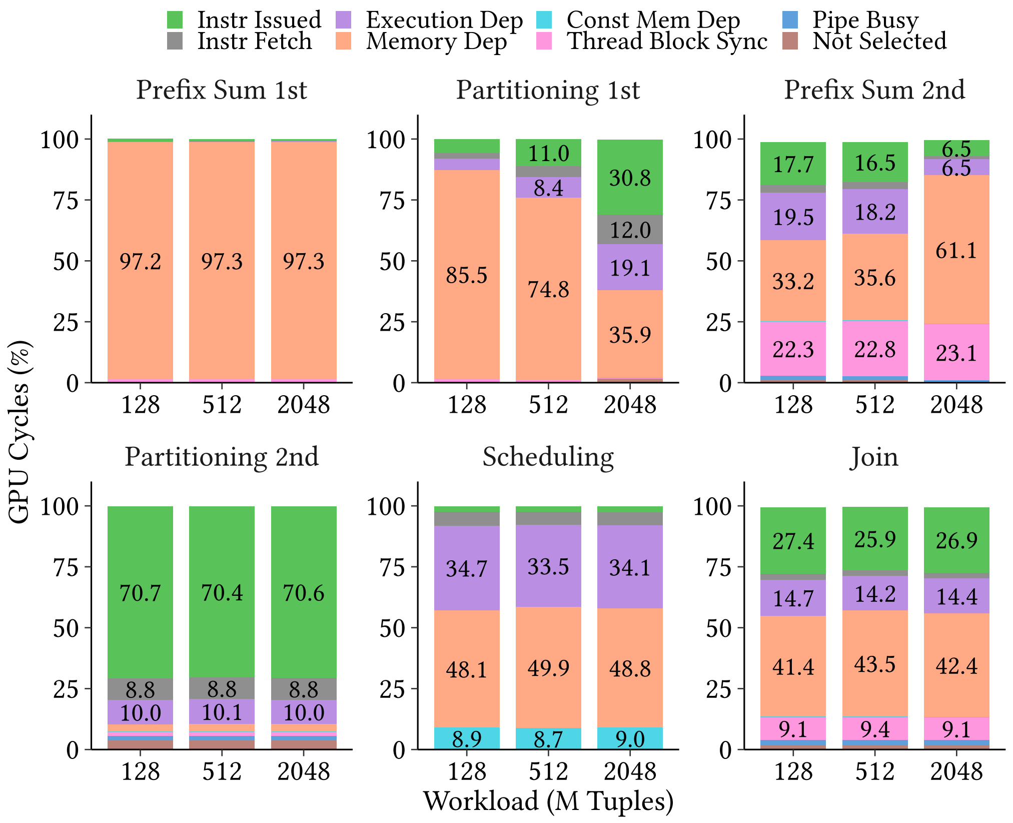
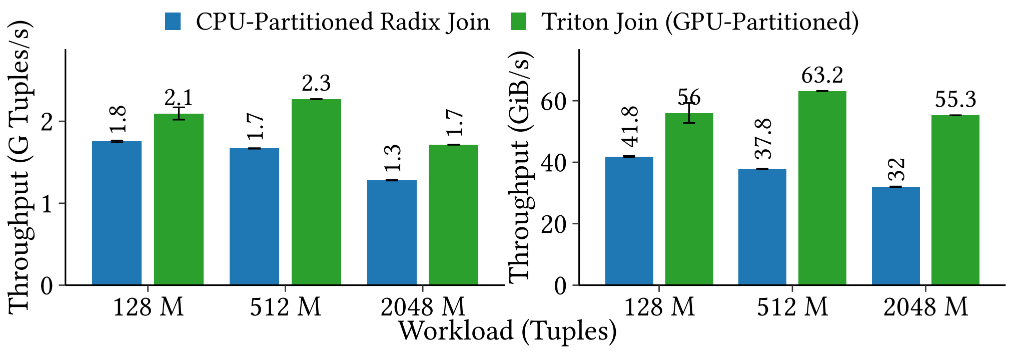


Figure 4.15: Microarchitectural profiling of the Triton join.



(a) End-to-End Join (b) Partitioning

Figure 4.16: Partitioning data using the CPU vs. the GPU.

our optimization efforts on the interconnect in the following sections.

CPU-Partitioned vs. GPU-Partitioned Join

We evaluate the impact of the processor used for partitioning on the end-to-end join in Figure 4.16(a). Following that, we investigate the partitioning phase in Figure 4.16(b). For a fair comparison, we reimplement the strategy of Sioulas et al. [385] and optimize it for the POWER9 and NVLink 2.0 (see Section 4.5.1). The join overlaps the transfer and second partitioning pass over R with the first pass over S, and caches its working set in GPU memory. We compare this CPU-partitioned radix join to our Triton join, that is GPU-partitioned. We run the default workloads, and plot the throughput in G tuples/s for the join and GiB/s for the partitioning.

End-to-End Join. The CPU-partitioned join reaches a throughput of 1.3–1.8 G tuples/s. The 128 M tuple workload has a 38% higher throughput than the 2048 M workload, due to caching the working set. In contrast, the Triton join achieves a 1.2–1.3× speedup.

Partitioning. A closer inspection reveals why the Triton join is faster. First, the GPU partitions data 1.5–1.7× faster than the CPU. Second, the Triton join caches intermediate results in GPU memory, leading to a lower transfer volume. In contrast, the CPU-partitioned join first has to write results to CPU memory and then read them again for the transfer to the GPU, which consumes memory bandwidth. However, the CPU-partitioned join overlaps the partitioning of the outer relation and the transfer of the inner relation. Thus, its join pipeline is 3–13% faster than that of the Triton join.

Overall, our Triton join is faster than the CPU-partitioned join due to partitioning data efficiently on the GPU, and the caching optimizations that this design enables.

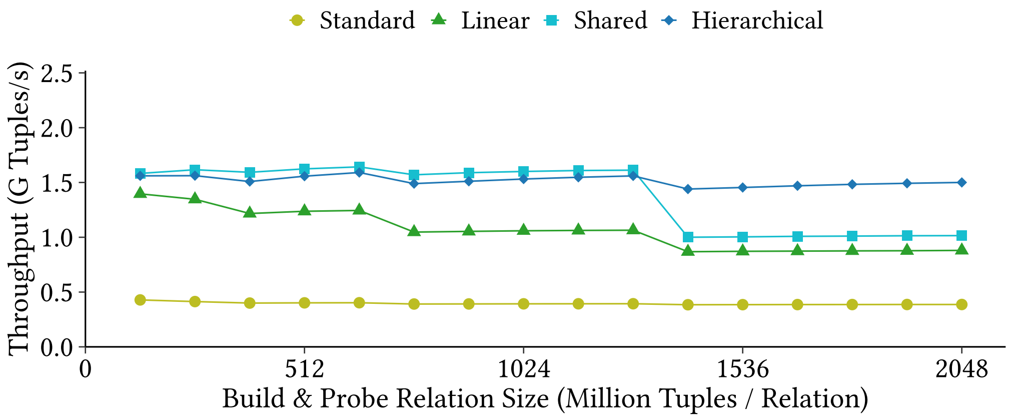


Figure 4.17: Effect of partitioning algorithms on a radix join.

Partitioning Algorithms

In Figure 4.17, we evaluate the impact of the partitioning algorithm on the join. We compare our Shared and Hierarchical to the Linear and *Standard* radix partitioning algorithms. We vary the algorithm used in the first pass and measure the end-to-end join throughput. We scale the base relations from 128 to 2048 M tuples. We disable caching to eliminate side-effects.

Observations. Our Shared algorithm achieves a throughput of 1.5–1.6 G tuples/s up to a size of 1280 M tuples. At this threshold, the flush granularity drops below 128 bytes due to the high fanout. For larger relation sizes, the throughput of Shared reduces to 0.9–1 G tuples/s. In contrast, our Hierarchical variant performs between 1.4–1.5 G tuples/s over the whole range, and degrades gracefully. Thus, Hierarchical achieves a speedup of 1.1–1.9× and 3.6–4× over the Linear and Standard algorithms, respectively.

Overall, the choice of the partitioning algorithm is important for scaling a GPU join. Most notably, our Hierarchical algorithm improves the scaling to large data sizes.

Why Hierarchical Outperforms the State-of-the-Art

To reveal the superior the partitioning throughput, we investigate all partitioning algorithms with hardware performance counters. We use 60 GiB of data, which are sufficiently large to incur TLB misses (see Section 4.2.4). The GPU reads 16-byte tuples from CPU memory, and writes the results back to CPU memory.

Throughput. In Figure 4.18(a), we measure the partitioning throughput in isolation while increasing the fanout. We highlight three aspects. First, the Linear algorithm never

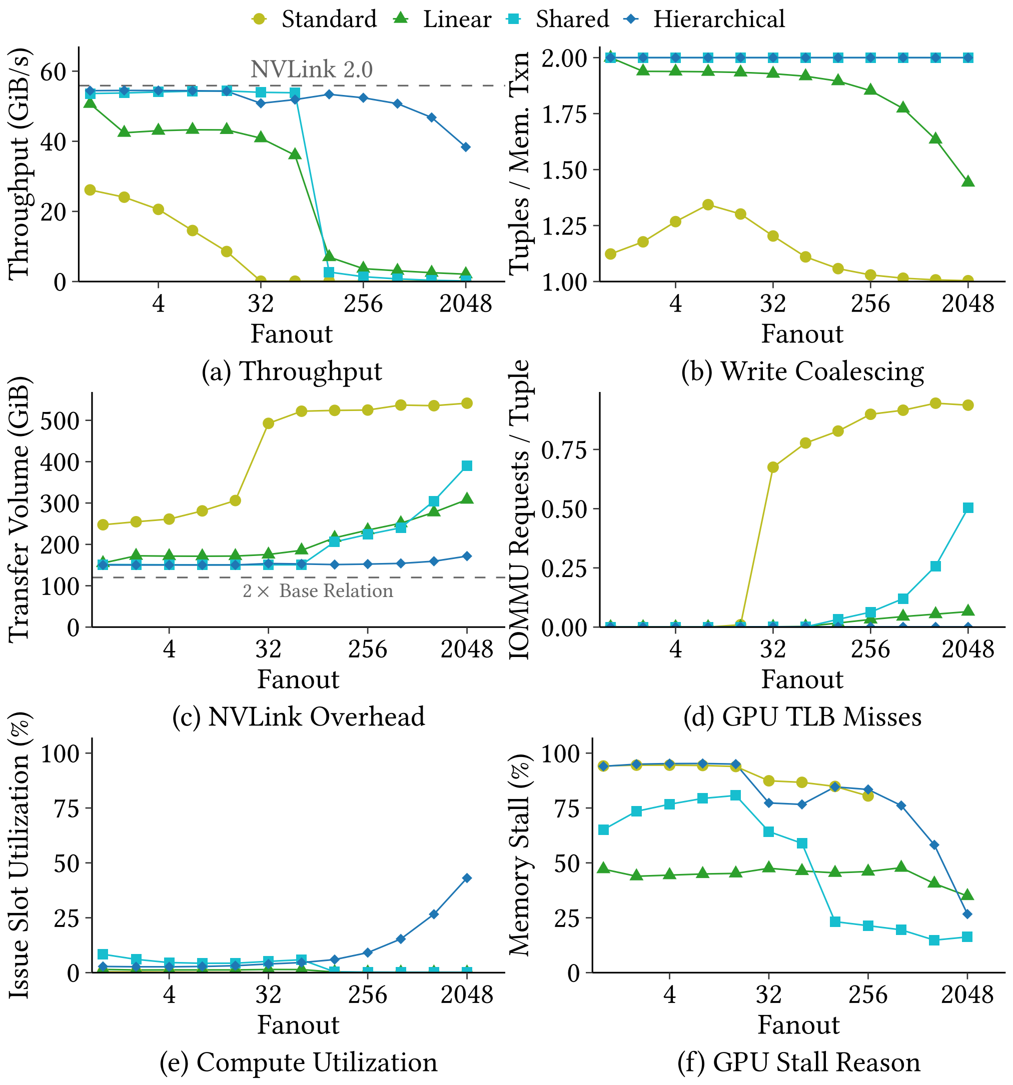


Figure 4.18: Profiling state-of-the-art partitioning algorithms.

achieves the bidirectional interconnect bandwidth of 55.9 GiB/s. It reaches 50.7 GiB/s for one partition (i.e., a memcopy), but then drops to 42.4 GiB/s for a fanout of 2. Second, Shared partitions at 54 GiB/s, but does not scale beyond a fanout of 64. In contrast, our Hierarchical algorithm achieves 38.3 GiB/s even at a fanout of 2048.

Write Coalescing. To reveal the reasons for the performance, we begin by recording the tuples per memory transaction in Figure 4.18(b). We find that Linear only partially coalesces writes. The reason is that sorting tuples by partition usually does not result in batches of exactly 128 bytes. High fanouts increase this effect as the tuples cached per partition decrease. In contrast, both of our algorithms perfectly coalesce writes by design.

NVLink Overhead. Ineffective coalescing leads to the high physical transfer volumes in Figure 4.18(c). This overhead results from the interconnect packet header attached to each payload, and is higher for small payload sizes. In the case of Linear, interconnect overhead accounts for up to 156% of the transfer volume. In contrast, our Hierarchical algorithm remains below 43%.

GPU TLB Misses. In Figure 4.18(d), we unmask the performance barrier at a fanout of 64 by measuring the IOMMU requests. Going from 64 to 128 partitions causes the TLB miss rate of Shared to increase by 33×, i.e., a miss on every second flush. In contrast, at a fanout of 2048 Hierarchical achieves a 1436×, 100×, and 771× lower miss rate compared to Standard, Linear, and Shared, respectively.

Compute Utilization. We inspect if computation limits throughput by reporting the "percentage of issue slots that issued at least one instruction" [301] in Figure 4.18(e). Typically, utilization remains below 5%. Only Hierarchical utilizes up to 43% of the GPU with high fanouts. The trend starts when the buffer size drops to 16 tuples at a fanout of 256, and flushing no longer occupies a full warp.

GPU Stall Reason. In Figure 4.18(f), we reveal why compute utilization is low. Shared and Hierarchical stall on memory dependencies 65–90% of the time. In contrast, Linear additionally stalls on synchronization and pipeline busy [301]. TLB misses manifest themselves as instruction latency, i.e., execution dependency and pipeline busy stalls. For Standard, the stall counters overflow for fanouts of 512–2048 due to its runtime of 10 minutes [387].

We conclude that the data access pattern and TLB miss tolerance of our Shared and Hierarchical algorithms are the main reasons they outperform the Standard and Linear approaches.

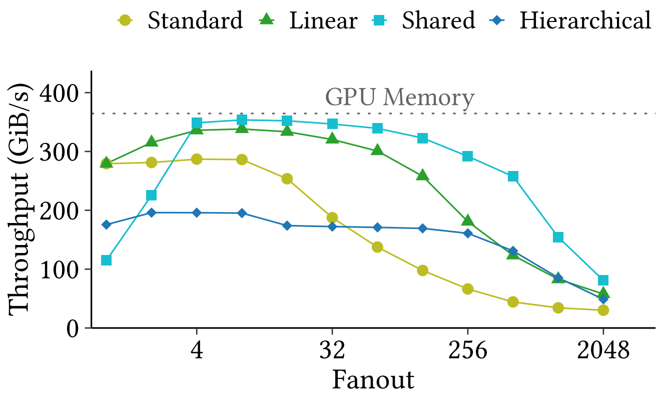


Figure 4.19: Second radix partitioning pass in GPU memory.

Second Radix Partitioning Pass

The second partitioning pass of the Triton join writes its partitions to GPU memory. Therefore, in Figure 4.19, we evaluate the throughput of state-of-the-art radix partitioning algorithms when the input and output data reside in GPU memory. We partition 470 M tuples (7 GiB) and plot the memcopy throughput as the GPU memory baseline.

Shared Algorithm. Our Shared algorithm outperforms Linear by 1.04–2.1× for fanouts above four. The reason is that the perfect coalescing of the Shared algorithm leads to a higher write bandwidth also in GPU memory. However, small fanouts perform poorly as threads must frequently wait until the software write-combine buffer is flushed. This issue could be resolved by asynchronously flushing the buffers similarly to the Hierarchical algorithm, at the cost of occupying more space in scratchpad memory. However, this solution would reduce performance for high fanouts.

Hierarchical Algorithm. In contrast, our Hierarchical algorithm reaches less than 60% of the Shared algorithm's throughput. This results from the two-level software write-combining performed by Hierarchical, which incurs extra accesses to GPU memory.

Overall, we utilize our Shared algorithm in the Triton join's second partitioning pass due to its high in-GPU performance.

Which fanouts to choose?

We configure the Triton join with two radix partitioning passes, which makes the join challenging to tune. In Figure 4.20, we evaluate which combination of fanouts achieves the highest join throughput. We measure the three default workloads, with all fanout combinations for which the resulting hash table fits into the GPU scratchpad cache. We

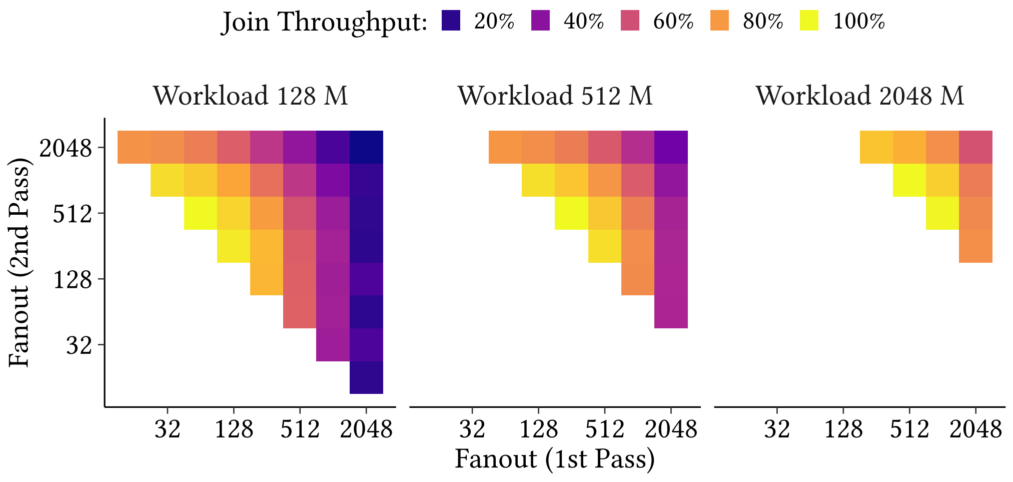


Figure 4.20: Tuning the partitioning fanouts for the Triton join.

present the measurements as a heat map. On the X- and Y-axis, we scale the fanouts of the first and second partitioning passes, respectively. In the heat map, we show the join throughput as a percentage of the maximum value. Bright colors map to a high throughput, and dark colors to a low throughput.

Observations. The "hot" diagonal of each heat map represents the *Pareto frontier*, on which the optimal parameter combination typically lies. Fanouts above the frontier over-partition the data, and fanouts below the frontier under-partition the data. We note that the maximum throughput for all workloads is on the Pareto frontier. Furthermore, the optimal fanout for second partitioning pass is 512. However, the best fanout for the first pass varies between 64–1024. This variation is due to the constant size of the scratchpad cache, into which the hash table must fit. Thus, we deduce that the optimal fanout depends on the size of R.

In Equation 4.1, we calculate the optimal fanout for the first partitioning pass based on the size of the inner relation R:

$$\text{fanout}_{\text{1st}} := 2^{\lceil \log_2(\text{size}(R)/\text{size}(\text{cache})) \rceil - \log_2(\text{fanout}_{\text{2nd}})} \tag{4.1}$$

In summary, our measurements suggest that the Triton join is more sensitive to the first partitioning pass than the second pass. Therefore, we simplify tuning by providing a closed-form equation to calculate the best fanout for the first pass.

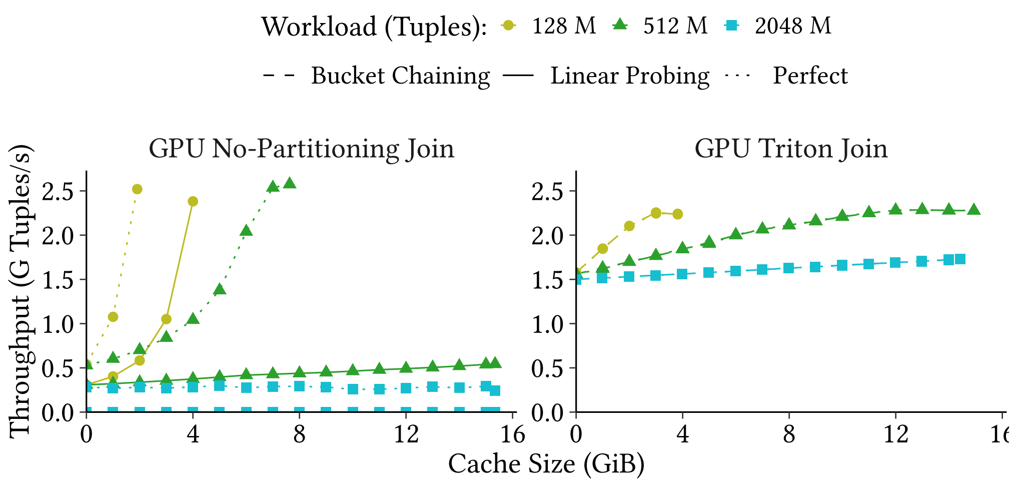


Figure 4.21: Scaling the GPU memory cache size.

Caching the Working Set in GPU Memory

We explore the effect of caching on the throughput of the no-partitioning join and the Triton join in Figure 4.21. We scale the cache size in GPU memory from 0 to 14.9 GiB. For the no-partitioning join, we cache part of the hash table [261]. We note that the Triton join with no cache is effectively a two-pass radix join, and that a part of the GPU memory is required for the join pipeline.

GPU No-Partitioning Join. Caching the entire hash table instead of not caching anything increases throughput by 4.6–4.8× for the 128 M and 512 M workloads using perfect hashing. In contrast, caching has no effect on the 2048 M workload. The reason is the high cache miss rate of 50%. A miss rate of only 4% reduces the gain to 1.8× for 512 M with linear probing. In contrast, for 2048 M with linear probing the reason is that GPU TLB misses slow down the join (see Section 4.5.2).

GPU Triton Join. In contrast, the 128 M and 512 M workloads improve performance by 1.4×, and the 2048 M workload by 1.1×. However, the 128 M workload slows down by 1.5% when the whole working set is cached, instead of only 79% of the working set. This is because the GPU memory and interconnect together provide more bandwidth than GPU memory alone [6, 339].

Our take-away is that the Triton join robustly scales with the cache size, and avoids sharp performance cliffs caused by the TLB range and the GPU memory capacity.

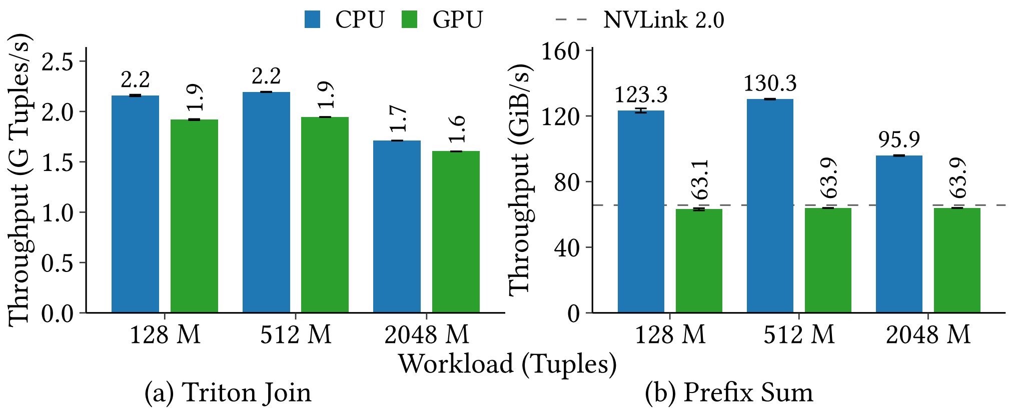


Figure 4.22: Prefix sum on the CPU vs. on the GPU.

CPU vs. GPU Prefix Sum

We determine which processor computes the prefix sum faster. We first assess the effect of the prefix sum on the end-to-end join (Figure 4.22(a)), and then measure the prefix sum throughput achieved by the CPU and GPU (Figure 4.22(b)). We run the experiment using our Triton join on the default working sets with 128, 512, and 2048 M tuples. We show the prefix sum throughput in GiB/s to enable a comparison with the memory bandwidth. We highlight that the prefix sum reads a single column per relation, due to the columnar layout.

Triton Join. We observe that when using the CPU, the Triton join achieves a throughput of 2.2 G tuples/s for the 128 M and 512 M workloads, and 1.6 G tuples/s for the 2048 M workload. These results are 1.1× faster than when computing the prefix sum on the GPU.

Prefix Sum. The CPU achieves up to 129.6 GiB/s, and is able to nearly saturate the CPU memory bandwidth. For the 2048 M tuples workload, the throughput decreases to 96 GiB/s. In contrast, the throughput of the GPU is constant at 63 GiB/s. The reason is that reads are unidirectional transfers, and thus the GPU is constrained by the interconnect bandwidth.

Overall, the CPU is able to sequentially scan data faster than the GPU, and thus computes the prefix sum 1.6–2.2× faster. However, the prefix sum has a small impact on the overall join throughput.

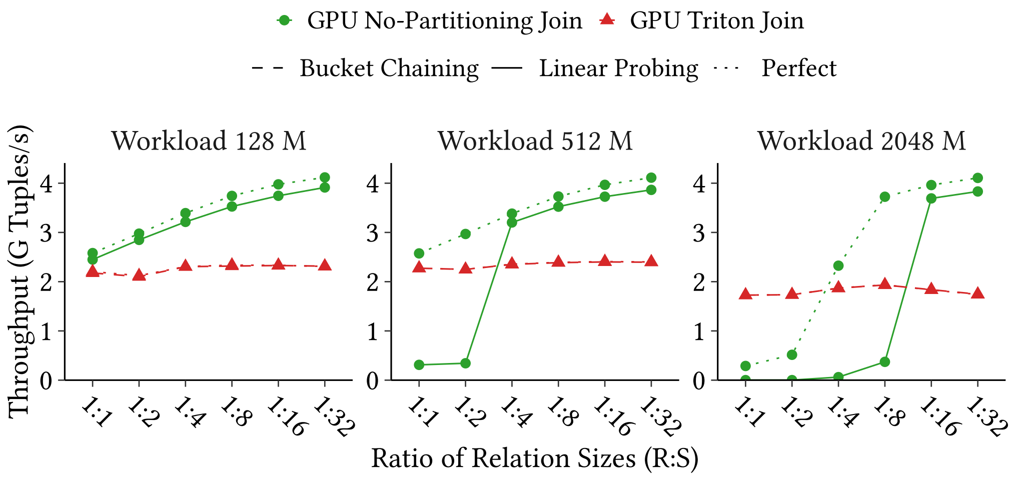


Figure 4.23: Varying build-to-probe ratios with the Triton join.

Build-to-probe Ratios

In Figure 4.23, we measure the throughput of the Triton join for different build-to-probe ratios. For each workload, we scale the ratio from 1:1 to 1:32 while keeping the data constant at 61 GiB. For example, for the 2048 M workload 1:1 means 2048:2048 M tuples, and 1:32 means 124:3972 M tuples.

Observations. The no-partitioning join is subject to two effects. First, the GPU memory capacity causes an abrupt performance cliff. The extreme case is linear probing, for which a 1:32 ratio is 3414× faster than 1:1 in the 2048 M workload. Second, reducing the build size within GPU memory causes a 60% speedup. Dissecting the perfect hashing variant shows that the probe throughput is 4.3 G tuples/s, whereas the build throughput is only 1.8 G tuples/s. In a deeper investigation, we find that random GPU memory reads are 3.2–6× faster than writes. In contrast, the throughput of the Triton join remains stable between 1.66-1.88 G tuples/s. This increase results from reducing the fanout from 1024 to 64 partitions, which increases partitioning throughput.

We conclude that the Triton join is insensitive to the build-to-probe ratio, due to partitioning the large outer relation. Thus, a no-partitioning join should be preferred for high ratios.

Tuple Width

The relation size is determined both by the number and width of tuples. In Figure 4.24, we investigate how materializing wide tuples affects the Triton join. Instead of reading

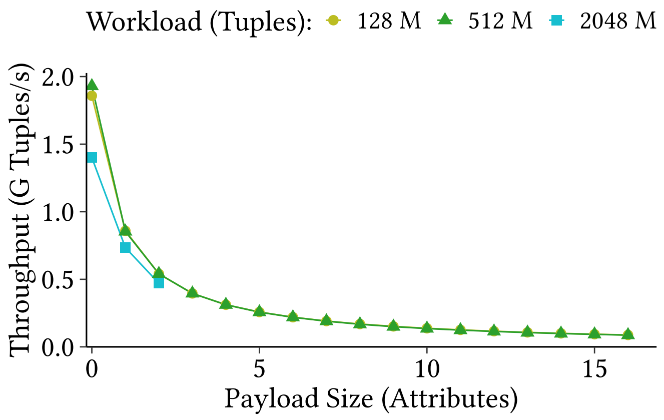


Figure 4.24: Scaling the number of payload attributes.

two attributes, we partition only the join key and generate row IDs on-the-fly in the first pass. Thus, the join results in a join index, with which we materialize and aggregate the out-of-core, 8-byte payload attributes of the outer relation.

Observations. At 2 G tuples/s and 1.5 G tuples/s, constructing a join index (i.e., no payload) achieves a similar throughput as our default setup, which early-materializes one payload attribute. In contrast, late materialization incurs a random CPU memory access per attribute, which causes performance to degrade to 86–88 M tuples/s for 16 payloads. The 2048 M workload stops at two payloads due to reaching the CPU memory capacity.

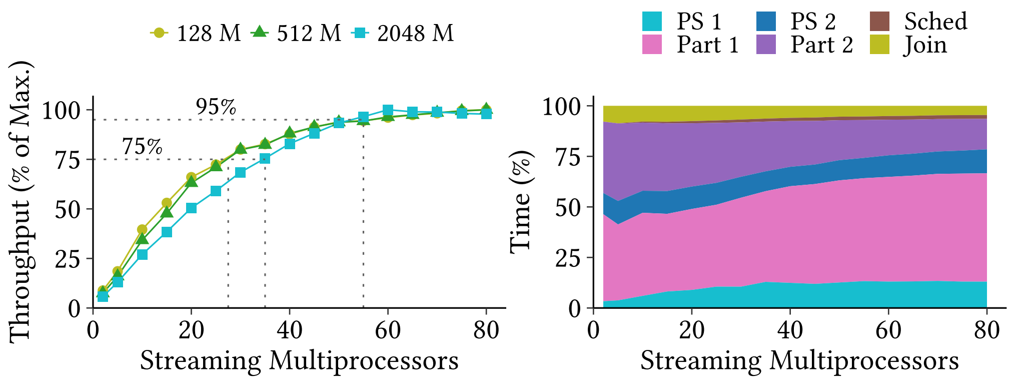
In conclusion, partitioning leads to expensive random accesses during late materialization. Our results indicate that materializing wide, out-of-core tuples requires further investigation.

Compute Power Scaling

We explore how future hardware might affect the Triton join by scaling number of streaming multiprocessors in Figure 4.25. We measure the throughput as a percentage of the maximum, and explain the scaling behavior by examining where the join spends time in the 512 M tuples workload.

Workloads. We observe that 28 SMs suffice to achieve 75% of the peak throughput for the 128 M and 512 M workloads. The 95% mark is passed for all our workloads with 55 SMs.

Time Breakdown. The Triton join scales quickly at first, as the first and the second partitioning passes are compute bound below 25 SMs. With more than 25 SMs, profiling



(a) Workloads (b) Time Breakdown (512 M)

Figure 4.25: Compute power required for high throughput.

shows that the first pass becomes interconnect bound and stops scaling. In contrast, the second pass remains compute bound and continues to scale, but with diminishing returns. As a result, the overall scaling levels out.

We conclude that the Triton join is interconnect bound. A faster interconnect would increase join throughput, whereas a faster GPU would not yield significant gains.

4.5.3 Discussion

In this chapter, we have investigated how fast interconnects can resolve the memory capacity limitation to scale the GPU join state, and have gained the following key insights.

GPUs with fast interconnects scale to a large join state. Fast interconnects provide sufficient bandwidth to spill large state to CPU memory. A 2× speedup over a strong CPU baseline is possible even when the state size exceeds the GPU memory capacity.

GPUs robustly spill state to CPU memory. We learned that partitioning and caching can be combined to gracefully degrade throughput. Thus, we are able to avoid performance cliffs caused by the TLB range and the GPU memory capacity.

GPUs are able to process tasks end-to-end. Fast interconnects obviate CPU involvement. For example, the CPU no longer must partition data or manage transfer pipelines. This enables DBMSs to efficiently use heterogeneous hardware.

Interconnect-consciousness enables fast random accesses. Perfect coalescing saturates a fast interconnect. Thus, adapting the access pattern to the interconnect makes new use-cases possible.

Concurrent kernel execution is a versatile replacement for DMA copy engines. In addition to overlapping computation and transfers from pageable memory, kernels are able to directly compute or reshape the data. Thus, concurrent kernel execution helps to reduce GPU memory traffic and improve data access patterns.

Interconnect bandwidth is no longer the main bottleneck. In some cases, the high interconnect bandwidth shifts the bottleneck to other resources, such as random access bandwidth, TLB misses, and computation. Optimization becomes challenging, as multiple constraints can simultaneously affect different parts of the program.

Summary. Fast interconnects enable GPUs to cover a broader spectrum of database use-cases, but we require new algorithms to fully exploit the performance potential of fast interconnects.

4.6 Related Work

In this section, we contrast our contributions to related work.

Scalable Co-Processing. Recent GPU-enabled DBMSs [78, 79, 102, 103, 146, 170, 176] and machine learning frameworks [40, 264] are able to process data sets larger than GPU memory. Our work complements these systems by scaling the operator state, thus enabling large data sizes.

Relational and ML operators stream data from CPU memory to the GPU to transfer data efficiently across the interconnect [213, 219, 259, 357]. In contrast to these works, we scale operator state in addition to scaling the data size.

Join Co-Processing. Speeding up joins on co-processors has been of particular interest for database research [170, 171, 173, 174, 213, 277, 288, 339, 420]. Recent works investigate radix-partitioned joins on GPUs [321, 361], MICs [97, 206, 343], and FPGAs [96, 167, 168, 240]. However, these approaches limit the join state to the co-processor's on-board memory, or assume a coupled architecture in which the co-processor has direct CPU memory access. In contrast, our Triton join handles large state on a discrete GPU.

Radix Partitioning on Co-Processors. Radix partitioning has been investigated on GPUs, MICs, and FPGAs. Early GPU works suggest a binary divide-and-conquer approach [363, 364], that requires a data pass per radix bit. More recently, GPUs with atomic additions enable a single-pass approach that sorts data in scratchpad memory [361, 395]. In contrast, our Shared algorithm extends software write-combining [364] to fully coalesce writes on GPUs.

SWWC partitioning has been ported to MICs by SIMD vectorization [344, 345, 346]. Our Shared is structurally similar to vectorized SWWC. However, in contrast to SIMD partitioning, Shared saves cache space by sharing buffers among warps. In analogous terms, in our design, SMT threads share buffers in the L1 cache, in addition to SIMD vectorization. To the best of our knowledge, no prior work considers such a design on any processor architecture.

On FPGAs, write-combining can be implemented in hardware instead of in software [215, 409]. However, previous studies have been limited by slow interconnects that incur a data transfer bottleneck.

End-to-End Join Queries. Join state compression [52, 57, 149, 243], filtering [49, 163, 365] and pipelining [50, 434] the outer relation, and efficient tuple materialization [268, 345, 346, 430] have been proposed to speed-up joins. These optimizations complement our work and remain open challenges for GPUs with fast interconnects.

Transfer Bottleneck. Previous works consider scaling operator state for joins [163, 340, 385], sorting [395], and the primitives underlying these operators [159]. However, these works assume that PCI-e causes a transfer bottleneck. In contrast, we take advantage of fast interconnects by proposing a new approach that eliminates CPU pre-processing steps.

Fast Interconnects. GPUs with NVLink have been explored to speed up query processing. Recent works investigate the data transfer bottleneck [261], lazy transfers and scan sharing for HTAP DBMSs [349], multi-GPU joins [149, 324, 360], CSV loading [235], and sorting [265]. FPGAs with OpenCAPI have been exploited to scale the outer relation of a join [216] and data loading [332, 333]. In contrast, we show that by carefully designing algorithms for fast interconnects, GPUs efficiently accelerate joins with a large state.

4.7 Conclusion

Fast interconnects are not a silver bullet for large-scale hash joins. Our analysis of NVLink 2.0 reveals that interconnect overhead and TLB misses reduce performance. We propose our Triton join to overcome these challenges. Our hardware insights lead to a GPU-partitioned join strategy based on a new GPU partitioning algorithm. Overall, our Triton join scales to large data volumes at up to 400× faster performance by being aware of the fast interconnect.

5 Scalable Iterative Algorithms

Our goal in this chapter is to efficiently process machine learning queries on large data sets using a GPU. We investigate *k*-means, a well-understood and versatile machine learning algorithm. We find that existing approaches do not utilize the interconnect efficiently, as these speed-up computations but neglect to optimize data transfers. In contrast, we propose a new execution strategy for *k*-means which reduces repeatedly occurring data transfer overhead. By considering transfers, we efficiently harness fast interconnects to scale *k*-means on GPUs.

5.1 Introduction

k-means [256, 263] is an essential tool in the data scientist's toolkit to find patterns in large data sets. In particular, practitioners of data-driven sciences, such as genome analysis [177, 407, 425] and climatology [88, 112, 228] require fast *k*-means implementations for a short data to knowledge time. Furthermore, many algorithms build on top of *k*-means to cover new use-cases, e.g., BIRCH [428], streaming *k*-means [383], and deep clustering [86, 87]. Thus, speeding up *k*-means enables data scientists to create new insights by exploiting larger data sets in higher quality. Although relational databases support *k*-means via SQL [178, 317], high-performance *k*-means requires specialized database features [320, 369]. The ubiquitous availability of GPUs provided by cloud computing platforms promises inexpensive and fast execution of machine learning queries. However, to exploit the full performance of GPUs, algorithms require careful design and tuning, as shown by previous research on accelerating relational data

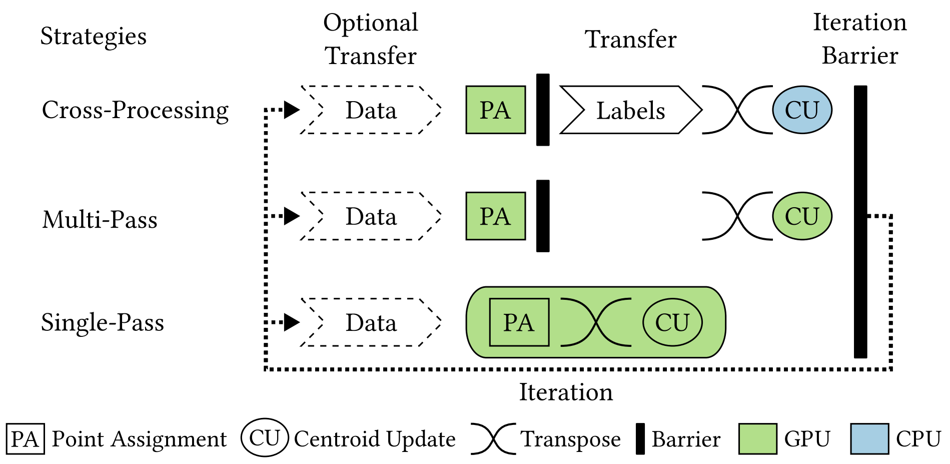


Figure 5.1: *k*-Means Execution Strategies.

management with GPUs [78, 79, 146, 170, 176, 219, 340, 341].

In Figure 5.1, we optimize *k*-means for GPUs, step by step. The *Cross-Processing* strategy results from the two phases in each iteration of *k*-means: point assignment and centroid update. Research over the last decade focused mostly on accelerating these phases on CPU or GPU separately [85, 95, 139, 166, 373, 412]. In particular, they compute the point assignment phase on the GPU and perform the centroid update on the CPU. This split between CPU and GPU causes the *cross-processing problem*, because the split requires a data exchange over the interconnect in each iteration. In contrast, some approaches avoid the split by performing point assignment and centroid update on the GPU [44, 252]. However, both types of approaches introduce artificial synchronization barriers between the two phases. The barriers require these approaches to make two passes over the data points. As large data sets cannot be stored entirely in GPU memory, each pass incurs a data transfer. The resulting *multi-pass problem* decreases the throughput up to a factor of two. Overall, current approaches do not scale to large data volumes because they cannot efficiently fuse both phases.

In this chapter, we address both problems to exploit the processing power of modern GPUs for running *k*-means on large data volumes. Our contributions are as follows:

1. We propose a novel centroid update algorithm for GPUs that solves the cross-processing problem by eliminating artificial synchronization barriers between the phases of *k*-means.

2. We introduce the Single-Pass execution strategy on GPUs to solve the multi-pass problem by making a single pass over the data points.

3. We show how our new *k*-means execution strategy scales to large data sets that exceed the GPU memory capacity.

4. We evaluate the Multi-Pass and Single-Pass strategies against the state-of-the-art Cross-Processing strategy on CPUs and GPUs in scenarios which are either data-intensive or compute-intensive.

The remainder of the chapter is structured as follows. In Section 5.2, we describe our GPU-optimized centroid update. After that, we contribute our novel *Single-Pass* execution strategy in Section 5.3 and show its application for arbitrarily large data sets in Section 5.4. In Section 5.5, we present our experimental results. Finally, we review related work in Section 5.6 and conclude the chapter in Section 5.7.

5.2 Efficient Centroid Update

In this section, we discuss different strategies to compute the centroid update efficiently on GPUs. Our efficient GPU implementation allows us to perform *k*-means completely on the GPU. This eliminates the transfer of data point labels over PCI-e, which Cross-Processing inherently requires. In general, the centroid update consists of two parts: First, computing the *Feature Sum* and second, the *Mass Sum*. We discuss these individually in Sections 5.2.1 and 5.2.2, as they require different approaches. The new centroids are obtained by dividing feature sum vectors by the mass sum vector.

5.2.1 Feature Sum

During the Feature Sum calculation, *k*-means adds up the individual feature values of all points that belong to the same cluster and stores the result in a vector of feature sums. As a result, Feature Sum is logically equivalent to a SQL group-by aggregation query, where we group by the labels and compute a sum for each feature. We discuss the relation to group-by in Section 5.6.

Cluster Merge. We now describe the design of Cluster Merge, depicted in Figure 5.2. The points are stored in column major format, such that each feature of a point is stored in a separate array. Together, the arrays represent a matrix.

In Cluster Merge, each thread processes a point (i.e., all features, see ① in Figure 5.2) and aggregates the point in a private *clusters* × *features* matrix (see ② in Figure 5.2) using the point's label as the cluster index. In a final step, all threads merge their partial results with a reduction [169] to the final feature sum vector (see ③ in Figure 5.2).

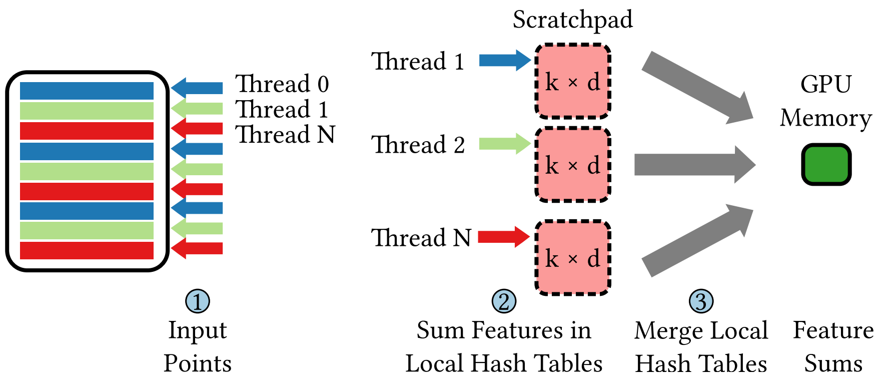


Figure 5.2: Feature Sum strategy: Cluster Merge. Each color represents a single thread processing one point at a time.

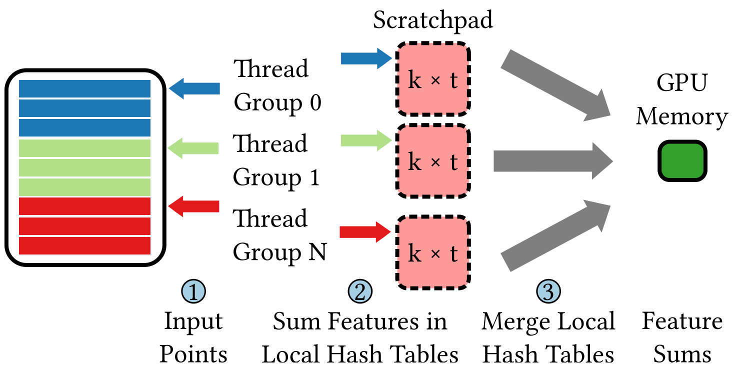


Figure 5.3: Feature Sum strategy: Partitioned Features. Each color represents a thread block processing multiple points at a time.

Every thread allocates memory to store a private replica of the *clusters* × *features* matrix (i.e., the working set). This has the benefit that no atomic writes or locks slow down performance when scaling the number of threads.

However, the thread-private matrix requires a large amount of cache space per thread. Each thread requires $4 \times k \times d$ bytes of additional cache space to store its working set, assuming a four-byte floating-point datatype. Thus, we can either scale the number of threads and face cache thrashing, or we use too few threads and, as a result, underutilize the GPU SMs and memory controllers.

Partitioned Features. Our *Partitioned Features* strategy, shown in Figure 5.3, optimizes Feature Sum for the hardware architecture of GPUs. The key idea of Partitioned Features is to exchange data through *thread block synchronization*. Within a thread block, we vertically partition a batch of data points, such that each thread is responsible for a particular feature instead of a complete data point. Thus, each thread of a thread block processes a different feature of the same data point. Depending on the number of features, a thread block processes multiple points at once. This allows us to avoid cache thrashing by reducing the working set of each thread to $4 \times k$ bytes (compared to $4 \times k \times d$ for Cluster Merge). This approach requires a barrier within the thread block, which is typically fast on GPUs.

Advantages. The main advantages over Cluster Merge are as follows. First, Partitioned Features shares a local $clusters \times features$ matrix *per thread block* (see ② in Figure 5.3). Each thread block consists of t threads, where each thread stores a vector of k sums, i.e., one sum per centroid. In total, each matrix has a size of $k \times t$ elements. Thus, as each thread writes exclusively to its own vector within the shared matrix, no atomic writes are needed.

Second, each thread of the same thread block reads one feature of the same point and adds it to the feature sum of the cluster indicated by the label. The whole thread block processes a batch of data points (see ① in Figure 5.3). Thus, a single matrix per thread block is sufficient to store the intermediate result, and the GPU can coalesce memory accesses to the data points.

As in Cluster Merge, each point's label is read from GPU memory only once. However, sharing the label among threads introduces a barrier within the thread block. In a final step, all thread blocks reduce their partial results to obtain the final feature sum matrix (see ③ in Figure 5.3).

Edge Cases. Depending on the data set, the number of features d can be less than or greater than the thread block size (i.e., $d < t$ or $d > t$). We handle these cases by either processing multiple points in the same thread block ($d < t$, depicted in Figure 5.3), or by partitioning features of the same point over multiple thread blocks ($d > t$).

Summary. Overall, our Partitioned Features strategy achieves better cache-efficiency than Cluster Merge through thread block synchronization. Thus, Partitioned Features runs with a higher number of threads on GPUs compared to Cluster Merge, which results in more efficient GPU execution.

5.2.2 Mass Sum

Mass Sum is the second part of the centroid update. It calculates a histogram which counts the number of points in every cluster. In contrast to a general histogram computation, Mass Sum uses the point label as an index to directly access a bucket, instead of first calculating the bucket's index. Three differences distinguish Mass Sum from Feature

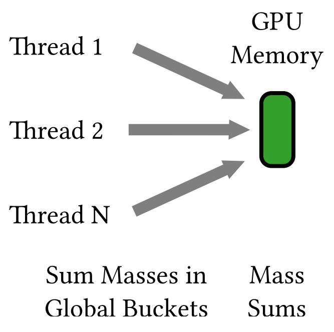


Figure 5.4: Mass Sum strategy: Global Atomic. Threads compute a global histogram.

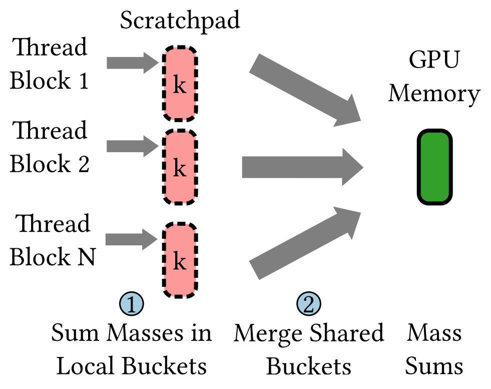


Figure 5.5: Mass Sum strategy: Partitioned Local. Each thread block computes a local histogram.

Sum: (a) it has only one dimension, (b) the labels are counted, thus no point data is read, and (c) it increments integers instead of adding floating point numbers. The last property is relevant because GPUs are faster at atomic integer increments than atomic floating point additions.

Depending on k, different approaches are needed to balance the synchronization cost between threads and the merging cost for combining intermediate results. Thus, we consider four strategies for Mass Sum: Global Atomic, Partitioned Global, Partitioned Local, and Partitioned Private.

Global Atomic. The most simple way to compute a histogram on a GPU is to create one global histogram that is updated by all threads, as we illustrate in Figure 5.4. We refer to this approach as Global Atomic, because all threads synchronize globally on each bucket using atomics to ensure a correct result. Global Atomic is easy to implement and performs well for a large number of clusters ($k > 1000$). However, it causes heavy contention for a small number of clusters. For these cases, we need a different strategy.

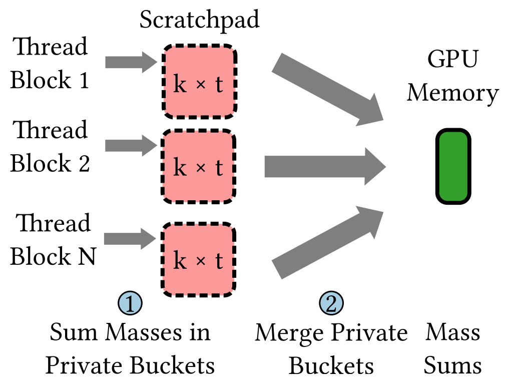
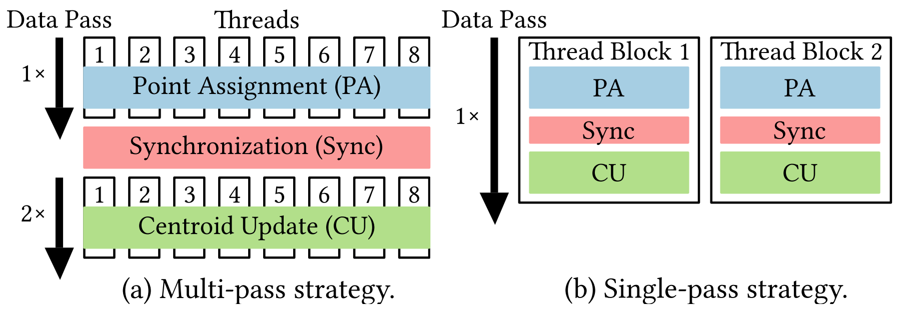


Figure 5.6: Mass Sum strategy: Partitioned Private. Each thread in a thread block writes into private histogram buckets.

Partitioned Global and Partitioned Local. We provide each thread block a dedicated histogram stored in GPU memory to reduce contention between threads. Therefore, threads in different thread blocks do not need to synchronize, which significantly improves performance. This comes at the cost of merging the individual histograms at the end of the computation by a parallel reduction step. Hardware support for atomic additions in scratchpad memory [298] enables a variant that stores the histograms in the scratchpad, which we show in Figure 5.5. Without scratchpad atomics, we fall back to Partitioned Global.

Partitioned Private. Partitioned Private provides each thread with its own copy of the histogram residing in scratchpad memory, as depicted in Figure 5.6. Partitioned Private incurs no contention over buckets in case of a small number of clusters, in contrast to the Partitioned Local strategy. However, Partitioned Private incurs a higher merging overhead than the previous strategies and also requires more space in scratchpad memory. This strategy outperforms the other approaches if k is small.

Summary. To summarize, a Multi-Pass strategy for the GPU requires an efficient centroid update on GPUs. In our analysis, we separate the centroid update into logically distinct parts, Feature Sum and Mass Sum. In Feature Sum, we address the large working set of Cluster Merge by introducing the space-efficient Partitioned Features strategy. For Mass Sum, Partitioned Private and Partitioned Local operate in the processor cache, but have different trade-offs. With these improvements, our centroid update is a cache-efficient, Multi-Pass k-means strategy for GPUs.



(a) Multi-pass strategy. (b) Single-pass strategy.

Figure 5.7: Fusing point assignment and centroid update by synchronizing the thread block.

5.3 Single-Pass GPU *k*-Means

All state-of-the-art *k*-means algorithms on GPUs compute point assignment and centroid update in two separate phases. During each phase, the point data is read from GPU memory, which leads to inefficient use of memory bandwidth. In this section, we show how both phases can be combined to process one iteration with a single pass over the data points. We illustrate the difference between single-pass and multi-pass execution in Figure 5.7.

Fusing point assignment and centroid update is challenging, because the phases prefer opposite data layouts for efficient data accesses. In point assignment, threads access individual points and thus prefer a row-wise layout. In contrast, during centroid update, threads access individual features of points and therefore prefer a column-oriented layout. To fuse these phases, we must decide on a single data layout. Our key idea is to cache a batch of data points in scratchpad memory and transpose it on-the-fly from a row-oriented to a column-oriented format. This presents two challenges: transposing the data layout of the points and efficiently synchronizing threads.

Transposing the Data Layout. Transposing data in cache requires sufficient space in scratchpad memory to store one point per thread (i.e., $4 \times t \times d$ bytes, with t threads). Additionally, there must be enough space left for the working set of the centroid update algorithm. The Partitioned Features strategy is essential for our idea to work on GPUs, as the working set of Partitioned Features is very small compared to Cluster Merge. Thus, we use the remaining space to transpose the data in scratchpad memory. Concretely, the Cluster Merge strategy with Partitioned Private (Mass Sum) uses $4t(kd + k + d)$ bytes of scratchpad memory, whereas the Partitioned Features strategy uses $4t(2k + d + 1)$ bytes, assuming t threads per thread block.

Block-wise Synchronization. Threads in Partitioned Features write to distinct

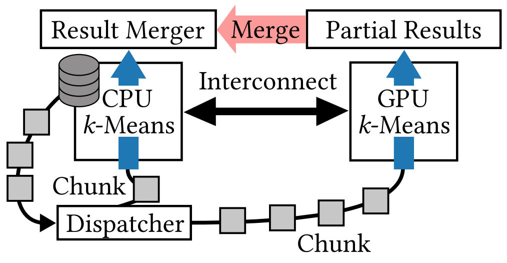


Figure 5.8: Support large data sets by streaming data to the GPU in chunks.

features of the new centroids. However, because of the transpose, each thread processes multiple points. Thus, each thread reads labels computed by other threads. In our solution, threads exchange labels only within a thread block. To this purpose, we synchronize the thread block with a thread barrier between point assignment and centroid update, as depicted in Figure 5.7(b).

Summary. In summary, the key to fusing the point assignment and the centroid update is to employ the Partitioned Features strategy (Feature Sum) and Partitioned Private strategy (Mass Sum) as the centroid update algorithm. Only with enough free cache space are we able to efficiently transpose the data layout on-the-fly.

5.4 Supporting Large Data Sets

In Figure 5.8, we extend our GPU-based execution strategies to handle data sets larger than GPU memory. We divide the point data into chunks, such that at least two chunks fit into GPU memory. Then, a dispatcher selects and transfers chunks to GPU memory via the interconnect. In parallel to ongoing transfers, the GPU computes a k-means iteration for each chunk, and adds the output to a partial result. This intermediate result resides in GPU memory and consists of partial sums and counts per cluster, as previously described in Feature Sum and Mass Sum. Optionally, the CPU also processes chunks. In this case, as multiple processors are used, chunks residing in the memory attached to a processor can be processed in-place using operator placement [78]. After all chunks are processed, we merge the intermediate results of the CPU and the GPU summing the intermediates and dividing final sums to obtain new centroids.

In contrast to previous chunking approaches [252, 395], we aggregate the results of chunks locally on the GPU, instead of transferring each individual result to CPU memory. Thus, we merge results in parallel on the GPU and reduce transfer overhead.

5.5 Evaluation

In this section, we evaluate our k-means strategies. We review our setup in Section 5.5.1. In Section 5.5.2, we present our results, and discuss them in Section 5.5.3.

5.5.1 Setup and Configuration

First, we introduce our methodology and experimental setup. After that, we describe our applied hardware tuning settings and data set. Finally, we introduce the experiments that we use to evaluate our strategies.

Methodology and Environment. We evaluate our implementation on a CPU and a GPU. Our implementation is based on OpenCL and we measure runtime of OpenCL kernels. CPU experiments are conducted on an Intel Core i7-6700K ("Skylake") CPU running at 3.4 GHz with 4 cores, 2-way SMT, and 32 GiB memory. In our GPU experiments, we use an Nvidia GeForce GTX 1080 ("Pascal") with 8 GiB memory. The test machine runs Ubuntu 16.04 LTS. We use the Intel OpenCL Runtime 16.1.1 for CPU code, and the Nvidia OpenCL 1.2 (CUDA 8.0.0) runtime for GPU code. Unless stated otherwise, on the GPU we exclude data transfer time to dedicated memory to avoid biased observations of execution time.

Hardware Tuning. To reach peak performance on each processor, we evaluate every OpenCL kernel with different tuning settings. We optimize our GPU code to employ the scratchpad memory and a coalesced memory access pattern. In contrast, the CPU implementation avoids the memory copy to OpenCL local memory and uses a sequential access pattern. We tune the vector lengths and thread block sizes. We format the input data in a column-oriented layout.

Data Sets. Our measurements use synthetically generated data sets following Arthur and Vassilvitskii [39]. Thus, we sample 10 centroids using a uniform random distribution in a hypercube, where each dimension ranges from -100 to 100. For each centroid, we sample an equal number of points from a normal distribution in a radius of 10 around the centroid. We measure throughput on 2 GiB data sets with varying k and d values. The 2 GiB data set size amortizes runtime system overheads while minimizing experiment runtime. We set the number of features as $d \in \{2, 4, 8, \dots, 256\}$ and adjust the number of points N such that data size remains constant.

Experiments. We conduct seven experiments that investigate the scalability of parameters on GPUs. The first and second experiment are microbenchmarks to determine the best strategies for Feature Sum and Mass Sum. The third experiment compares different k-means strategies and breaks down execution times for each strategy. Experiments

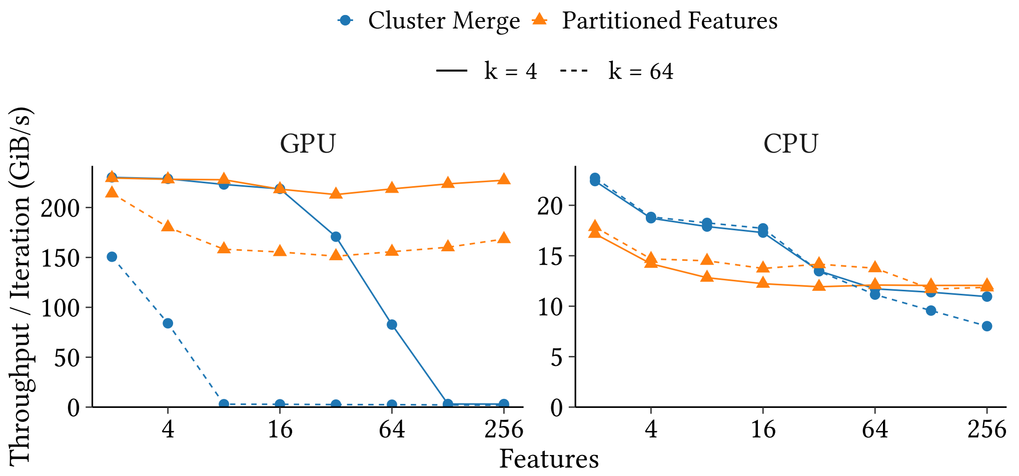


Figure 5.9: Comparison of Feature Sum strategies.

four and five examine the scaling behavior of each strategy while varying either the *k* or the *d* values. We transfer varying chunk sizes over PCI-e in experiment six. The last experiment investigates scalability for datasets that exceed the GPU memory capacity.

We choose these experiments because they explore the parameter space and analyze the trade-offs of different strategies. We report the mean and standard deviation (if above 5%) over 30 iterations.

Baselines. We compare our results to two open source frameworks, Armadillo [362] version 8.400 on the CPU and Rodinia [95] version 3.1 on the GPU. Armadillo is a linear algebra library for C++. It uses a Single-Pass *k*-means strategy on the CPU with OpenMP multi-threading support. We configure 8 threads. Rodinia features a CUDA *k*-means benchmark that uses the Cross-Processing strategy. Due to limited hardware texture memory and lack of data streaming support, Rodinia is constrained to a 512 MiB data size. We extrapolate results to 2 GiB. In addition to Rodinia, we have implemented our own Cross-Processing strategy optimized for our hardware.

5.5.2 Results

In this section, we present our evaluation results.

Feature Sum Strategies

In Figure 5.9, we compare the Cluster Merge strategy proposed by Li et al. [252] to compute Feature Sums with our Partitioned Features strategy. We investigate the number

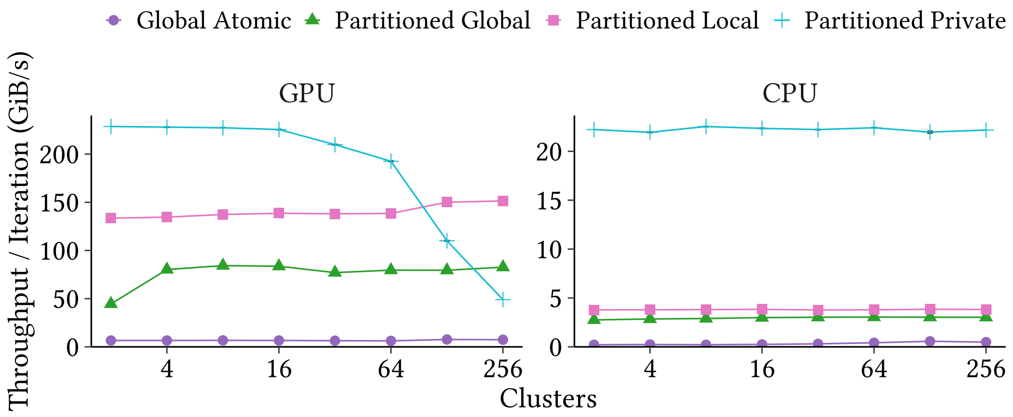


Figure 5.10: Comparison of Mass Sum strategies.

of features and the number of clusters, because both parameters impact performance and size of the working set. We set $k = 4$ and $k = 64$. To keep data size constant, N decreases for higher values of d.

Observations on GPU. Cluster Merge and Partitioned Features show equal throughput if both parameters are set to small values, i.e., $k = 4$ and $d \leq 16$. However, Partitioned Features outperforms Cluster Merge when $k = 64$ or $d > 16$. The reason is that less thread blocks fit into scratchpad memory, because Cluster Merge's working set grows with higher values of k and d. Consequently, the GPU's hardware scheduler runs less thread blocks (i.e., lower *occupancy*), thus reducing parallelism. If $k \times d > 384$, Cluster Merge falls back to GPU memory, as the working set of a single thread block does not fit into scratchpad memory. For Partitioned Features, we observe a small drop in performance for increasing k and d.

Observations on CPU. The throughput of Cluster Merge halves for both settings of k and continues to drop for $k = 64$. Partitioned Features is initially slower than Cluster Merge. However, Partitioned Features does not fall back to L2 cache for $k = 64$, because features are partitioned over multiple threads. Thus, Cluster Merge and Partitioned Features have similar performance on CPUs.

Summary. Overall, our Partitioned Features strategy scales to a higher number of features than Cluster Merge on the GPU.

Mass Sum Strategies

We evaluate the throughput of different Mass Sum strategies for a varying k in Figure 5.10. Since Mass Sum accesses only labels, d has no impact on performance.

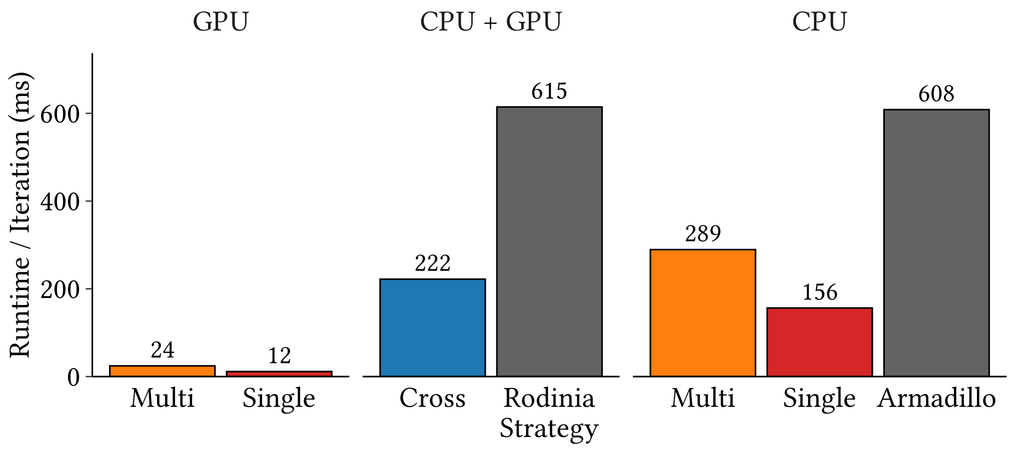


Figure 5.11: Comparison of execution times between k-means strategies on CPU and GPU for 4 features and 4 clusters.

Observations on GPU. Partitioned Private performs best until $k = 16$. After this point, the growing working set decreases the GPU occupancy, which decreases throughput. Starting from $k = 128$, Partitioned Local outperforms Partitioned Private. Partitioned Global's throughput increases above $k = 4$, as write contention is lower. Global Atomic achieves very low throughput due to write contention involving thousands of threads.

Observations on CPU. Partitioned Private has sufficient cache space for a stable throughput. In contrast, Partitioned Local and Global achieve five times lower throughput than Partitioned Private. Global Atomic also has low throughput, despite having less write contention than on the GPU.

Summary. Partitioned Private achieves the highest peak throughput on both processors. However, Partitioned Local scales to a higher number of clusters on the GPU.

Runtime Performance

In this experiment, we compare the overall execution times of k-means strategies per processor and break down the individual execution times. In Figure 5.11, we show the execution times of all strategies for $k = 4$ and $d = 4$. For strategies with multiple phases, we show the relative time spent per phase in percent in Figure 5.12.

Strategies. From the previous experiments, we learn that the Partitioned Feature (Feature Sum) and the Partitioned Private (Mass Sum) strategies combine to a fast centroid update. We derive three major strategies which we compare. The *Cross-Processing strategy* performs the point assignment on the GPU and performs the centroid update on the CPU, transferring the labels to the CPU between steps (state-of-the-art). The *Multi-Pass*

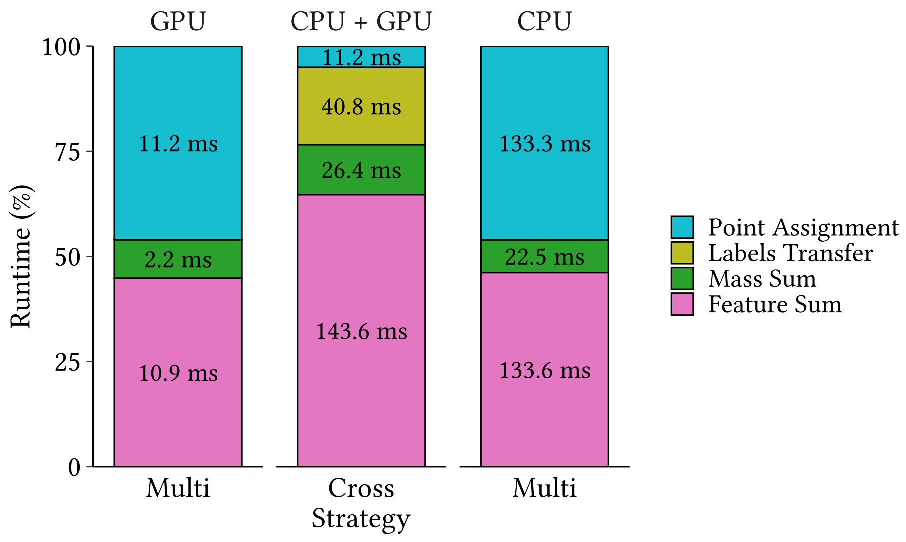


Figure 5.12: Execution time breakdowns for Cross-Processing and Multi-Pass on CPU and GPU for 4 features and 4 clusters.

strategy uses our fast centroid update and performs the point assignment and centroid update on the same processor (Section 5.2). The *Single-Pass strategy* fuses our centroid update routine with the point assignment to processes one k-means iteration with a single pass over the data points (Section 5.3).

Observations. The Cross-Processing strategy is dominated by the Feature Sum and label transfers from the GPU to the CPU. On the CPU, the Multi-Pass strategy has the highest execution time and is dominated by point assignment and Feature Sum. The Single-Pass strategy halves the execution time because it needs to read the data points only once. On the GPU, the Multi-Pass strategy outperforms the Cross-Processing strategy by 9.1×. The Single-Pass strategy improves the performance by 2× compared to the Multi-Pass strategy and by 19.3× compared to the Cross-Processing strategy.

Summary. Our Single-Pass strategy reduces memory accesses and is thus able to outperform the other strategies by more than 1.8× on all processors.

Scaling Clusters

In Figure 5.13, we investigate the impact of different numbers of clusters k on the throughput.

Observations on GPU. The Single-Pass strategy outperforms the Multi-Pass strategy for $k \leq 16$ by up to 2×. However, the Multi-Pass strategy outperforms the Single-Pass strategy starting from $k = 32$. Both strategies converge to the same performance of the

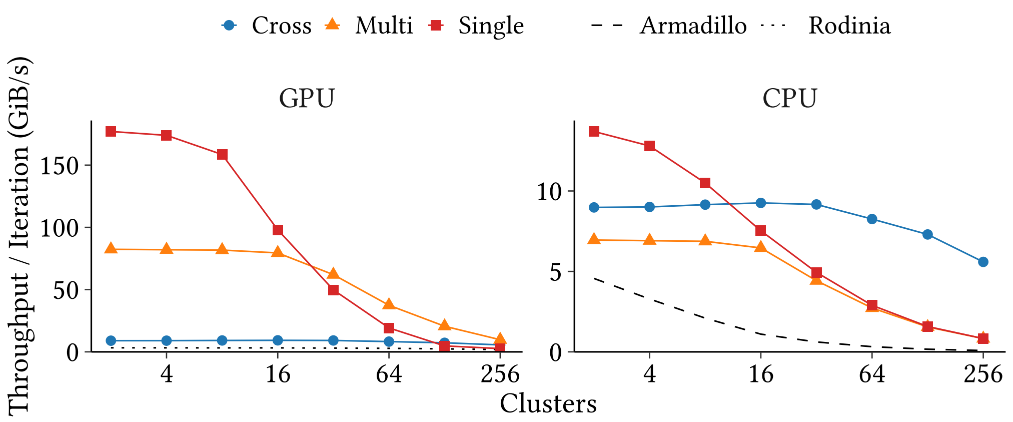


Figure 5.13: Performance of strategies for varying number of clusters on CPU and GPU with 4 features.

Cross-Processing strategy starting from $k = 128$. This is because the computational intensity of point assignment increases with growing k.

Observations on CPU. The Single-Pass strategy outperforms the Cross-Processing strategy until $k \geq 16$. Starting from this point, the single and Multi-Pass strategies have a similar performance and are outperformed by the Cross-Processing strategy. The reason is that the Cross-Processing strategy exploits the GPU to run point assignment.

Summary. Overall, throughput generally decreases as k increases. However, our GPU Multi-Pass strategy retains a speedup of at least 1.8× over Cross-Processing.

Scaling Features

We scale the number of features d and measure the throughput in Figure 5.14. We set the number of clusters to $k = 4$. As we keep data size constant with scaling d, the number of data points N halves with every doubling of d. Thus, the computational intensity of point assignment and Feature Sum does not change. However, the computational intensity of Mass Sum scales with $\frac{1}{d}$.

Observations on GPU. Our Single-Pass strategy outperforms the Multi-Pass strategy for $d \leq 32$ by 1.1–2.1×. For greater d, the throughput of both strategies is similar and is more than 3.4× faster compared to Cross-Processing.

Observations on CPU. The Single-Pass strategy consistently outperforms the Multi-Pass strategy and the Armadillo baseline.

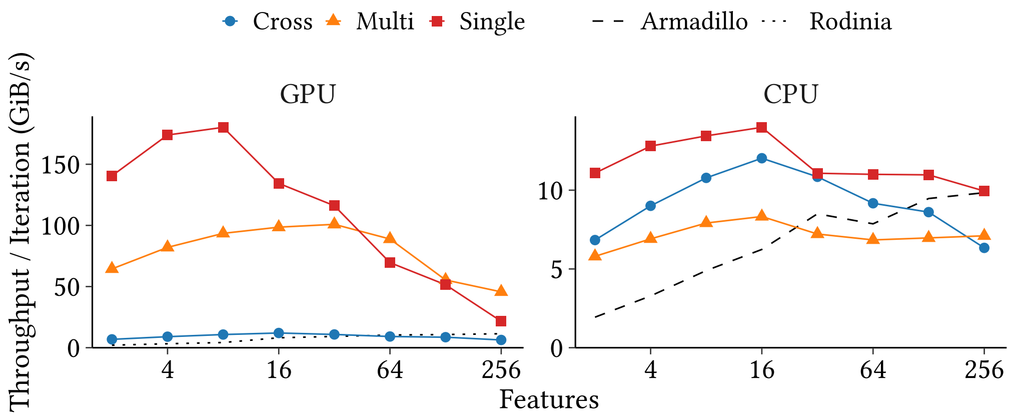
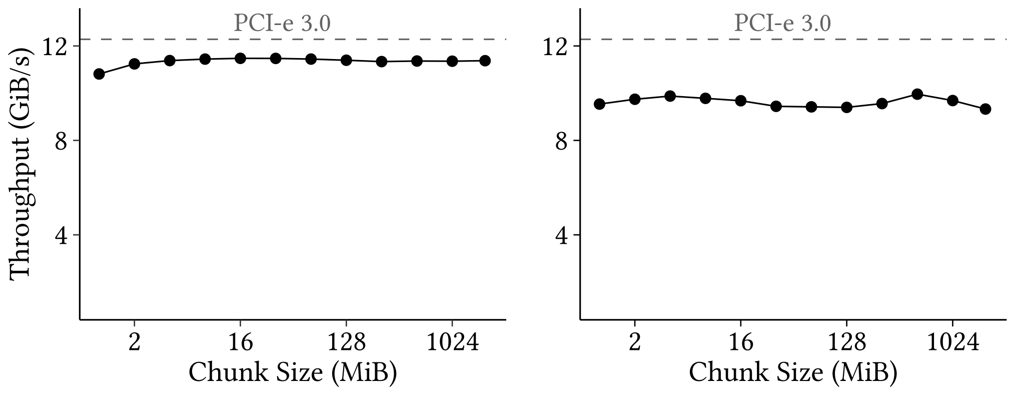


Figure 5.14: Performance of strategies for varying number of features on CPU and GPU with 4 clusters.



(a) Without Memcpy.

(b) With Memcpy.

Figure 5.15: Vary chunk sizes in transfer from main memory to GPU, performing (a) only transfer, and (b) memcpy—transfer—execute pipeline.

Chunk Transfers

We investigate the impact of chunk sizes on our chunk-wise transfer strategy in two parts. First, we observe only the transfer from main memory to GPU memory in Figure 5.15(a). Next, in Figure 5.15(b), we observe a three-stage pipeline. In this pipeline, each chunk is copied into a pinned buffer in CPU memory, then transferred to GPU memory over PCI-e 3.0, and finally processed on the GPU. To measure the transfer bandwidth, we call an empty GPU function on each chunk. We show the mean and standard deviation (if above 5%) over 100 transfers. As an upper bound, we measure the maximum bandwidth using the CUDA bandwidth utility.

Observations. The transfer without `memcpy` (up to 11.4 GiB/s) nearly reaches the throughput limit of 12.2 GiB/s. We achieve maximum throughput with chunk sizes between 8 and 64 MiB. In contrast, when running the complete pipeline, we observe a maximum throughput of 9.8 GiB/s with chunk sizes of 4, 8, 16, and 512 MiB. However, all measurements are within 5% of the maximum observation.

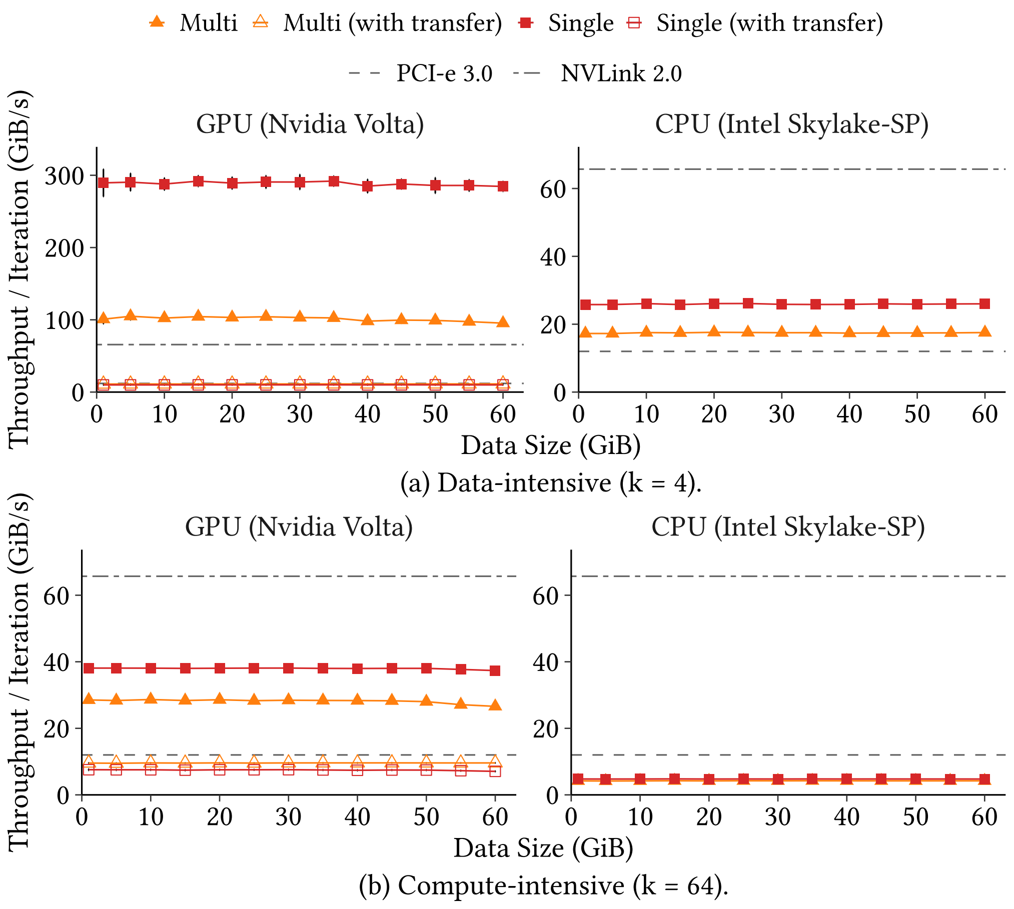
Thus, we conclude that main memory copies slow down throughput by 20%. In contrast, chunk sizes have only a small impact on overall PCI-e throughput.

Data Scaling

In Figure 5.16, we investigate the scalability of our Single-Pass and Multi-Pass strategies in the case when data exceeds the size of GPU memory. We run the experiment on recent hardware, using an Intel Xeon Gold 6126 (“Skylake-SP”) CPU with 12 cores at 2.6 GHz and an Nvidia Tesla V100-PCIE (“Volta”) GPU connected via PCI-e 3.0. We scale the size of point data from 1 GiB to 60 GiB and consider data-intensive ($k = 4$) as well as compute-intensive ($k = 64$) scenarios with $d = 4$ features. We configure 16 MiB chunks, based on our results in Section 5.5.2. To emphasize the transfer process, we transfer all data to the GPU from CPU memory and do not cache any data in GPU memory.

Observations on GPU. Our Single-Pass and Multi-Pass strategies always cluster points with the full PCI-e bandwidth. This is possible because we completely overlap data transfer with computation and our strategies complete the clustering before the next chunk arrives. We also show the kernel execution times of both strategies and relate them to the bandwidths of PCI-e 3.0 (measured) and the novel NVLink 2.0 (projected[1]) interconnects. In the data-intensive case, both strategies are able to fully utilize the interconnect bandwidth of PCI-e 3.0 and NVLink 2.0. In contrast, for a compute-intensive k, both strategies saturate PCI-e 3.0, but neither would saturate NVLink 2.0. However,

[1]Nvidia does not support OpenCL on IBM POWER platforms, thus we could not measure NVLink.



(a) Data-intensive (k = 4).

(b) Compute-intensive (k = 64).

Figure 5.16: Performance of strategies for increasing data size on CPU and GPU with (a) 4 and (b) 64 clusters, and 4 features.

our Single-Pass strategy achieves 58% of NVLink's bandwidth, whereas the Multi-Pass strategy reaches only 43%.

Observations on CPU. In the compute-intensive scenario, the GPU is 1.5× faster than the CPU. However, when data-intensive, the CPU outperforms the GPU by 2.6× due to the PCI-e 3.0 bandwidth limit. In contrast, NVLink 2.0 would enable the GPU to achieve a speedup of 2.5× over the CPU.

Summary. Overall, the GPU is bound by the data transfer bottleneck when connected by PCI-e 3.0. The GPU would only be capable of processing data faster than the CPU when we assume an NVLink 2.0 interconnect. Overall, we show the feasibility of GPU co-processing for data-intensive algorithms.

5.5.3 Discussion

In this chapter, we have investigated how a single-pass strategy is able to scale *k*-means to large, out-of-core data sets. We summarize our lessons learned.

Centroid Update Strategies. In our experiments, we showed that the Cluster Merge strategy of Feature Sum has a performance difference of two orders-of-magnitude between $d = 2$ and $d = 256$. In the worst case, Feature Sum is 4.6× slower on GPU than on CPU, thus motivating the Cross-Processing strategy. In contrast, our Partitioned Features strategy improves such cases by up to 96.7× on the GPU. Furthermore, we showed that, unlike on CPU, there is no single, best Mass Sum strategy on GPU. Rather, Partitioned Private performs up to 1.7× faster than Partitioned Local while $k \leq 64$. Falling back from the scratchpad to GPU memory incurs a penalty of 1.8–3×. In sum, Partitioned Features and Partitioned Private/Local lay the foundation for efficient centroid update on GPUs.

Runtime Performance. In analyzing *k*-means as a whole, we discovered that the Cross-Processing strategy (i.e., centroid update on the CPU) is often more than ten times slower compared to the Multi-Pass strategy (i.e., centroid update on the GPU). The main reason is the aforementioned slow centroid update in combination with the cross-processing problem. Furthermore, avoiding the multi-pass problem yields another 2× speedup between Multi-Pass and Single-Pass strategies on both processor types.

Parameter Scalability. We further observed that parameters are impacted unequally. When scaling the number of clusters *k*, computational intensity increases, which transforms *k*-means from a memory-bound algorithm for $k \leq 32$ to a compute-bound algorithm for larger *k*. Even though the computational intensity remains constant when scaling the number of features *d* for a fixed *k*, performance decreases. If the cache footprint of point assignment grows, the GPU occupancy decreases and point assignment eventually falls back to GPU memory.

Data Scalability. Our experiments highlight that GPU performance is bounded by the data transfer bottleneck. Thus, a fast interconnect is required for the GPU to consistently provide a speedup over a CPU. Our Single-Pass k-means strategy achieves a higher interconnect utilization than other strategies when considering a fast interconnect. However, more efficient point assignment techniques will be needed to overcome the high computational intensity of large k parameters.

5.6 Related Work

In the following, we contrast our k-means strategies to related work.

Intermediate Results. While early GPU-based k-means implementations where limited to OpenGL [85, 373], recent languages such as OpenCL or CUDA are designed for computational tasks and enable advanced optimizations. In particular, these languages allow us to avoid materializing intermediate results.

Point Assignment. Avoiding materialization is especially relevant for assigning points to clusters, because it is space- and memory-intensive to materialize all point-to-centroid distances. This can be further optimized through caching centroids [139] and data points [252] in scratchpad memory or the L1 cache. Our point assignment phase builds on this approach and adds processor-specific optimizations.

Optimizations that reduce the theoretical computational complexity of point assignment emphasize our work, because high k-values do not make k-means computation-bound. Specifically, Hall and Hart [166] apply Elkan's kd-tree approach [132] to a GPU implementation. They propose to store centroids in a kd-tree, such that finding the nearest centroid requires less comparisons for large k. These optimizations are orthogonal and complementary to our work.

Centroid Update. Updating centroids on GPUs [252] or MICs [245] has been proposed previously. We evaluate this Multi-Pass strategy for a wide range of cluster and feature parameters. In particular, we show that Cluster Merge is inefficient on GPUs and introduce new optimizations with our cache-efficient Partitioned Features strategy.

Feature Sum. Regarding the individual parts of centroid update, researchers proposed different solutions using GPUs. First, Feature Sum was implemented by using SQL group-by aggregation [178, 320], which has been implemented on CPUs [283] and GPUs [170, 219]. However, Feature Sum in k-means and relational-style group-by aggregation differ significantly. In particular, having tens or hundreds of features is common in data sets, but aggregating over this many attributes is uncommon in relational queries (e.g., TPC-H [1]). Thus, to the best of our knowledge, we are the first who optimize co-processor group-by aggregation for this use case.

Mass Sum. Second, Mass Sum on the GPU could exploit histogram computation [292]. These general implementations sort pixel values into buckets, which requires, e.g., divisions or branches. However, frequent branch divergence reduces performance on GPUs [386]. In contrast, in our Mass Sum strategies, labels directly index into buckets and thus avoid branches. Furthermore, our Partitioned Local strategy uses hardware-native atomic operations in scratchpad memory, which were emulated in software at the time of previous work [298].

Cross Processing. GPUMiner [137] reduces transfer overhead for the labels in the Cross-Processing strategy with bitmap compression. In contrast, we update centroids directly on the GPU with our Multi-Pass and Single-Pass strategies to eliminate this source of overhead entirely.

Single Data Pass. CPU implementations that compute k-means in a single data pass exist [276, 362]. However, on GPUs, we require a different approach because our Single-Pass strategy must reshuffle data between threads on-the-fly.

Data Transfers. Transferring data from CPU memory to the GPU has been found to have negligible overhead [44, 416]. In contrast, our results show that efficient k-means implementations for recent GPUs can incur a data transfer bottleneck. Thus, we propose to increase transfer bandwidth by utilizing a fast interconnect.

5.7 Conclusion

In this chapter, we propose a GPU-optimized algorithm for k-means. Our algorithm centers around a highly-optimized strategy for updating centroids on GPUs. In our algorithm, we solve two fundamental problems of previous approaches: *cross-processing* and *multi-pass execution*. In contrast to previous approaches, we focus on reducing cache space usage through architectural features of GPU hardware, such that we are able to increase the effective parallelism. As a result, we propose a highly-optimized strategy for k-means that runs entirely on a GPU and requires only a single pass over the data. The evaluation shows that the Single-Pass strategy achieves up to 2× and 20× higher throughput than the Multi-Pass and Cross-Processing strategies, respectively. We show that our approach scales to large data sets exceeding the GPU memory capacity. In our experiments, our GPU strategies perform at least as well as a CPU despite transferring data over PCI-e 3.0. Finally, our Single-Pass strategy is the only strategy capable of saturating the bandwidth of the NVLink 2.0 interconnect for data-intensive scenarios.

6 Conclusion

For a decade, researchers have pursued shorter query response times by processing data on GPUs. With fast interconnects becoming commercially available, the success of data management on GPUs must no longer be thwarted by the data transfer bottleneck. We advocate redesigning DBMSs to process data out-of-core by exploiting fast interconnects.

In this dissertation, we have investigated the ways in which fast interconnects benefit DBMSs through their high bandwidth and cache-coherence (Chapter 3). With our Triton join, we have demonstrated that it is possible to efficiently join large relations on GPUs by spilling the join state to CPU memory (Chapter 4). With our single-pass *k*-means strategy, we have shown how a machine learning algorithm is able to efficiently iterate over a large data set multiple times (Chapter 5). Thus, by enhancing GPUs with fast interconnects, relational and machine learning scale to large data volumes stored in CPU memory, and improve performance by up to 2.5–7.3× compared to a CPU.

During the course of this thesis, we have learned that high out-of-core performance is attainable by designing new algorithms to be interconnect-conscious. Our insights have come from pushing data management workloads to the limits of the hardware and thereby evoking unanticipated performance behavior. A better understanding of the hardware has led us to, i.a., improve irregular memory access bandwidth, freely manage pageable memory, and orchestrate data transfers from within GPU kernels. As a result, we have broadened the design space for out-of-core algorithms, which in turn has allowed us to improve throughput. Overall, we conclude that future GPU-enabled, interconnect-conscious DBMSs will be well-suited for large-scale data management.

6.1 Research Outlook

This dissertation lays the foundation for research on scalable data management using GPUs with fast interconnects. In the following, we discuss five open research challenges.

DBMS Design. In our research, we have focused on joins and *k*-means to show that principle limitations of data management on GPUs can be solved using fast interconnects. Future work could broaden our findings to other relational operators such as selections, group-by aggregations, and set operators. We are currently investigating interconnect-conscious index structures to scale, e.g., highly selective queries, range queries, outer joins, and inequality joins. Moreover, placing operators across heterogeneous processors could exploit the sequential-access CPU memory bandwidth of the CPU and the random-access GPU memory bandwidth, which we have explored in a Master's thesis [342]. Our heterogeneous, morsel-driven work scheduling approach could be extended to operator pipelines, whereby the GPU could itself schedule work using native atomics. The trade-offs inherent to data compression should be reevaluated for fast interconnects and hardware-accelerated compression [4, 351]. These innovations should be combined to improve the overall query performance of a DBMS.

Data Streaming. Fast interconnects are particularly interesting for data stream management due to the inherent network I/O. Modern network interface cards can achieve a bandwidth comparable to that of CPU memory, and thus require new approaches to ingest and process data [235, 427]. As individual real-world streams are unlikely to reach these speeds, taking advantage of the hardware would involve either joining and unioning thousands of streams or processing many queries in parallel, which would necessitate new algorithms that efficiently scale to a large number of streams. High data velocities could result in large windows that require spilling a large state to CPU memory via the interconnect. Furthermore, fast interconnects can also integrate network interface cards into the system, which provides new research opportunities [353].

Deep Learning. Recent neural network models such as BERT [123] and GPT-3 [80] are affected by the data transfer bottleneck due to their large size. In effect, parameter servers must scale to a large model size with fast response times to parameter queries. A new out-of-core parameter server could scale the model size by applying our insights into data transfer methods and data caching.

Data Loading. We notice that the data load time is typically not measured in research publications. However, benchmarks such as TPC-H reflect that real-world workloads often bulk load data before executing queries [1, §4.3]. Loading converts the data format from, e.g., CSV, to a DBMS-specific format. In a Master's thesis and subsequent publication, we have shown promising results by leveraging GPUs to reduce the data load time [234, 235]. Future work could revisit our approach to load multiple data streams in parallel, create indices during loading, and validate the input format to guard against errors. Ultimately, this line of research would result in fast end-to-end query performance.

Memory Hierarchy. Non-volatile memory and flash disks could be investigated to scale the data volume beyond CPU memory. Fast interconnects provide GPUs access to these storage technologies, which present additional challenges due to their read and write characteristics. In contrast, disaggregated memory scales to petabytes of space as well, but might involve complex data access paths [109, 394]. Instead of GPUs with fast interconnects, CPUs with on-board high bandwidth memory could provide an interesting alternative solution [61, 144, 426].

Fast Interconnect Technologies. In our work, we have evaluated NVLink 2.0 on account of its current commercial availability. However, fast interconnects from multiple hardware vendors are currently available for FPGAs (e.g., OpenCAPI) and announced for GPUs in the near future (i.e., NVLink 4.0, Infinity Fabric, and CXL). We have provided an overview of these technologies in Section 2.3.3. Future work could evaluate and contrast the upcoming fast interconnects in practice.

In summary, we envision that future work will evolve GPU-enabled DBMSs alongside the fast interconnects and will continue to expand the scope of GPUs in data management.

List of Figures

List of Tables

List of Listings

Bibliography

[1] 2017. *Transaction Processing Performance Council. TPC-H.* http://www.tpc.org/tpch

[2] AMD 2019. *AMD EPYC CPUs, AMD Radeon Instinct GPUs and ROCm open source software to power world's fastest supercomputer at Oak Ridge National Laboratory*. AMD. Retrieved July 5, 2019 from https://www.amd.com/en/press-releases/2019-05-07-amd-epyc-cpus-radeon-instinct-gpus-and-rocm-open-source-software-to-power

[3] Tor M. Aamodt, Wilson W. L. Fung, and Ali Bakhoda. 2022. GPGPU-Sim. Retrieved Jan 3, 2022 from https://github.com/gpgpu-sim/gpgpu-sim_distribution

[4] Bülent Abali, Bart Blaner, John J. Reilly, Matthias Klein, Ashutosh Mishra, Craig B. Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J. Starke, Haren Myneni, and Charlie Wang. 2020. Data compression accelerator on IBM POWER9 and z15 processors. In *ISCA*. IEEE, Washington, DC, USA, 1–14. https://doi.org/10.1109/ISCA45697.2020.00012

[5] Neha Agarwal, David W. Nellans, Eiman Ebrahimi, Thomas F. Wenisch, John Danskin, and Stephen W. Keckler. 2016. Selective GPU caches to eliminate CPU-GPU HW cache coherence. In *HPCA*. IEEE Computer Society, Los Alamitos, CA, USA, 494–506. https://doi.org/10.1109/HPCA.2016.7446089

[6] Neha Agarwal, David W. Nellans, Mike O'Connor, Stephen W. Keckler, and Thomas F. Wenisch. 2015. Unlocking bandwidth for GPUs in CC-NUMA systems. In *HPCA*. IEEE Computer Society, Los Alamitos, CA, USA, 354–365. https://doi.org/10.1109/HPCA.2015.7056046

[7] Jasmin Ajanovic. 2009. PCI express 3.0 overview. In *HCS*. IEEE, Washington, DC, USA, 1–61. https://doi.org/10.1109/HOTCHIPS.2009.7478337

[8] Dan A. Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. 2012. Building an efficient hash table on the GPU. In *GPU computing gems* (Jade ed.). Morgan Kaufmann, Waltham, MA, USA, Chapter 4, 39–53. https://doi.org/10.1016/B978-0-12-385963-1.00004-6

[9] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU concurrency: Weak behaviours and programming assumptions. In *ASPLOS*. ACM, New York, NY, USA, 577–591. https://doi.org/10.1145/2694344.2694391

[10] Alpha Data 2019. *ADM-PCIE-9H7 datasheet*. Alpha Data. https://www.alpha-data.com/xml/product_datasheets/adm-pcie-9h7_v2.1.pdf Revision 2.1.

[11] Johnathan Alsop, Matthew D. Sinclair, and Sarita V. Adve. 2018. Spandex: A flexible interface for efficient heterogeneous coherence. In *ISCA*. IEEE Computer Society, Los Alamitos, CA, USA, 261–274. https://doi.org/10.1109/ISCA.2018.00031

[12] Amazon AWS. 2016. *AWS Snowmobile — Migrate or transport exabyte-scale data sets into and out of AWS*. Retrieved Feb 2, 2022 from https://aws.amazon.com/snowmobile

[13] AMD 2012. *AMD Graphics Core Next GCN architecture*. AMD. https://www.techpowerup.com/gpu-specs/docs/amd-gcn1-architecture.pdf

[14] AMD 2015. *Asynchronous shaders: Unlocking the full potential of the GPU*. AMD. https://developer.amd.com/wordpress/media/2012/10/Asynchronous-Shaders-White-Paper-FINAL.pdf

[15] AMD 2017. *Radeon's next-generation Vega architecture*. AMD. https://en.wikichip.org/w/images/a/a1/vega-whitepaper.pdf

[16] AMD 2018. *GCN ISA manuals: PCIe features*. AMD. Retrieved Feb 22, 2022 from https://rocmdocs.amd.com/en/latest/GCN_ISA_Manuals/PCIe-features.html Git Revision 59db884.

[17] AMD 2019. *AMD RDNA architecture*. AMD. https://www.amd.com/system/files/documents/rdna-whitepaper.pdf

[18] AMD 2020. *AMD CDNA architecture*. AMD. https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf

[19] AMD 2020. *AMD Instinct MI100 instruction set architecture reference guide*. AMD. https://developer.amd.com/wp-content/resources/CDNA1_Shader_ISA_14December2020.pdf

[20] AMD 2020. *OpenCL optimization*. AMD. https://rocmdocs.amd.com/en/latest/Programming_Guides/Opencl-optimization.html Git Revision 1f057816.

[21] AMD 2020. *OpenCL programming guide*. AMD. https://rocmdocs.amd.com/en/latest/Programming_Guides/Opencl-programming-guide.html Git Revision 611e249.

[22] AMD 2021. *AMD CDNA 2 architecture*. AMD. https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf

[23] AMD 2021. *AMD EPYC 7003 series processors*. AMD. https://www.amd.com/system/files/documents/amd-epyc-7003-series-datasheet.pdf Revision LE-77202-00 02/21.

[24] AMD 2021. *AMD HIP programming guide*. AMD. https://raw.githubusercontent.com/RadeonOpenCompute/ROCm/rocm-4.5.2/AMD_HIP_Programming_Guide.pdf Version 1.0, Revision 1210.

[25] AMD 2021. *AMD Instinct MI200 instruction set architecture reference guide*. AMD. https://developer.amd.com/wp-content/resources/CDNA2_Shader_ISA_18November2021.pdf

[26] AMD 2021. *AMD Instinct MI200 series accelerator*. AMD. https://www.amd.com/system/files/documents/amd-instinct-mi200-datasheet.pdf

[27] AMD 2021. *AMD I/O virtualization technology (IOMMU) specification*. AMD. https://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf Revision 3.06-PUB.

[28] AMD 2021. *GPU performance API*. AMD. https://gpuperfapi.readthedocs.io/en/latest/ Git Revision 3642849d.

[29] AMD 2022. *AMD ROCm platform*. AMD. https://rocmdocs.amd.com Git Revision 20d619e.

[30] Ampere Computing 2022. *Ampere Altra Max datasheet*. Ampere Computing. https://amperecomputing.com/wp-content/uploads/2022/01/Altra_Max_Rev_A1_DS_v0.91_20220113.pdf Document issue 0.91.

[31] Guilherme Andrade, Gabriel Spada Ramos, Daniel Madeira, Rafael Sachetto Oliveira, Renato Ferreira, and Leonardo Rocha. 2013. G-DBSCAN: A GPU accelerated algorithm for density-based clustering. 18 (2013), 369–378. https://doi.org/10.1016/j.procs.2013.05.200

[32] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The case for heterogeneous HTAP. In *CIDR*. www.cidrdb.org, 1–9. http://cidrdb.org/cidr2017/papers/p21-appuswamy-cidr17.pdf

[33] Mohamed Arafa, Bahaa Fahim, Sailesh Kottapalli, Akhilesh Kumar, Lily Pao Looi, Sreenivas Mandava, Andy Rudoff, Ian M. Steiner, Bob Valentine, Geetha Vedaraman, and Sujal Vora. 2019. Cascade Lake: Next generation Intel Xeon Scalable processor. *IEEE Micro* 39, 2 (2019), 29–36. https://doi.org/10.1109/MM.2019.2899330

[34] Yehia Arafa, Abdel-Hameed A. Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan J. Eidenbenz. 2019. Low overhead instruction latency characterization for Nvidia GPGPUs. In *HPEC*. IEEE, Washington, DC, USA, 1–8. https://doi.org/10.1109/HPEC.2019.8916466

[35] L. Baba Arimilli, Bart Blaner, Ben C. Drerup, Charles F. Marino, Derek Williams, Eric N. Lais, Francesco A. Campisano, Guy L. Guthrie, Michael S. Floyd, Ross Leavens, Scot M. Willenborg, Ronald N. Kalla, and Bülent Abali. 2018. IBM POWER9 processor and system features for computing in the cognitive era. *IBM Journal of Research and Development* 62, 4/5 (2018), 1:1–1:11. https://doi.org/10.1147/JRD.2018.2859564

[36] Arm 2021. *AMBA 5 CHI architecture specification*. Arm. https://documentation-service.arm.com/static/611b9cf1674a052ae36c77aa Document number ARM IHI 0050E.b.

[37] Arm 2021. *Arm system memory management unit architecture specification*. Arm. https://documentation-service.arm.com/static/60804fe25e70d934bc69f12d Document number ARM IHI 0070.

[38] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-performance ACID table storage over cloud object stores. *PVLDB* 13, 12 (2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[39] David Arthur and Sergei Vassilvitskii. 2007. k-Means++: The advantages of

careful seeding. In *SODA*. SIAM, Philadelphia, PA, USA, 1027–1035. http://dl.acm.org/citation.cfm?id=1283383.1283494

[40] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P. Sadayappan. 2015. On optimizing machine learning workloads via kernel fusion. In *PPoPP*. ACM, New York, NY, USA, 173–182. https://doi.org/10.1145/2688500.2688521

[41] Saman Ashkiani, Shengren Li, Martin Farach-Colton, Nina Amenta, and John D. Owens. 2018. GPU LSM: A dynamic dictionary data structure for the GPU. In *IPDPS*. IEEE Computer Society, Los Alamitos, CA, USA, 430–440. https://doi.org/10.1109/IPDPS.2018.00053

[42] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU memory manager with application-transparent support for multiple page sizes. In *MICRO*. ACM, New York, NY, USA, 136–150. https://doi.org/10.1145/3123939.3123975

[43] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martin Farach-Colton, and John D. Owens. 2019. Engineering a high-performance GPU B-Tree. In *PPoPP*. ACM, New York, NY, USA, 145–157. https://doi.org/10.1145/3293883.3295706

[44] Hongtao Bai, Lili He, Dantong Ouyang, Zhan-Shan Li, and He Li. 2009. k-Means on commodity GPUs with CUDA. In *CSIE WRI*. IEEE Computer Society, Los Alamitos, CA, USA, 651–655. https://doi.org/10.1109/CSIE.2009.491

[45] Peter Bakkum and Kevin Skadron. 2010. Accelerating SQL database operations on a GPU with CUDA. In *GPGPU*, Vol. 425. ACM, New York, NY, USA, 94–103. https://doi.org/10.1145/1735688.1735706

[46] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB* 7, 1 (2013), 85–96. https://doi.org/10.14778/2732219.2732227

[47] Cagri Balkesen, Nitin Kunal, Georgios Giannikis, Pit Fender, Seema Sundara, Felix Schmidt, Jarod Wen, Sandeep R. Agrawal, Arun Raghavan, Venkatanathan Varadarajan, Anand Viswanathan, Balakrishnan Chandrasekaran, Sam Idicula, Nipun Agarwal, and Eric Sedlar. 2018. RAPID: In-memory analytical query

processing engine with extreme performance per watt. In *SIGMOD*. ACM, New York, NY, USA, 1407–1419. https://doi.org/10.1145/3183713.3190655

[48] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*. IEEE Computer Society, Los Alamitos, CA, USA, 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[49] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To partition, or not to partition, that is the join question in a real system. In *SIGMOD*. ACM, New York, NY, USA, 168–180. https://doi.org/10.1145/3448016.3452831

[50] Ronald Barber, Guy M. Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, Gopi K. Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. 2014. Memory-efficient hash joins. *PVLDB* 8, 4 (2014), 353–364. https://doi.org/10.14778/2735496.2735499

[51] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation caching: Skip, don't walk (the page table). In *ISCA*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/1815961.1815970

[52] Claude Barthels, Gustavo Alonso, Torsten Hoefler, Timo Schneider, and Ingo Müller. 2017. Distributed join algorithms on thousands of cores. *PVLDB* 10, 5 (2017), 517–528. https://doi.org/10.14778/3055540.3055545

[53] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-scale in-memory join processing using RDMA. In *SIGMOD*. ACM, New York, NY, USA, 1463–1475. https://doi.org/10.1145/2723372.2750547

[54] Trinayan Baruah, Yifan Sun, Saiful A. Mojumder, José L. Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, and David R. Kaeli. 2020. Valkyrie: Leveraging inter-TLB locality to enhance GPU performance. In *PACT*. ACM, New York, NY, USA, 455–466. https://doi.org/10.1145/3410463.3414639

[55] Michael Bauer, Henry Cook, and Brucek Khailany. 2011. CudaDMA: Optimizing GPU memory bandwidth via warp specialization. In *SC*. ACM, New York, NY, USA, 12:1–12:11. https://doi.org/10.1145/2063384.2063400

[56] Michael Bauer, Sean Treichler, and Alex Aiken. 2014. Singe: Leveraging warp specialization for high performance on GPUs. In *PPoPP*. ACM, New York, NY, USA, 119–130. https://doi.org/10.1145/2555243.2555258

[57] Steven Keith Begley, Zhen He, and Yi-Ping Phoebe Chen. 2012. MCJoin: A memory-constrained join for column-store main-memory databases. In *SIGMOD*. ACM, New York, NY, USA, 121–132. https://doi.org/10.1145/2213836.2213851

[58] Muli Ben-Yehuda, Jon Mason, Orran Krieger, Jimi Xenidis, Leendert Van Doorn, Asit Mallick, and Elsie Wahlig. 2006. Utilizing IOMMUs for virtualization in Linux and Xen. In *Proceedings of the Ottawa Linux Symposium*, Vol. 1. 71–86. https://www.kernel.org/doc/ols/2006/ols2006v1-pages-71-86.pdf

[59] Srikant Bharadwaj, Guilherme Cox, Tushar Krishna, and Abhishek Bhattacharjee. 2018. Scalable distributed last-level TLBs using low-latency interconnects. In *MICRO*. IEEE Computer Society, Los Alamitos, CA, USA, 271–284. https://doi.org/10.1109/MICRO.2018.00030

[60] Abhishek Bhattacharjee. 2017. Advanced concepts on address translation. In *Computer architecture — A quantitative approach* (6th ed.), John L. Hennessy and David A. Patterson (Eds.). Morgan Kaufmann, Cambridge, MA, USA, Appendix L, 1–69.

[61] Arijit Biswas. 2021. Sapphire Rapids: Next-gen Intel Xeon Scalable processor. In *HCS*. IEEE, Washington, DC, USA, 1–22. https://doi.org/10.1109/HCS52781.2021.9566865

[62] Mark Blacher, Joachim Giesen, Sören Laue, Julien Klaus, and Viktor Leis. 2022. Machine learning, linear algebra, and more: Is SQL all you need?. In *CIDR*. www.cidrdb.org, 1–6. https://www.cidrdb.org/cidr2022/papers/p17-blacher.pdf

[63] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1989323.1989328

[64] Bartholomew Blaner, Jay G. Heaslip, Robert D. Herzl, and Jody B. Joyner. 2020. Maintaining consistency between address translations in a data processing system. Patent No. US10599569B2, Filed Jun. 23th., 2016, Issued Mar. 24th., 2020.

[65] Bartholomew Blaner, Jody B. Joyner, Ronald N. Kalla, Jon K. Kriegel, and Charles D. Wait. 2018. Maintaining agent inclusivity within a distributed MMU. Patent App. No. US20180300256A1, Filed Apr. 13th., 2017.

[66] BlazingSQL. 2021. *BlazingSQL*. Retrieved Jan 28, 2022 from https://blazingsql.com

[67] Emily R. Blem, Jaikrishnan Menon, Thiruvengadam Vijayaraghavan, and Karthikeyan Sankaralingam. 2015. ISA Wars: Understanding the relevance of ISA being RISC or CISC to performance, power, and energy on modern architectures. *ACM Trans. Comput. Syst.* 33, 1 (2015), 3:1–3:34. https://doi.org/10.1145/2699682

[68] David Blythe. 2021. X^{e}HPC Ponte Vecchio. In *HCS*. IEEE, New York, NY, USA, 1–34. https://doi.org/10.1109/HCS52781.2021.9567038

[69] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. 2020. SystemDS: A declarative machine learning system for the end-to-end data science lifecycle. In *CIDR*. www.cidrdb.org, 1–8. http://cidrdb.org/cidr2020/papers/p22-boehm-cidr20.pdf

[70] Matthias Boehm, Alexandre V. Evfimievski, Niketan Pansare, and Berthold Reinwald. 2016. Declarative machine learning — A classification of basic properties and types. arXiv:1605.05826 [cs.DB]

[71] Matthias Boehm, Arun Kumar, and Jun Yang. 2019. *Data management in machine learning systems*. Synthesis Lectures on Data Management, Vol. 14. Morgan & Claypool Publishers, San Rafael, CA, USA. https://doi.org/10.2200/S00895ED1V01Y201901DTM057

[72] Nils Boeschen and Carsten Binnig. 2022. GaccO — A GPU-accelerated OLTP DBMS. In *SIGMOD*. ACM, New York, NY, USA, 1003–1016. https://doi.org/10.1145/3514221.3517876

[73] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*. www.cidrdb.org, 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[74] Rajesh Bordawekar and Pidad Gasfer D'Souza. 2018. Evaluation of hybrid cache-coherent concurrent hash table on IBM POWER9 AC922 system with NVLink2. http://on-demand.gputechconf.com/gtc/2018/video/S8172/. In *GTC*. Nvidia.

[75] Shekhar Borkar and Andrew A. Chien. 2011. The future of microprocessors. *Commun. ACM* 54, 5 (2011), 67–77. https://doi.org/10.1145/1941487.1941507

[76] Dan Bouvier and Ben Sander. 2014. Applying AMD's Kaveri APU for heterogeneous computing. In *HCS*. IEEE, New York, NY, USA, 1–42. https://doi.org/10.1109/HOTCHIPS.2014.7478810

[77] Sebastian Breß. 2014. The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum* 14, 3 (2014), 199–209. https://doi.org/10.1007/s13222-014-0164-z

[78] Sebastian Breß, Henning Funke, and Jens Teubner. 2016. Robust query processing in co-processor-accelerated databases. In *SIGMOD*. ACM, New York, NY, USA, 1891–1906. https://doi.org/10.1145/2882903.2882936

[79] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2018. Generating custom code for efficient query execution on heterogeneous processors. *VLDB J.* 27, 6 (2018), 797–822. https://doi.org/10.1007/s00778-018-0512-y

[80] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *NeurIPS*, Vol. 33. Curran Associates, Inc., Red Hook, NY, USA, 1877–1901. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[81] Kristin Bryson. 2021. *Nvidia announces CPU for giant AI and high performance computing workloads*. Nvidia. Retrieved Mar 23, 2022 from https://nvidianews.nvidia.com/news/nvidia-announces-cpu-for-giant-ai-and-high-performance-computing-workloads

[82] Brytlyt. 2022. *BrytlytDB*. Retrieved Jan 28, 2022 from https://www.brytlyt.com/what-we-do/brytlytdb

[83] Matt Burns. 2018. *Alpha Data announces 12x100g network accelerator board, featuring Samtec Twinax Flyover systems and Xilinx Ultrascale+ FPGA*. Alpha Data. Retrieved

Mar 1, 2022 from https://www.alpha-data.com/alpha-data-announces-12x100g-network-accelerator-board-featuring-samtec-twinax-flyover-systems-and-xilinx-ultrascale-fpga

[84] Alexandre Bicas Caldeira. 2018. *IBM power system AC922 introduction and technical overview* (1st ed.). IBM, International Technical Support Organization. https://www.redbooks.ibm.com/redpapers/pdfs/redp5472.pdf REDP-5472-00.

[85] Feng Cao, Anthony K. H. Tung, and Aoying Zhou. 2006. Scalable clustering using graphics processors. In *WAIM (LNCS, Vol. 4016)*. Springer Berlin Heidelberg, Heidelberg, Germany, 372–384. https://doi.org/10.1007/11775300_32

[86] Mathilde Caron, Piotr Bojanowski, Armand Joulin, and Matthijs Douze. 2018. Deep clustering for unsupervised learning of visual features. In *ECCV (LNCS, Vol. 11218)*. Springer, Cham, Switzerland, 139–156. https://doi.org/10.1007/978-3-030-01264-9_9

[87] Mathilde Caron, Ishan Misra, Julien Mairal, Priya Goyal, Piotr Bojanowski, and Armand Joulin. 2020. Unsupervised learning of visual features by contrasting cluster assignments. 33 (2020), 9912–9924. https://proceedings.neurips.cc/paper/2020/file/70feb62b69f16e0238f741fab228fec2-Paper.pdf

[88] Christophe Cassou. 2008. Intraseasonal interaction between the Madden–Julian Oscillation and the North Atlantic Oscillation. *Nature* 455, 7212 (Sept. 2008), 523–527. https://doi.org/10.1038/nature07286

[89] Adriana Grazia Castriotta and Federica Volpi. 2021. *Copernicus Sentinel data access annual report Y2020*. Retrieved Feb 2, 2022 from https://scihub.copernicus.eu/twiki/pub/SciHubWebPortal/AnnualReport2020/COPE-SERCO-RP-21-1141_-_Sentinel_Data_Access_Annual_Report_Y2020_final_v2.3.pdf

[90] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *MICRO*. IEEE Computer Society, 7:1–7:13. https://doi.org/10.1109/MICRO.2016.7783710

[91] CCIX Consortium 2019. *CCIX base specification revision 1.0a*. CCIX Consortium. Version 1.0.

[92] CCIX Consortium 2019. *An introduction to CCIX*. CCIX Consortium. https://www.ccixconsortium.com/wp-content/uploads/2019/11/CCIX-White-Paper-Rev111219.pdf

[93] Naveen Krishna Chatradhi. 2022. *Linux Kernel Mailing List*. AMD. Retrieved Feb 22, 2022 from https://lkml.org/lkml/2022/2/3/673

[94] Surajit Chaudhuri and Umeshwar Dayal. 1997. An overview of data warehousing and OLAP technology. *SIGMOD Rec.* 26, 1 (1997), 65–74. https://doi.org/10.1145/248603.248616

[95] Shuai Che et al. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*. IEEE Computer Society, Los Alamitos, CA, USA, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[96] Xinyu Chen, Yao Chen, Ronak Bajaj, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2020. Is FPGA useful for hash joins?. In *CIDR*. www.cidrdb.org, 1–9. http://cidrdb.org/cidr2020/papers/p27-chen-cidr20.pdf

[97] Xuntao Cheng, Bingsheng He, Xiaoli Du, and Chiew Tong Lau. 2017. A study of main-memory hash joins on many-core processor: A case with Intel Knights Landing architecture. In *CIKM*. ACM, New York, NY, USA, 657–666. https://doi.org/10.1145/3132847.3132916

[98] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. Nvidia A100 tensor core GPU: Performance and innovation. *IEEE Micro* 41, 2 (2021), 29–35. https://doi.org/10.1109/MM.2021.3061394

[99] Jack Choquette, Olivier Giroux, and Denis Foley. 2018. Volta: Performance and programmability. *IEEE Micro* 38, 2 (2018), 42–52. https://doi.org/10.1109/MM.2018.022071134

[100] Jack Choquette, Edward Lee, Ronny Krashinsky, Vishnu Balan, and Brucek Khailany. 2021. The A100 datacenter GPU and Ampere architecture. In *ISSCC*. IEEE, Washington, DC, USA, 48–50. https://doi.org/10.1109/ISSCC42613.2021.9365803

[101] Stavros Christodoulakis. 1984. Implications of certain assumptions in database performance evaluation. *TODS* 9, 2 (1984), 163–186. https://doi.org/10.1145/329.318578

[102] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating heterogeneous CPU–GPU parallelism in JIT compiled engines. *PVLDB* 12, 5 (2019), 544–556. https://doi.org/10.14778/3303753.3303760

[103] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. 2019. Hardware-conscious query processing in GPU-accelerated analytical engines. In *CIDR*. www.cidrdb.org, 1–9. http://cidrdb.org/cidr2019/papers/p127-chrysogelos-cidr19.pdf

[104] Hawon Chu, Seounghyun Kim, Joo-Young Lee, and Young-Kyoon Suh. 2020. Empirical evaluation across multiple GPU-accelerated DBMSes. In *DaMoN*. ACM, New York, NY, USA, 16:1–16:3. https://doi.org/10.1145/3399666.3399907

[105] Sungjun Chun, Wiren Dale Becker, Jon Casey, Steve Ostrander, Daniel Dreps, Jose Ale Hejase, Ryan Nett, Brian Beaman, and Jason R. Eagle. 2018. IBM POWER9 package technology and design. *IBM Journal of Research and Development* 62, 4/5 (2018), 12:1–12:10. https://doi.org/10.1147/JRD.2018.2847178

[106] Brett W. Coon and John Erik Lindholm. 2008. System and method for managing divergent threads in a SIMD architecture. Patent No. US7353369B1, Filed Jul. 13th., 2005, Issued Apr. 1st., 2008.

[107] Jonathan Corbet. 2015. Making kernel pages movable. *LWN.net* (July 2015). https://lwn.net/Articles/650917/

[108] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Çetintemel, and Stan Zdonik. 2015. An architecture for compiling UDF-centric workflows. *PVLDB* 8, 12 (2015), 1466–1477. https://doi.org/10.14778/2824032.2824045

[109] CXL 2019. *Compute Express Link specification, Revision 1.1*. CXL. https://www.computeexpresslink.org

[110] CXL 2020. *Compute Express Link specification, Revision 2.0*. CXL. https://www.computeexpresslink.org

[111] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley,

Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake elastic data warehouse. In *SIGMOD*. ACM, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[112] M. Dall'Osto, D. C. S. Beddows, P. Tunved, R. Krejci, J. Ström, H.-C. Hansson, Y. J. Yoon, Ki-Tae Park, S. Becagli, R. Udisti, T. Onasch, C. D. O'Dowd, R. Simó, and Roy M. Harrison. 2017. Arctic sea ice melt leads to atmospheric new particle formation. *Scientific reports* 7, 1, Article 3318 (2017). https://doi.org/10.1038/s41598-017-03328-1

[113] William James Dally and Brian Patrick Towles. 2004. Introduction to interconnection networks. In *Principles and practices of interconnection networks*. Morgan Kaufmann, San Francisco, CA, USA, Chapter 1, 1–23.

[114] William James Dally and Brian Patrick Towles. 2004. A simple interconnection network. In *Principles and practices of interconnection networks*. Morgan Kaufmann, San Francisco, CA, USA, Chapter 2, 25–73.

[115] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. 2019. From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms. *TODS* 44, 3 (2019), 9:1–9:46. https://doi.org/10.1145/3323991

[116] John Danskin. 2016. Assymetric coherent caching for heterogeneous computing. Patent No. US20160342513A1, Filed Oct. 14th., 2015, Issued Nov. 24th., 2016.

[117] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP*. ACM, New York, NY, USA, 33–48. https://doi.org/10.1145/2517349.2522714

[118] Jack W. Davidson and Sanjay Jinturkar. 1994. Memory access coalescing: A technique for eliminating redundant memory accesses. In *PLDI*. ACM, New York, NY, USA, 186–195. https://doi.org/10.1145/178243.178259

[119] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. 2010. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In *ISORC*. IEEE, Washington, DC, USA, 185–192. https://doi.org/10.1109/ISORC.2010.10

[120] Deloitte. 2017. *Hitting the accelerator: the next generation of machine-learning chips*. Retrieved Oct 1, 2019 from https://www2.deloitte.com/content/dam/Deloitte/

global/Images/infographics/technologymediatelecommunications/gx-deloitte-tmt-2018-nextgen-machine-learning-report.pdf

[121] James Leroy Deming, Mark Allen Mosley, and William Craig McKnight. 2017. Method for automatic page table compression. Patent No. US9569348B1, Filed Jul. 23rd., 2010, Issued Feb. 14th., 2011.

[122] James Leroy Deming, Mark Allen Mosley, William Craig McKnight, Emmett M. Kilgrariff, Steven E. Molnar, and Colyn Scott Case. 2011. Efficient memory translator with variable size cache line coverage. Patent No. US20110072235A1, Filed Oct. 8th., 2005, Issued Mar. 24th., 2011.

[123] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*. Association for Computational Linguistics, Stroudsburg, PA, USA, 4171–4186. https://doi.org/10.18653/v1/n19-1423

[124] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. 1984. Implementation techniques for main memory database systems. In *SIGMOD*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/602259.602261

[125] Gregory Frederick Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. 2011. SIMD re-convergence at thread frontiers. In *MICRO*. ACM, New York, NY, USA, 477–488. https://doi.org/10.1145/2155620.2155676

[126] Gregory Frederick Diamos, Richard Craig Johnson, Vinod Grover, Olivier Giroux, Jack H. Choquette, Michael Alan Fetterman, Ajay S. Tirumala, Peter Nelson, and Ronny Meir Krashinsky. 2016. Execution of divergent threads using a convergence barrier. Patent No. US2016019066A1, Filed Jul. 13th., 2015, Issued Jan. 21st., 2016.

[127] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. 1997. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms* 25, 1 (1997), 19–51. https://doi.org/10.1006/jagm.1997.0873

[128] Ulrich Drepper. 2007. *Memory part 5: What programmers can do*. LWN. Retrieved Jun 23, 2021 from https://lwn.net/Articles/255364/

[129] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, Garret Swart, and Weiwei Gong. 2019. A morsel-driven query execution engine for heterogeneous multi-cores. *PVLDB* 12, 12 (2019), 2218–2229.

[130] Cliff Edwards. 2022. *Nvidia opens NVLink for custom silicon integration*. Nvidia. Retrieved Mar 23, 2022 from https://nvidianews.nvidia.com/news/nvidia-opens-nvlink-for-custom-silicon-integration

[131] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. 2016. Compressed linear algebra for large-scale machine learning. *PVLDB* 9, 12 (2016), 960–971. https://doi.org/10.14778/2994509.2994515

[132] Charles Elkan. 2003. Using the triangle inequality to accelerate k-means. In *ICML*. AAAI Press, Menlo Park, CA, USA, 147–153. https://www.aaai.org/Papers/ICML/2003/ICML03-022.pdf

[133] Ahmed ElTantawy and Tor M. Aamodt. 2016. MIMD synchronization on SIMT architectures. In *MICRO*. IEEE, 11:1–11:14. https://doi.org/10.1109/MICRO.2016.7783714

[134] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *ISCA*. 365–376. https://doi.org/10.1145/2000064.2000108

[135] Jose M. Faleiro and Daniel J. Abadi. 2017. Latch-free synchronization in database systems: Silver bullet or fool's gold?. In *CIDR*. www.cidrdb.org, 1–12. http://cidrdb.org/cidr2017/papers/p121-faleiro-cidr17.pdf

[136] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database Compression on Graphics Processors. *PVLDB* 3, 1 (2010), 670–680. https://doi.org/10.14778/1920841.1920927

[137] Wenbin Fang, Ka Keung Lau, Mian Lu, Xiangye Xiao, Chi Kit Lam, Philip Yang Yang, Bingsheng He, Qiong Luo, Pedro V Sander, and Ke Yang. 2008. *Parallel data mining on graphics processors*. Technical Report HKUST-CS08-07. HKUST.

[138] Zhen Fang, Lixin Zhang, John B. Carter, Ali Ibrahim, and Michael A. Parker. 2007. Active memory operations. In *ICS*. ACM, New York, NY, USA, 232–241. https://doi.org/10.1145/1274971.1275004

[139] Reza Farivar, Daniel Rebolledo, Ellick Chan, and Roy H. Campbell. 2008. A parallel implementation of k-means clustering on GPUs. In *PDPTA*. CSREA Press, 340–345.

[140] FASTDATA$^{.io}$. 2019. *PlasmaENGINE*. Retrieved Jan 28, 2022 from https://fastdata.io/plasma-engine

[141] Denis Foley and John Danskin. 2017. Ultra-performance Pascal GPU and NVLink interconnect. *IEEE Micro* 37, 2 (2017), 7–17. https://doi.org/10.1109/MM.2017.37

[142] Philip Werner Frey and Gustavo Alonso. 2009. Minimizing the hidden cost of RDMA. In *ICDCS*. IEEE Computer Society, Los Alamitos, CA, USA, 553–560. https://doi.org/10.1109/ICDCS.2009.32

[143] Yupeng Fu and Chinmay Soman. 2021. Real-time data infrastructure at Uber. In *SIGMOD*. ACM, New York, NY, USA, 2503–2516. https://doi.org/10.1145/3448016.3457552

[144] Fujitsu 2021. *A64FX microarchitecture manual*. Fujitsu, Kawasaki, Japan. https://raw.githubusercontent.com/fujitsu/A64FX/master/doc/A64FX_Microarchitecture_Manual_en_1.6.pdf Revision 1.6.

[145] Wilson W. L. Fung, Ivan Sham, George L. Yuan, and Tor M. Aamodt. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *MICRO*. IEEE Computer Society, Los Alamitos, CA, USA, 407–420. https://doi.org/10.1109/MICRO.2007.30

[146] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined query processing in coprocessor environments. In *SIGMOD*. ACM, New York, NY, USA, 1603–1618. https://doi.org/10.1145/3183713.3183734

[147] Henning Funke and Jens Teubner. 2020. Data-parallel query processing on non-uniform data. *PVLDB* 13, 6 (2020), 884–897. https://doi.org/10.14778/3380750.3380758

[148] Ilya K. Ganusov, Mahesh A. Iyer, Ning Cheng, and Alon Meisler. 2020. Agilex generation of Intel FPGAs. In *HCS*. IEEE, Washington, DC, USA, 1–26. https://doi.org/10.1109/HCS49909.2020.9220557

[149] Hao Gao and Nikolay Sakharnykh. 2021. Scaling joins to a thousand GPUs. In *ADMS*. 55–64. http://www.adms-conf.org/2021-camera-ready/gao_adms21.pdf

[150] Brian Garabedian. 2020. *Xilinx announces world's highest bandwidth, highest compute density adaptable platform for network and cloud acceleration*. Xilinx. Retrieved Mar 1, 2022 from https://www.xilinx.com/news/media-kits/world-highest-bandwidth-highest-compute-density-adaptable-platform.html

[151] Victor Garcia-Flores, Eduard Ayguadé, and Antonio J. Peña. 2017. Efficient data sharing on heterogeneous systems. In *ICPP*. IEEE Computer Society, Los Alamitos, CA, USA, 121–130. https://doi.org/10.1109/ICPP.2017.21

[152] Gartner. 2019. *Gartner Says the Future of the Database Market Is the Cloud*. Retrieved Oct 1, 2019 from https://www.gartner.com/en/newsroom/press-releases/2019-07-01-gartner-says-the-future-of-the-database-market-is-the

[153] Vivek Gautam. 2021. *Shared virtual addressing for high performance Arm Infrastructure platforms*. Arm. Retrieved Apr 13, 2022 from https://static.linaro.org/connect/lvc21f/presentations/LVC21F-217.pdf

[154] Gen-Z Consortium 2021. *Exploring the future: CXL consortium & Gen-Z consortium sign letter of intent to advance interconnect technology*. Gen-Z Consortium. https://genzconsortium.org/exploring-the-future-cxl-consortium-gen-z-consortium-sign-letter-of-intent-to-advance-interconnect-technology

[155] Guin Gilman, Samuel S. Ogden, Tian Guo, and Robert J. Walls. 2020. Demystifying the placement policies of the Nvidia GPU thread block scheduler for concurrent kernels. *SIGMETRICS Perform. Evaluation Rev.* 48, 3 (2020), 81–88. https://doi.org/10.1145/3453953.3453972

[156] Google. 2022. *How Google Search organizes information*. Retrieved Feb 2, 2022 from https://www.google.com/intl/en/search/howsearchworks/how-search-works/organizing-information

[157] Mel Gorman. 2004. *Understanding the Linux virtual memory manager*. Prentice Hall PTR, Upper Saddle River, NJ, USA. https://www.kernel.org/doc/gorman/pdf/understand.pdf

[158] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming C. Lin, and Dinesh Manocha. 2004. Fast computation of database operations using graphics processors. In *SIGMOD*. ACM, New York, NY, USA, 215–226. https://doi.org/10.1145/1007568.1007594

[159] Michael Gowanlock, Ben Karsin, Zane Fink, and Jordan Wright. 2019. Accelerating the unacceleratable: Hybrid CPU/GPU algorithms for memory-bound database primitives. In *DaMoN*. ACM, New York, NY, USA, 7:1–7:11. https://doi.org/10.1145/3329785.3329926

[160] Aaron Grabein and Laura Graves. 2021. *AMD unveils workload-tailored innovations and products at the accelerated data center premiere*. AMD. Retrieved Feb 22, 2022 from https://www.amd.com/en/press-releases/2021-11-08-amd-unveils-workload-tailored-innovations-and-products-the-accelerated

[161] Chris Gregg and Kim M. Hazelwood. 2011. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *ISPASS*. IEEE Computer Society, Los Alamitos, CA, USA, 134–144. https://doi.org/10.1109/ISPASS.2011.5762730

[162] Phillip Grote. 2020. *Lock-based data structures on GPUs with independent thread scheduling*. Bachelor's thesis. TU Berlin. https://www.clemenslutz.com/pdfs/bsc_thesis_phillip_grote.pdf

[163] Tim Gubner, Diego G. Tomé, Harald Lang, and Peter A. Boncz. 2019. Fluid co-processing: GPU Bloom-filters for CPU joins. In *DaMoN*. ACM, New York, NY, USA, 9:1–9:10. https://doi.org/10.1145/3329785.3329934

[164] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the case for simpler data warehouses. In *SIGMOD*. ACM, New York, NY, USA, 1917–1923. https://doi.org/10.1145/2723372.2742795

[165] Prabhat K. Gupta. 2016. Accelerating datacenter workloads. Slides. https://www.fpl2016.org/slides/Gupta%20--%20Accelerating%20Datacenter%20Workloads.pdf Keynote speech at FPL.

[166] Jesse Hall and John Hart. 2004. GPU acceleration of iterative clustering. In *ACM Workshop on General-Purpose Computing on Graphics Processors*.

45–52. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.74.4859&rep=rep1&type=pdf

[167] Robert J. Halstead, Ildar Absalyamov, Walid A. Najjar, and Vassilis J. Tsotras. 2015. FPGA-based multithreading for in-memory hash joins. In *CIDR*. www.cidrdb.org, 1–9. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper12.pdf

[168] Robert J. Halstead, Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Sameh W. Asaad, and Balakrishna Iyer. 2013. Accelerating join operation for relational databases with FPGAs. In *FCCM*. IEEE Computer Society, Los Alamitos, CA, USA, 17–20. https://doi.org/10.1109/FCCM.2013.17

[169] Mark Harris, Shubhabrata Sengupta, and John D. Owens. 2007. Parallel prefix sum (scan) with CUDA. In *GPU gems 3*, Hubert Nguyen (Ed.). Addison-Wesley, Chapter 39. https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

[170] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational query coprocessing on graphics processors. *TODS* 34, 4, Article 21 (2009), 39 pages. https://doi.org/10.1145/1620585.1620588

[171] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2008. Relational joins on graphics processors. In *SIGMOD*. ACM, New York, NY, USA, 511–524. https://doi.org/10.1145/1376616.1376670

[172] Bingsheng He and Jeffrey Xu Yu. 2011. High-throughput transaction executions on graphics processors. *PVLDB* 4, 5 (2011), 314–325. https://doi.org/10.14778/1952376.1952381

[173] Jiong He, Mian Lu, and Bingsheng He. 2013. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. *PVLDB* 6, 10 (2013), 889–900. https://doi.org/10.14778/2536206.2536216

[174] Jiong He, Shuhao Zhang, and Bingsheng He. 2014. In-cache query co-processing on coupled CPU-GPU architectures. *PVLDB* 8, 4 (2014), 329–340. https://doi.org/10.14778/2735496.2735497

[175] Max Heimel and Volker Markl. 2012. A first step towards GPU-assisted query optimization. In *ADMS*. 33–44. http://www.adms-conf.org/heimel_adms12.pdf

[176] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB* 6, 9 (2013), 709–720. https://doi.org/10.14778/2536360.2536370

[177] Nathaniel D. Heintzman, Rhona K. Stuart, Gary Hon, Yutao Fu, Christina W. Ching, R. David Hawkins, Leah O. Barrera, Sara Van Calcar, Chunxu Qu, Keith A. Ching, Wei Wang, Zhiping Weng, Roland D. Green, Gregory E. Crawford, and Bing Ren. 2007. Distinct and predictive chromatin signatures of transcriptional promoters and enhancers in the human genome. *Nature Genetics* 39, 3 (2007), 311–318. https://doi.org/10.1038/ng1966

[178] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib analytics library or MAD skills, the SQL. *PVLDB* 5, 12 (2012), 1700–1711. https://doi.org/10.14778/2367502.2367510

[179] John L. Hennessy and David A. Patterson. 2017. Pipelining: Basic and intermediate concepts. In *Computer architecture — A quantitative approach* (6th ed.). Morgan Kaufmann, Cambridge, MA, USA, Appendix C, C1–C71.

[180] John L. Hennessy and David A. Patterson. 2017. Review of memory hierarchy. In *Computer architecture — A quantitative approach* (6th ed.). Morgan Kaufmann, Cambridge, MA, USA, Appendix B, B1–B60.

[181] John L. Hennessy and David A. Patterson. 2017. Thread-level parallelism. In *Computer architecture — A quantitative approach* (6th ed.). Morgan Kaufmann, Cambridge, MA, USA, Chapter 5, 367–462.

[182] John L. Hennessy and David A. Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (2019), 48–60. https://doi.org/10.1145/3282307

[183] Joel Hestness, Stephen W. Keckler, and David A. Wood. 2014. A comparative analysis of microarchitecture effects on CPU and GPU memory system behavior. In *IISWC*. IEEE Computer Society, Los Alamitos, CA, USA, 150–160. https://doi.org/10.1109/IISWC.2014.6983054

[184] HeteroDB. 2021. *PG-Strom*. Retrieved Jan 28, 2022 from https://en.heterodb.com

[185] Tony Hey, Stewart Tansley, and Kristin M. Tolle (Eds.). 2009. *The fourth paradigm: Data-intensive scientific discovery* (2nd ed.). Microsoft Research, Redmond, WA, USA. http://research.microsoft.com/en-us/collaboration/fourthparadigm

[186] Mark Horowitz. 2014. Computing's energy problem (and what we can do about it). In *ISSCC*. IEEE, Washington, DC, USA, 10–14. https://doi.org/10.1109/ISSCC.2014.6757323

[187] Jen-Cheng Huang, Joo Hwan Lee, Hyesoon Kim, and Hsien-Hsin S. Lee. 2014. GPUMech: GPU performance modeling technique based on interval analysis. In *MICRO*. IEEE Computer Society, Los Alamitos, CA, USA, 268–279. https://doi.org/10.1109/MICRO.2014.59

[188] IBM 2017. *POWER ISA version 3.0B*. IBM.

[189] IBM 2018. *POWER9 performance monitor unit user's guide*. IBM. Version 1.2.

[190] IBM 2018. *POWER9 processor user's manual*. IBM. Version 2.0.

[191] IBM POWER9 NPU team. 2018. Functionality and performance of NVLink with IBM POWER9 processors. *IBM Journal of Research and Development* 62, 4/5 (2018), 9:1–9:10. https://doi.org/10.1147/JRD.2018.2846978

[192] Kees Schouhamer Immink. 1990. Runlength-limited sequences. *Proc. IEEE* 78, 11 (1990), 1745–1759. https://doi.org/10.1109/5.63306

[193] Intel 2018. *Intel 64 and IA-32 architectures software developer's manual*. Intel.

[194] Intel 2019. *AN 835: PAM4 signaling fundamentals*. Intel. https://www.intel.com/content/www/us/en/docs/programmable/683852/current/introduction.html ID 683852.

[195] Intel. 2019. *Intel Agilex I-series 027 FPGA*. Retrieved Mar 1, 2022 from https://www.intel.com/content/www/us/en/products/sku/210676/intel-agilex-iseries-027-fpga-r29a/specifications.html

[196] Intel. 2019. *Intel Stratix 10 DX FPGA Product Brief*. Retrieved Oct 2, 2019 from https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/solution-sheets/stratix-10-dx-product-brief.pdf

[197] Intel. 2019. *Intel unveils new GPU architecture with high-performance computing and AI acceleration, and oneAPI software stack with unified and scalable*

abstraction for heterogeneous architectures. Intel. Retrieved Jun 10, 2021 from https://newsroom.intel.com/news-releases/intel-unveils-new-gpu-architecture-optimized-for-hpc-ai-oneapi

[198] Intel 2021. *3rd gen Intel Xeon Scalable processors product brief*. Intel. https://www.intel.com/content/dam/www/public/us/en/documents/a1171486-icelake-productbrief-updates-r1v2.pdf

[199] Intel 2021. *Intel Iris X^e MAX graphics open source programmer's reference manual*. Intel. https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-dg1-vol09-renderengine.pdf Revision 1.0.

[200] Intel 2021. *Intel Xeon Gold 6338 processor*. Intel. https://ark.intel.com/content/www/us/en/ark/products/212285/intel-xeon-gold-6338-processor-48m-cache-2-00-ghz.html

[201] Intel 2021. *oneAPI specification*. Intel. https://spec.oneapi.io/versions/1.1-rev-1/oneAPI-spec.pdf Release 1.1-rev-1.

[202] Intel 2022. *Intel virtualization technology for directed I/O*. Intel. http://software.intel.com/sites/default/files/managed/c5/15/vt-directed-io-spec.pdf Revision 3.4.

[203] Zsolt István, Kaan Kara, and David Sidler. 2020. FPGA-accelerated analytics: From single nodes to clusters. *Found. Trends Databases* 9, 2 (2020), 101–208. https://doi.org/10.1561/1900000072

[204] Akshay Jain, Mahmoud Khairy, and Timothy G. Rogers. 2018. A quantitative evaluation of contemporary GPU simulation methodology. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 2 (2018), 35:1–35:28. https://doi.org/10.1145/3224430

[205] Aamer Jaleel, Eiman Ebrahimi, and Sam Duncan. 2019. DUCATI: high-performance address translation by extending TLB reach of GPU-accelerated systems. *ACM Trans. Archit. Code Optim.* 16, 1 (2019), 6:1–6:24. https://doi.org/10.1145/3309710

[206] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. 2015. Improving main memory hash joins on Intel Xeon Phi processors: An experimental approach. *PVLDB* 8, 6 (2015), 642–653. https://doi.org/10.14778/2735703.2735704

[207] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the Nvidia Turing T4 GPU via microbenchmarking. arXiv:1903.07486 [cs.DC]

[208] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the Nvidia Volta GPU architecture via microbenchmarking. arXiv:1804.06826 [cs.DC]

[209] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-scale similarity search with GPUs. *IEEE Trans. Big Data* 7, 3 (2021), 535–547. https://doi.org/10.1109/TBDATA.2019.2921572

[210] D. Britton Johnston and Andrew Caldwell. 2019. *Amazon Redshift reimagined: RA3 and AQUA*. Amazon Web Services. Retrieved Jan 28, 2022 from https://d1.awsstatic.com/events/reinvent/2019/NEW_LAUNCH_Amazon_Redshift_reimagined_RA3_and_AQUA_ANT230.pdf

[211] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter C. Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David A. Patterson. 2021. Ten lessons from three generations shaped Google's TPUv4i. In *ISCA*. IEEE, 1–14. https://doi.org/10.1109/ISCA52012.2021.00010

[212] Krzysztof Kaczmarski. 2012. B $^{+}$ -Tree optimized for GPGPU. In *OTM (LNCS, Vol. 7566)*. Springer Berlin Heidelberg, Heidelberg, Germany, 843–854. https://doi.org/10.1007/978-3-642-33615-7_27

[213] Tim Kaldewey, Guy M. Lohman, René Müller, and Peter Benjamin Volk. 2012. GPU join processing revisited. In *DaMoN*. ACM, New York, NY, USA, 55–62. https://doi.org/10.1145/2236584.2236592

[214] Kaan Kara, Ken Eguro, Ce Zhang, and Gustavo Alonso. 2018. ColumnML: Column-store machine learning with on-the-fly data transformation. *PVLDB* 12, 4 (2018), 348–361. https://doi.org/10.14778/3297753.3297756

[215] Kaan Kara, Jana Giceva, and Gustavo Alonso. 2017. FPGA-based data partitioning. In *SIGMOD*. ACM, New York, NY, USA, 433–445. https://doi.org/10.1145/3035918.3035946

[216] Kaan Kara, Christoph Hagleitner, Dionysios Diamantopoulos, Dimitris Syrivelis, and Gustavo Alonso. 2020. High bandwidth memory on FPGAs: A data analytics perspective. arXiv:2004.01635 [cs.DC]

[217] Tomas Karnagel, Tal Ben-Nun, Matthias Werner, Dirk Habich, and Wolfgang Lehner. 2017. Big data causing big (TLB) problems: Taming random memory accesses on the GPU. In *DaMoN*. ACM, New York, NY, USA, 6:1–6:10. https://doi.org/10.1145/3076113.3076115

[218] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. 2017. Adaptive work placement for query processing on heterogeneous computing resources. *PVLDB* 10, 7 (2017), 733–744. https://doi.org/10.14778/3067421.3067423

[219] Tomas Karnagel, René Müller, and Guy M. Lohman. 2015. Optimizing GPU-accelerated group-by and aggregation. In *ADMS*. 13–24. http://www.adms-conf.org/2015/gpu-optimizer-camera-ready.pdf

[220] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. IEEE Computer Society, Los Alamitos, CA, USA, 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[221] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *ISCA*. IEEE, Washington, DC, USA, 473–486. https://doi.org/10.1109/ISCA45697.2020.00047

[222] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. AC/DC: In-database learning thunderstruck. In *DEEM*. ACM, New York, NY, USA, 8:1–8:10. https://doi.org/10.1145/3209889.3209896

[223] Farzad Khorasani, Mehmet E. Belviranli, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Stadium hashing: Scalable and flexible hashing on GPUs. In *PACT*. IEEE Computer Society, Los Alamitos, CA, USA, 63–74. https://doi.org/10.1109/PACT.2015.13

[224] Khronos 2012. *The OpenCL specification*. Khronos. https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf Version 1.2, Revision 19.

[225] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*. ACM, New York, NY, USA, 339–350. https://doi.org/10.1145/1807167.1807206

[226] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. 2009. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *PVLDB* 2, 2 (2009), 1378–1389. https://doi.org/10.14778/1687553.1687564

[227] Kinetica. 2022. *Kinetica*. Retrieved Jan 28, 2022 from https://www.kinetica.com

[228] Kristin M. Kleisner, Michael J. Fogarty, Sally McGee, Analie Barnett, Paula Fratantoni, Jennifer Greene, Jonathan A. Hare, Sean M. Lucey, Christopher McGuire, Jay Odell, Vincent S. Saba, Laurel Smith, Katherine J. Weaver, and Malin L. Pinsky. 2016. The effects of sub-regional climate velocity on the distribution and spatial extent of marine species assemblages. *PLOS ONE* 11, 2 (Feb. 2016), 1–21. https://doi.org/10.1371/journal.pone.0149220

[229] Donald E. Knuth. 1981. *Seminumerical algorithms* (2nd ed.). The art of computer programming, Vol. 2. Addison-Wesley, Reading, MA, USA.

[230] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter R. Pietzuch. 2019. Crossbow: Scaling deep learning with small batch sizes on multi-GPU servers. *PVLDB* 12, 11 (2019), 1399–1413. https://doi.org/10.14778/3342263.3342276

[231] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter R. Pietzuch. 2016. SABER: Window-based hybrid stream processing for heterogeneous architectures. In *SIGMOD*. ACM, New York, NY, USA, 555–569. https://doi.org/10.1145/2882903.2882906

[232] Martin Koppehel, Tobias Groth, Sven Groppe, and Thilo Pionteck. 2021. CuART — A CUDA-based, scalable radix-tree lookup and update engine. In *ICPP*. ACM, New York, NY, USA, 12:1–12:10. https://doi.org/10.1145/3472456.3472511

[233] Martin Krulis and Miroslav Kratochvíl. 2020. Detailed analysis and optimization of CUDA k-means algorithm. In *ICPP*. ACM, New York, NY, USA, 69:1–69:11. https://doi.org/10.1145/3404397.3404426

[234] Alexander Kumaigorodski. 2020. *Fast CSV loading using GPUs and RDMA for in-memory data processing*. Master's thesis. TU Berlin. https://www.clemenslutz.com/pdfs/msc_thesis_alexander_kumaigorodski.pdf

[235] Alexander Kumaigorodski, Clemens Lutz, and Volker Markl. 2021. Fast CSV loading using GPUs and RDMA for in-memory data processing. In *BTW (LNI,*

Vol. P-311). Gesellschaft für Informatik, Bonn, Germany, 19–38. https://doi.org/10.18420/btw2021-01

[236] Arun Kumar, Matthias Boehm, and Jun Yang. 2017. Data management in machine learning: Challenges, techniques, and systems. In *SIGMOD*. ACM, New York, NY, USA, 1717–1722. https://doi.org/10.1145/3035918.3054775

[237] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. An intermediate representation for optimizing machine learning pipelines. *PVLDB* 12, 11 (2019), 1553–1567. https://doi.org/10.14778/3342263.3342633

[238] Aki Kuusela and Clint Smullen. 2021. Video coding unit (VCU). In *HCS*. IEEE, Washington, DC, USA, 1–30. https://doi.org/10.1109/HCS52781.2021.9567040

[239] Zhuohang Lai, Qiong Luo, and Xiaoying Jia. 2018. Revisiting multi-pass scatter and gather on GPUs. In *ICPP*. ACM, New York, NY, USA, 25:1–25:11. https://doi.org/10.1145/3225058.3225095

[240] Robert Lasch, Mehdi Moghaddamfar, Norman May, Süleyman Sirri Demirsoy, Christian Färber, and Kai-Uwe Sattler. 2022. Bandwidth-optimal relational joins on FPGAs. In *EDBT*. OpenProceedings.org, Konstanz, Germany, 1–13. https://doi.org/10.5441/002/edbt.2022.03

[241] Ahmad Lashgar, Ebad Salehi, and Amirali Baniasadi. 2015. A case study in reverse engineering GPGPUs: Outstanding memory handling resources. *SIGARCH Comput. Archit. News* 43, 4 (2015), 15–21. https://doi.org/10.1145/2927964.2927968

[242] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. 2014. Industry 4.0. *Bus. Inf. Syst. Eng.* 6, 4 (2014), 239–242. https://doi.org/10.1007/s12599-014-0334-4

[243] Jae-Gil Lee, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, Oliver Draese, Frederick Ho, Stratos Idreos, Min-Soo Kim, Sam Lightstone, Guy M. Lohman, Konstantinos Morfonios, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Vincent Kulandai Samy, Richard Sidle, Knut Stolze, and Liping Zhang. 2014. Joins on encoded and partitioned data. *PVLDB* 7, 13 (2014), 1355–1366. https://doi.org/10.14778/2733004.2733008

[244] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The art of balance: A RateupDB experience of building a CPU/GPU hybrid database product. *PVLDB* 14, 12 (2021), 2999–3013. https://doi.org/10.14778/3476311.3476378

[245] Sunwoo Lee, Wei-keng Liao, Ankit Agrawal, Nikos Hardavellas, and Alok N. Choudhary. 2016. Evaluation of k-means data clustering algorithm on Intel Xeon Phi. In *Big Data*. IEEE, Washington, DC, USA, 2251–2260. https://doi.org/10.1109/BigData.2016.7840856

[246] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. 2010. Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In *ISCA*. ACM, New York, NY, USA, 451–460. https://doi.org/10.1145/1815961.1816021

[247] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*. ACM, New York, NY, USA, 743–754. https://doi.org/10.1145/2588555.2610507

[248] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2020. Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Trans. Parallel Distributed Syst.* 31, 1 (2020), 94–110. https://doi.org/10.1109/TPDS.2019.2928289

[249] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2018. Tartan: Evaluating Modern GPU Interconnect via a Multi-GPU Benchmark Suite. In *IISWC*. IEEE, Washington, DC, USA, 191–202. https://doi.org/10.1109/IISWC.2018.8573483

[250] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: Balancing I/O and GPU bandwidth in big data analytics. *PVLDB* 9, 14 (2016), 1647–1658. https://doi.org/10.14778/3007328.3007331

[251] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. 2013. NUMA-aware algorithms: The case of data shuffling. In

CIDR. www.cidrdb.org, 1–10. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper121.pdf

[252] You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu. 2010. Speeding up k-means algorithm by GPUs. In *CIT*. IEEE Computer Society, Los Alamitos, CA, USA, 115–122. https://doi.org/10.1109/CIT.2010.60

[253] Yuan Lin and Vinod Grover. 2018. *Using CUDA warp-level primitives*. Nvidia. Retrieved Jan 12, 2022 from https://developer.nvidia.com/blog/using-cuda-warp-level-primitives

[254] Erik Lindholm, John Nickolls, Stuart F. Oberman, and John Montrym. 2008. Nvidia Tesla: A unified graphics and computing architecture. *IEEE Micro* 28, 2 (2008), 39–55. https://doi.org/10.1109/MM.2008.31

[255] Nusrat Jahan Lisa, Annett Ungethüm, Dirk Habich, Wolfgang Lehner, Tuan D. A. Nguyen, and Akash Kumar. 2018. Column scan acceleration in hybrid CPU-FPGA systems. In *ADMS*. 22–33. http://www.adms-conf.org/2018-camera-ready/habich_adms2018.pdf

[256] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE Trans. Inf. Theory* 28, 2 (1982), 129–136. https://doi.org/10.1109/TIT.1982.1056489

[257] Andrea Lottarini, João Pedro Cerqueira, Thomas J. Repetti, Stephen A. Edwards, Kenneth A. Ross, Mingoo Seok, and Martha A. Kim. 2019. Master of none acceleration: A comparison of accelerator architectures for analytical query processing. In *ISCA*. ACM, New York, NY, USA, 762–773. https://doi.org/10.1145/3307650.3322220

[258] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A formal analysis of the NVIDIA PTX memory consistency model. In *ASPLOS*. ACM, New York, NY, USA, 257–270. https://doi.org/10.1145/3297858.3304043

[259] Clemens Lutz, Sebastian Breß, Tilmann Rabl, Steffen Zeuch, and Volker Markl. 2018. Efficient and scalable k-means on GPUs. *Datenbank-Spektrum* 18, 3 (2018), 157–169. https://doi.org/10.1007/s13222-018-0293-x

[260] Clemens Lutz, Sebastian Breß, Tilmann Rabl, Steffen Zeuch, and Volker Markl. 2018. Efficient k-means on GPUs. In *DaMoN*. ACM, New York, NY, USA, 3:1–3:3. https://doi.org/10.1145/3211922.3211925

[261] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump up the volume: Processing large data on GPUs with fast interconnects. In *SIGMOD*. ACM, New York, NY, USA, 1633–1649. https://doi.org/10.1145/3318464.3389705

[262] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton join: Efficiently scaling to a large join state on GPUs with fast interconnects. In *SIGMOD*. ACM, New York, NY, USA, 1017–1032. https://doi.org/10.1145/3514221.3517911

[263] James MacQueen. 1967. Some methods for classification and analysis of multivariate observations. In *Proc. Fifth Berkeley Symp. on Math. Statist. and Prob.*, Vol. 1. 281–297. https://digitalassets.lib.berkeley.edu/math/ucb/text/math_s5_v1_article-17.pdf

[264] Divya Mahajan, Joon Kyung Kim, Jacob Sacks, Adel Ardalan, Arun Kumar, and Hadi Esmaeilzadeh. 2018. In-RDBMS hardware acceleration of advanced analytics. *PVLDB* 11, 11 (2018), 1317–1331. https://doi.org/10.14778/3236187.3236188

[265] Tobias Maltenberger, Ivan Ilic, Ilin Tolovski, and Tilmann Rabl. 2022. Evaluating multi-GPU sorting with modern interconnects. In *SIGMOD*. ACM, New York, NY, USA, 1795–1809. https://doi.org/10.1145/3514221.3517842

[266] Data Management and Interchange Committee. 2003. *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*. Standard ISO/IEC 9075-2:2003 (E). International Organization for Standardization, Geneva, Switzerland. 799–800 pages. https://www.iso.org/standard/34133.html

[267] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2002. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.* 14, 4 (2002), 709–730. https://doi.org/10.1109/TKDE.2002.1019210

[268] Stefan Manegold, Peter A. Boncz, and Niels Nes. 2004. Cache-conscious radix-decluster projections. In *PVLDB*. Morgan Kaufmann, St. Louis, MO, USA, 684–695. https://doi.org/10.1016/B978-012088469-8.50061-9

[269] Market Data Forecast. 2021. *GPU database market*. Retrieved Jan 28, 2022 from https://www.marketdataforecast.com/market-reports/gpu-database-market

[270] Volker Markl, Peter J. Haas, Marcel Kutsch, Nimrod Megiddo, Utkarsh Srivastava, and Tam Minh Tran. 2007. Consistent selectivity estimation via maximum entropy. *VLDB J.* 16, 1 (2007), 55–76. https://doi.org/10.1007/s00778-006-0030-1

[271] Maximize Market Research. 2020. *Global GPU database market*. Retrieved Jan 28, 2022 from https://www.maximizemarketresearch.com/market-report/global-gpu-database-market/22455

[272] David Mayhew and Venkata Krishnan. 2003. PCI express and advanced switching: Evolutionary path to building next generation interconnects. In *HOTIC*. IEEE Computer Society, Los Alamitos, CA, USA, 21–29. https://doi.org/10.1109/CONECT.2003.1231473

[273] Xinxin Mei and Xiaowen Chu. 2017. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Trans. Parallel Distributed Syst.* 28, 1 (2017), 72–86. https://doi.org/10.1109/TPDS.2016.2549523

[274] Sina Meraji, Berni Schiefer, Lan Pham, Lee Chu, Peter Kokosielis, Adam Storm, Wayne Young, Chang Ge, Geoffrey Ng, and Kajan Kanagaratnam. 2016. Towards a hybrid design for fast query processing in DB2 with BLU acceleration using graphical processing units: A technology demonstration. In *SIGMOD*. ACM, New York, NY, USA, 1951–1960. https://doi.org/10.1145/2882903.2903735

[275] Duane Merrill and Michael Garland. 2016. *Single-pass parallel prefix scan with decoupled look-back*. Technical Report NVR-2016-002. Nvidia. https://research.nvidia.com/sites/default/files/pubs/2016-03_Single-pass-Parallel-Prefix/nvr-2016-002.pdf

[276] Disa Mhembere, Da Zheng, Carey E. Priebe, Joshua T. Vogelstein, and Randal Burns. 2017. knor: A NUMA-optimized in-memory, distributed and semi-external-memory k-means library. In *HPDC*. ACM, New York, NY, USA, 67–78. https://doi.org/10.1145/3078597.3078607

[277] Adrian Michalke, Philipp M. Grulich, Clemens Lutz, Steffen Zeuch, and Volker Markl. 2021. An energy-efficient stream join for the Internet of Things. In *DaMoN*. ACM, New York, NY, USA, 8:1–8:6. https://doi.org/10.1145/3465998.3466005

[278] Frank Mietke, Robert Rex, Robert Baumgartl, Torsten Mehlan, Torsten Hoefler, and Wolfgang Rehm. 2006. Analysis of the memory registration process in the Mellanox

InfiniBand software stack. In *Euro-Par (LNCS, Vol. 4128)*. Springer Berlin Heidelberg, Heidelberg, Germany, 124–133. https://doi.org/10.1007/11823285_13

[279] Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramírez, and David W. Nellans. 2017. Beyond the socket: NUMA-aware GPUs. In *MICRO*. IEEE/ACM, New York, NY, USA, 123–135. https://doi.org/10.1145/3123939.3124534

[280] David J. Miller, Philip M. Watts, and Andrew W. Moore. 2009. Motivating future interconnects: A differential measurement analysis of PCI latency. In *ANCS*. ACM, New York, NY, USA, 94–103. https://doi.org/10.1145/1882486.1882513

[281] Seungwon Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-Mei Hwu. 2020. EMOGI: Efficient memory-access for out-of-memory graph-traversal in GPUs. *PVLDB* 14, 2 (2020), 114–127. https://doi.org/10.14778/3425879.3425883

[282] Andreas Moshovos, Gokhan Memik, Babak Falsafi, and Alok N. Choudhary. 2001. JETTY: Filtering snoops for reduced energy consumption in SMP servers. In *HPCA*. IEEE Computer Society, Los Alamitos, CA, USA, 85–96. https://doi.org/10.1109/HPCA.2001.903254

[283] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-efficient aggregation: Hashing is sorting. In *SIGMOD*. ACM, New York, NY, USA, 1123–1136. https://doi.org/10.1145/2723372.2747644

[284] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. *A primer on memory consistency and cache coherence* (2nd ed.). Morgan & Claypool Publishers, San Rafael, CA, USA. https://doi.org/10.2200/S00962ED2V01Y201910CAC049

[285] Milan Navrátil, Laura Bailey, and Charlie Boyle. 2018. *Red Hat Enterprise Linux 7 performance tuning guide*. Red Hat. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/performance_tuning_guide Revision 10.13-59.

[286] Dan Negrut, Radu Serban, Ang Li, and Andrew Seidl. 2014. *Unified memory in CUDA 6.0: A brief overview of related data access and transfer issues*. University of Wisconsin-Madison. https://sbel.wisc.edu/wp-content/uploads/sites/569/2018/05/TR-2014-09.pdf TR-2014-09.

[287] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe performance for end host networking. In *SIGCOMM*. ACM, New York, NY, USA, 327–341. https://doi.org/10.1145/3230543.3230560

[288] Anh Nguyen, Masato Edahiro, and Shinpei Kato. 2018. GPU-accelerated VoltDB: A case for indexed nested loop join. In *HPCS*. IEEE, Washington, DC, USA, 204–212. https://doi.org/10.1109/HPCS.2018.00046

[289] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet D. Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchý. 2019. Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: A survey. *Artif. Intell. Rev.* 52, 1 (2019), 77–124. https://doi.org/10.1007/s10462-018-09679-z

[290] Ritesh Nohria, Gustavo Santos, and Volker Haug. 2019. *IBM power system AC922 introduction and technical overview* (1st ed.). IBM International Technical Support Organization, Poughkeepsie, NY, USA. https://www.redbooks.ibm.com/redpapers/pdfs/redp5494.pdf REDP-5494-00.

[291] Bryon S Nordquist and Stephen D Lew. 2009. Apparatus, system, and method for coalescing parallel memory requests. Patent No. US7492368B1, Filed Jan. 24th., 2006, Issued Feb. 17th., 2009.

[292] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Bart Mesman. 2011. High performance predictable histogramming on GPUs: Exploring and evaluating algorithm trade-offs. In *GPGPU*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/1964179.1964181

[293] Nvidia 2009. *Nvidia's next generation CUDA compute architecture: Fermi*. Nvidia. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf Version 1.1.

[294] Nvidia 2013. *Nvidia's next generation CUDA compute architecture: Kepler GK110*. Nvidia. https://compas.cs.stonybrook.edu/~nhonarmand/courses/sp16/cse502/res/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf Version 1.0.

[295] Nvidia 2014. *Nvidia's next generation CUDA compute architecture: Kepler GK110/210*. Nvidia. https://www.nvidia.com/content/dam/en-

zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf Version 1.1.

[296] Nvidia 2016. *Nvidia Tesla P100*. Nvidia. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf WP-08019-001_v01.1.

[297] Nvidia 2017. *Nvidia Tesla V100 GPU architecture*. Nvidia. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf WP-08608-001_v1.1.

[298] Nvidia. 2017. *Tuning CUDA applications for Maxwell*. Technical Report DA-07173-001_v9.0. http://docs.nvidia.com/cuda/pdf/Maxwell_Tuning_Guide.pdf

[299] Nvidia 2019. *Pascal MMU format changes*. Nvidia. https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf Git Revision 60b67c3.

[300] Nvidia 2019. *Tuning CUDA applications for Pascal*. Nvidia. https://docs.nvidia.com/cuda/pdf/Pascal_Tuning_Guide.pdf DA-08134-001_v10.1.

[301] Nvidia 2020. *CUPTI user's guide*. Nvidia. https://docs.nvidia.com/cupti/pdf/Cupti.pdf DA-05679-001_v11.1.

[302] Nvidia 2020. *Nvidia A100 tensore core GPU architecture*. Nvidia. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf Version 1.0.

[303] Nvidia 2021. *CUDA C++ best practices guide*. Nvidia. https://docs.nvidia.com/cuda/archive/11.3.1/pdf/CUDA_C_Best_Practices_Guide.pdf DG-05603-001_v11.3.

[304] Nvidia 2021. *CUDA C++ programming guide*. Nvidia. https://docs.nvidia.com/cuda/archive/11.3.1/pdf/CUDA_C_Programming_Guide.pdf PG-02829-001_v11.3.

[305] Nvidia 2021. *Inline PTX assembly in CUDA*. Nvidia. https://docs.nvidia.com/cuda/archive/11.5.1/pdf/Inline_PTX_Assembly.pdf SP-04456-001_v11.5.

[306] Nvidia 2021. *Nvidia A100 tensore core GPU*. Nvidia. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf

[307] Nvidia 2021. *Parallel thread execution ISA*. Nvidia. https://docs.nvidia.com/cuda/archive/11.3.1/pdf/ptx_isa_7.3.pdf Version 7.3.

[308] Nvidia 2022. *Nvidia H100 tensore core GPU architecture*. Nvidia. https://nvdam.widen.net/s/9bz6dw7dqr/gtc22-whitepaper-hopper Version 1.0.

[309] Nvidia 2022. *Tuning CUDA applications for Ampere*. Nvidia. https://docs.nvidia.com/cuda/pdf/Ampere_Tuning_Guide.pdf DA-09073-001_v11.6.

[310] Lars Nyland and Stephen Jones. 2013. Understanding and using atomic memory operations. In *GTC*. Nvidia, 1–61. https://on-demand.gputechconf.com/gtc/2013/presentations/S3101-Atomic-Memory-Operations.pdf

[311] Ignacio Sanudo Olmedo, Nicola Capodieci, Jorge Luis Martinez, Andrea Marongiu, and Marko Bertogna. 2020. Dissecting the CUDA scheduling hierarchy: A performance and predictability perspective. In *RTAS*. IEEE, Washington, DC, USA, 213–225. https://doi.org/10.1109/RTAS48715.2020.000-5

[312] Lena E. Olson, Mark D. Hill, and David A. Wood. 2017. Crossing guard: Mediating host-accelerator coherence interactions. In *ASPLOS*. ACM, New York, NY, USA, 163–176. https://doi.org/10.1145/3037697.3037715

[313] OmniSci. 2021. *OmniSciDB*. Retrieved Jan 28, 2022 from https://www.omnisci.com/platform/omniscidb

[314] OpenCAPI Consortium 2020. *OpenCAPI 3.0 transaction layer specification*. OpenCAPI Consortium. Version 1.0.

[315] OpenCAPI Consortium 2020. *OpenCAPI 4.0 transaction layer specification*. OpenCAPI Consortium. Version 1.0.

[316] OpenCAPI Consortium 2020. *OpenCAPI data link layer (DL 3.0/3.1/4.0) architecture specification*. OpenCAPI Consortium. Version 2.0.

[317] Carlos Ordonez. 2004. Programming the k-means clustering algorithm in SQL. In *SIGKDD*. ACM, New York, NY, USA, 823–828. https://doi.org/10.1145/1014052.1016921

[318] Muhsen Owaida, Gustavo Alonso, Laura Fogliarini, Anthony Hock-Koon, and Pierre-Etienne Melet. 2019. Lowering the latency of data processing pipelines through FPGA based hardware acceleration. *PVLDB* 13, 1 (2019), 71–85. https://doi.org/10.14778/3357377.3357383

[319] Mark S. Papamarcos and Janak H. Patel. 1984. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA*. ACM, New York, NY, USA, 348–354. https://doi.org/10.1145/800015.808204

[320] Linnea Passing, Manuel Then, Nina C. Hubig, Harald Lang, Michael Schreier, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. 2017. SQL- and operator-centric data analytics in relational main-memory databases. In *EDBT*. OpenProceedings.org, Konstanz, Germany, 84–95. https://doi.org/10.5441/002/edbt.2017.09

[321] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2019. Revisiting hash join on graphics processors: A decade later. In *ICDEW*. IEEE, Washington, DC, USA, 294–299. https://doi.org/10.1109/ICDEW.2019.00008

[322] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Improving execution efficiency of just-in-time compilation based query processing on GPUs. *PVLDB* 14, 2 (2020), 202–214. https://doi.org/10.14778/3425879.3425890

[323] Johns Paul, Jiong He, and Bingsheng He. 2016. GPL: A GPU-based pipelined query processing engine. In *SIGMOD*. ACM, New York, NY, USA, 1935–1950. https://doi.org/10.1145/2882903.2915224

[324] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. MG-Join: A scalable join for massively parallel multi-GPU architectures. In *SIGMOD*. ACM, New York, NY, USA, 1413–1425. https://doi.org/10.1145/3448016.3457254

[325] PCI-SIG 2009. *Address translation services revision 1.1*. PCI-SIG.

[326] PCI-SIG 2017. *PCI express base specification revision 4.0*. PCI-SIG. Version 1.0.

[327] PCI-SIG 2019. *PCI express base specification revision 5.0*. PCI-SIG. Version 1.0.

[328] Carl Pearson, I-Hsin Chung, Zehra Sura, Wen-Mei Hwu, and Jinjun Xiong. 2018. NUMA-aware data-transfer measurements for Power/NVLink multi-GPU systems. In *IWOPH (LNCS, Vol. 11203)*. Springer, Cham, Switzerland, 448–454. https://doi.org/10.1007/978-3-030-02465-9_32

[329] Carl Pearson, Abdul Dakkak, Sarah Hashash, Cheng Li, I-Hsin Chung, Jinjun Xiong, and Wen-Mei Hwu. 2019. Evaluating characteristics of CUDA communication primitives on high-bandwidth interconnects. In *ICPE*. ACM, New York, NY, USA, 209–218. https://doi.org/10.1145/3297663.3310299

[330] Andrea Pellegrini. 2021. Arm Neoverse N2: Arm's 2$^{\text{nd}}$ generation high performance infrastructure CPUs and system IPs. In *HCS*. IEEE, Washington, DC, USA, 1–27. https://doi.org/10.1109/HCS52781.2021.9567483

[331] Andrea Pellegrini, Ashok Kumar Tummala, Jamshed Jalal, Mark Werkheiser, Anitha Kona, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, and Tushar Ringe. 2020. The Arm Neoverse N1 platform: Building blocks for the next-gen cloud-to-edge infrastructure SoC. *IEEE Micro* 40, 2 (2020), 53–62. https://doi.org/10.1109/MM.2020.2972222

[332] Johan Peltenburg, Ákos Hadnagy, Matthijs Brobbel, Robert Morrow, and Zaid Al-Ars. 2021. Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators. In *FPT*. IEEE, Washington, DC, USA, 1–9. https://doi.org/10.1109/ICFPT52863.2021.9609833

[333] Johan Peltenburg, Lars T. J. van Leeuwen, Joost Hoozemans, Jian Fang, Zaid Al-Ars, and H. Peter Hofstee. 2020. Battling the CPU bottleneck in Apache Parquet to Arrow conversion using FPGA. In *FPT*. IEEE, Washington, DC, USA, 281–286. https://doi.org/10.1109/ICFPT51103.2020.00048

[334] Larry Peterson and Bruce Davie. 2007. Direct Link Networks. In *Computer networks — A systems approach* (4th ed.). Morgan Kaufmann, San Francisco, CA, USA, Chapter 2, 64–165.

[335] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *HPCA*. IEEE Computer Society, Los Alamitos, CA, USA, 558–567. https://doi.org/10.1109/HPCA.2014.6835964

[336] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced large-reach TLBs. In *MICRO*. IEEE Computer Society, Los Alamitos, CA, USA, 258–269. https://doi.org/10.1109/MICRO.2012.32

[337] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural support for address translation on GPUs: Designing memory management units for CPU/GPUs with unified address spaces. In *ASPLOS*. ACM, New York, NY, USA, 743–758. https://doi.org/10.1145/2541940.2541942

[338] Holger Pirk, Sam Madden, and Mike Stonebraker. 2015. By their fruits shall ye know them: A data analyst's perspective on massively parallel system design. In *DaMoN*. ACM, New York, NY, USA, 5:1–5:6. https://doi.org/10.1145/2771937.2771944

[339] Holger Pirk, Stefan Manegold, and Martin L. Kersten. 2011. Accelerating foreign-key joins using asymmetric memory channels. In *ADMS*. 27–35. http://www.adms-conf.org/p27-PIRK.pdf

[340] Holger Pirk, Stefan Manegold, and Martin L. Kersten. 2014. Waste not... Efficient co-processing of relational data. In *ICDE*. IEEE Computer Society, Los Alamitos, CA, USA, 508–519. https://doi.org/10.1109/ICDE.2014.6816677

[341] Holger Pirk, Oscar R. Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - A vector algebra for portable database performance on modern hardware. *PVLDB* 9, 14 (2016), 1707–1718. https://doi.org/10.14778/3007328.3007336

[342] Apostolos Planas. 2022. *Cooperative heterogeneous query execution*. Master's thesis. TU Berlin. https://www.clemenslutz.com/pdfs/msc_thesis_apostolos_planas.pdf

[343] Constantin Pohl and Kai-Uwe Sattler. 2018. Joins in a heterogeneous memory hierarchy: Exploiting high-bandwidth memory. In *DaMoN*. ACM, New York, NY, USA, 8:1–8:10. https://doi.org/10.1145/3211922.3211929

[344] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*. ACM, New York, NY, USA, 1493–1508. https://doi.org/10.1145/2723372.2747645

[345] Orestis Polychroniou and Kenneth A. Ross. 2019. Towards practical vectorized analytical query engines. In *DaMoN*. ACM, New York, NY, USA, 10:1–10:7. https://doi.org/10.1145/3329785.3329928

[346] Orestis Polychroniou and Kenneth A Ross. 2020. VIP: A SIMD vectorized analytical query engine. *VLDB J.* 29, 6 (2020), 1243–1261. https://doi.org/10.1007/s00778-020-00621-w

[347] Jason Power, Mark D. Hill, and David A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *HPCA*. IEEE Computer Society, Los Alamitos, CA, USA, 568–578. https://doi.org/10.1109/HPCA.2014.6835965

[348] Rateup. 2021. *RateupDB*. Retrieved Jan 28, 2022 from http://www.rateup.com.cn/dist/#/product

[349] Aunn Raza, Periklis Chrysogelos, Panagiotis Sioulas, Vladimir Indjic, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. 2020. GPU-accelerated data management under the test of time. In *CIDR*. www.cidrdb.org, 1–11. http://cidrdb.org/cidr2020/papers/p18-raza-cidr20.pdf

[350] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. 2020. Survey of machine learning accelerators. In *HPEC*. IEEE, Washington, DC, USA, 1–12. https://doi.org/10.1109/HPEC43674.2020.9286149

[351] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W. Keckler. 2018. Compressing DMA engine: Leveraging activation sparsity for training deep neural networks. In *HPCA*. IEEE Computer Society, Los Alamitos, CA, USA, 78–91. https://doi.org/10.1109/HPCA.2018.00017

[352] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A seven-dimensional analysis of hashing methods and its implications on query processing. *PVLDB* 9, 3 (2015), 96–107. https://doi.org/10.14778/2850583.2850585

[353] Steve Roberts, Pradeep Ramanna, and John Walthour. 2018. AC922 data movement for CORAL. In *HPEC*. IEEE, Washington, DC, USA, 1–5. https://doi.org/10.1109/HPEC.2018.8547707

[354] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-conscious wavefront scheduling. In *MICRO*. IEEE Computer Society, Los Alamitos, CA, USA, 72–83. https://doi.org/10.1109/MICRO.2012.16

[355] Christopher Root and Todd Mostak. 2016. MapD: A GPU-powered big data analytics and visualization platform. In *SIGGRAPH*. ACM, New York, NY, USA, 73:1–73:2. https://doi.org/10.1145/2897839.2927468

[356] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2022. Query processing on heterogeneous CPU/GPU systems. *ACM Comput. Surv.* 55, 1, Article 11 (Jan. 2022), 38 pages. https://doi.org/10.1145/3485126

[357] Viktor Rosenfeld, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2019. Performance analysis and automatic tuning of hash aggregation on GPUs.

In *DaMoN*. ACM, New York, NY, USA, 8:1–8:11. https://doi.org/10.1145/3329785.3329922

[358] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: A compiler and runtime for heterogeneous systems. In *SOSP*. ACM, New York, NY, USA, 49–68. https://doi.org/10.1145/2517349.2522715

[359] Eyal Rozenberg and Peter A. Boncz. 2017. Faster across the PCIe bus: A GPU library for lightweight decompression: including support for patched compression schemes. In *DaMoN*. ACM, New York, NY, USA, 8:1–8:5. https://doi.org/10.1145/3076113.3076122

[360] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient join algorithms for large database tables in a multi-GPU environment. *PVLDB* 14, 4 (2020), 708–720. https://doi.org/10.14778/3436905.3436927

[361] Ran Rui and Yi-Cheng Tu. 2017. Fast equi-join algorithms on GPUs: Design and implementation. In *SSDBM*. ACM, New York, NY, USA, 17:1–17:12. https://doi.org/10.1145/3085504.3085521

[362] Conrad Sanderson and Ryan R. Curtin. 2016. Armadillo: A template-based C++ library for linear algebra. *J. Open Source Softw.* 1, 2 (2016), 26. https://doi.org/10.21105/joss.00026

[363] Nadathur Satish, Mark J. Harris, and Michael Garland. 2009. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS*. IEEE, Washington, DC, USA, 1–10. https://doi.org/10.1109/IPDPS.2009.5161005

[364] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. 2010. Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. In *SIGMOD*. ACM, New York, NY, USA, 351–362. https://doi.org/10.1145/1807167.1807207

[365] Tobias Schmidt, Maximilian Bandle, and Jana Giceva. 2021. A four-dimensional analysis of partitioned approximate filters. *PVLDB* 14, 11 (2021), 2355–2368. https://doi.org/10.14778/3476249.3476286

[366] Klaus-Dieter Schubert, Syed Saif Abrar, Duane Averill, Ellen Bauman, Aaron C. Brown, Ron Cash, Debapriya Chatterjee, John Gullickson, Mark Nelson, Kevin A.

Pasnik, and Krishnan Sugavanam. 2018. Addressing verification challenges of heterogeneous systems based on IBM POWER9. *IBM Journal of Research and Development* 62, 4/5 (2018), 11:1–11:12. https://doi.org/10.1147/JRD.2018.2848418

[367] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An experimental comparison of thirteen relational equi-joins in main memory. In *SIGMOD*. ACM, New York, NY, USA, 1961–1976. https://doi.org/10.1145/2882903.2882917

[368] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. 2015. On the surprising difficulty of simple things: The case of radix partitioning. *PVLDB* 8, 9 (2015), 934–937. https://doi.org/10.14778/2777598.2777602

[369] Maximilian E. Schüle, Jakob Huber, Alfons Kemper, and Thomas Neumann. 2020. Freedom for the SQL-lambda: Just-in-time-compiling user-injected functions in PostgreSQL. In *SSDBM*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3400903.3400915

[370] Maximilian E. Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Günnemann. 2021. In-database machine learning with SQL on GPUs. In *SSDBM*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/3468791.3468840

[371] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. 2008. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graph.* 27, 3 (2008), 18. https://doi.org/10.1145/1360612.1360617

[372] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2016. A hybrid B+-tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms. In *SIGMOD*. ACM, New York, NY, USA, 1523–1538. https://doi.org/10.1145/2882903.2882918

[373] S. A. Arul Shalom, Manoranjan Dash, and Minh Tue. 2008. Efficient k-means clustering using accelerated graphics processors. In *DaWaK (LNCS, Vol. 5182)*. Springer Berlin Heidelberg, Heidelberg, Germany, 166–175. https://doi.org/10.1007/978-3-540-85836-2_16

[374] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *SIGMOD*. ACM, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595

[375] Anil Shanbhag, Nesime Tatbul, David Cohen, and Samuel Madden. 2020. Large-scale in-memory analytics on Intel Optane DC persistent memory. In *DaMoN*. ACM, New York, NY, USA, 4:1–4:8. https://doi.org/10.1145/3399666.3399933

[376] Debendra Das Sharma. 2020. PCI Express 6.0 specification at 64.0 GT/s with PAM-4 signaling: A low latency, high bandwidth, high reliability and cost-effective interconnect. In *HOTI*. IEEE, Washington, DC, USA, 1–8. https://doi.org/10.1109/HOTI51249.2020.00016

[377] Debendra Das Sharma. 2021. A low latency approach to delivering alternate protocols with coherency and memory semantics using PCI Express 6.0 PHY at 64.0 GT/s. In *HOTI*. IEEE, Washington, DC, USA, 35–42. https://doi.org/10.1109/HOTI52880.2021.00019

[378] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. 1994. Cache conscious algorithms for relational query processing. In *PVLDB*. Morgan Kaufmann, San Francisco, CA, USA, 510–521. https://dl.acm.org/doi/10.5555/645920.758363

[379] David E. Shaw, Peter J. Adams, Asaph Azaria, Joseph A. Bank, Brannon Batson, Alistair Bell, Michael Bergdorf, Jhanvi Bhatt, J. Adam Butts, Timothy Correia, Robert M. Dirks, Ron O. Dror, Michael P. Eastwood, Bruce Edwards, Amos Even, Peter Feldmann, Michael Fenn, Christopher H. Fenton, Anthony Forte, Joseph Gagliardo, Gennette Gill, Maria Gorlatova, Brian Greskamp, J. P. Grossman, Justin Gullingsrud, Anissa Harper, William Hasenplaugh, Mark Heily, Benjamin Colin Heshmat, Jeremy Hunt, Douglas J. Ierardi, Lev Iserovich, Bryan L. Jackson, Nick P. Johnson, Mollie M. Kirk, John L. Klepeis, Jeffrey S. Kuskin, Kenneth M. Mackenzie, Roy J. Mader, Richard McGowen, Adam McLaughlin, Mark A. Moraes, Mohamed H. Nasr, Lawrence J. Nociolo, Lief O'Donnell, Andrew Parker, Jon L. Peticolas, Goran Pocina, Cristian Predescu, Terry Quan, John K. Salmon, Carl Schwink, Keun Sup Shim, Naseer Siddique, Jochen Spengler, Tamas Szalay, Raymond Tabladillo, Reinhard Tartler, Andrew G. Taube, Michael Theobald, Brian Towles, William Vick, Stanley C. Wang, Michael Wazlowski, Madeleine J. Weingarten, John M. Williams, and Kevin A. Yuh. 2021. Anton 3: Twenty

microseconds of molecular dynamics simulation before lunch. In *SC*. ACM, New York, NY, USA, 1:1–1:11. https://doi.org/10.1145/3458817.3487397

[380] Jian Shen, Ze Wang, David Wang, Jeremy Shi, and Steven Chen. 2019. *Introducing AresDB: Uber's GPU-powered open source, real-time analytics engine*. Retrieved Jan 28, 2022 from https://eng.uber.com/aresdb

[381] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel H. Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. 2018. Scheduling page table walks for irregular GPU applications. In *ISCA*. IEEE Computer Society, Los Alamitos, CA, USA, 180–192. https://doi.org/10.1109/ISCA.2018.00025

[382] Seunghee Shin, Michael LeBeane, Yan Solihin, and Arkaprava Basu. 2018. Neighborhood-aware address translation for irregular GPU applications. In *MICRO*. IEEE Computer Society, Los Alamitos, CA, USA, 352–363. https://doi.org/10.1109/MICRO.2018.00036

[383] Michael Shindler, Alex Wong, and Adam Meyerson. 2011. Fast and accurate k-means for large datasets. 24 (2011), 2375–2383. https://proceedings.neurips.cc/paper/2011/file/52c670999cdef4b09eb656850da777c4-Paper.pdf

[384] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. 2013. Cache coherence for GPU architectures. In *HPCA*. IEEE, New York, NY, USA, 578–590. https://doi.org/10.1109/HPCA.2013.6522351

[385] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-conscious hash-joins on GPUs. In *ICDE*. IEEE, Washington, DC, USA, 698–709. https://doi.org/10.1109/ICDE.2019.00068

[386] Evangelia A. Sitaridi and Kenneth A. Ross. 2013. Optimizing select conditions on GPUs. In *DaMoN*. ACM, New York, NY, USA, 4:1–4:8. https://doi.org/10.1145/2485278.2485282

[387] Greg Smith. 2014. *Stack Overflow*. Nvidia. Retrieved Nov 8, 2021 from https://stackoverflow.com/questions/27253434/what-does-overflow-mean-during-cuda-profiling/27262797#27262797

[388] Greg Smith. 2019. *Nvidia Developer Forums*. Nvidia. Retrieved Oct 13, 2020 from https://forums.developer.nvidia.com/t/pascal-l1-cache/49571/14

[389] Greg Smith. 2019. *Nvidia Developer Forums*. Nvidia. Retrieved Oct 13, 2020 from https://forums.developer.nvidia.com/t/performance-counters-similar-to-cpu/74949/2

[390] Tyler Sorensen and Alastair F. Donaldson. 2016. Exposing errors related to weak memory in GPU applications. In *PLDI*. ACM, New York, NY, USA, 100–113. https://doi.org/10.1145/2908080.2908114

[391] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. 2017. KeystoneML: Optimizing pipelines for large-scale advanced analytics. In *ICDE*. IEEE Computer Society, 535–546. https://doi.org/10.1109/ICDE.2017.109

[392] SQream. 2022. *SQream DB*. Retrieved Jan 28, 2022 from https://sqream.com/product/data-acceleration-platform/sql-gpu-database

[393] William J. Starke, J. S. Dodson, Jeffrey Stuecheli, Eric Retter, Brad W. Michael, Stephen J. Powell, and James A. Marcella. 2018. IBM POWER9 memory architectures for optimized systems. *IBM Journal of Research and Development* 62, 4/5 (2018), 3:1–3:13. https://doi.org/10.1147/JRD.2018.2846159

[394] William J. Starke, Brian W. Thompto, Jeffrey Stuecheli, and José E. Moreira. 2021. IBM's POWER10 processor. *IEEE Micro* 41, 2 (2021), 7–14. https://doi.org/10.1109/MM.2021.3058632

[395] Elias Stehle and Hans-Arno Jacobsen. 2017. A memory bandwidth-efficient hybrid radix sort on GPUs. In *SIGMOD*. ACM, New York, NY, USA, 417–432. https://doi.org/10.1145/3035918.3064043

[396] Jeffrey Stuecheli, William J. Starke, John D. Irish, L. Baba Arimilli, Daniel M. Dreps, Bart Blaner, Curt Wollbrink, and Brian Allison. 2018. IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI. *IBM Journal of Research and Development* 62, 4/5 (2018), 8:1–8:8. https://doi.org/10.1147/JRD.2018.2856978

[397] Yifan Sun, Nicolas Bohm Agostini, Shi Dong, and David Kaeli. 2020. Summarizing CPU and GPU design trends with product data. arXiv:1911.11313v2 [cs.DC]

[398] Martin Svedin, Steven Wei Der Chien, Gibson Chikafa, Niclas Jansson, and Artur Podobas. 2021. Benchmarking the Nvidia GPU lineage: From early K80 to modern A100 with asynchronous memory transfers. In *HEART*. ACM, New York, NY, USA, 9:1–9:6. https://doi.org/10.1145/3468044.3468053

[399] Nathan R. Tallent, Nitin A. Gawande, Charles Siegel, Abhinav Vishnu, and Adolfy Hoisie. 2017. Evaluating On-Node GPU Interconnects for Deep Learning Workloads. In *PMBS@SC*. IEEE Computer Society, Los Alamitos, CA, USA, 3–21. https://doi.org/10.1007/978-3-319-72971-8_1

[400] The Linux Kernel Organization 2022. *Shared virtual addressing (SVA) with ENQCMD*. The Linux Kernel Organization. https://www.kernel.org/doc/html/latest/x86/sva.html

[401] Diego G. Tomé, Tim Gubner, Mark Raasveldt, Eyal Rozenberg, and Peter A. Boncz. 2018. Optimizing group-by and aggregation using GPU-CPU co-processing. In *ADMS*. 1–10. http://www.adms-conf.org/2018-camera-ready/tome_groupby.pdf

[402] Top500. 2021. *Top500 Highlights*. Retrieved Jan 28, 2022 from https://www.top500.org/lists/top500/2021/11/highs

[403] Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Clemens Lutz, Martin Schmatz, and Thomas R. Gross. 2015. RStore: A direct-access DRAM-based data store. In *ICDCS*. IEEE Computer Society, Los Alamitos, CA, USA, 674–685. https://doi.org/10.1109/ICDCS.2015.74

[404] William Tsu. 2020. *Introducing Nvidia HGX A100: The most powerful accelerated server platform for AI and high performance computing*. Nvidia. Retrieved Feb 22, 2022 from https://developer.nvidia.com/blog/introducing-hgx-a100-most-powerful-accelerated-server-platform-for-ai-hpc

[405] Erwin L. van Dijk, Yan Jaszczyszyn, Delphine Naquin, and Claude Thermes. 2018. The third revolution in sequencing technology. *Trends in Genetics* 34, 9 (2018), 666–681. https://doi.org/10.1016/j.tig.2018.05.008

[406] Ján Veselý, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. 2016. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *ISPASS*. IEEE Computer Society, Los Alamitos, CA, USA, 161–171. https://doi.org/10.1109/ISPASS.2016.7482091

[407] Sarah A. Vitak, Kristof A. Torkenczy, Jimi L. Rosenkrantz, Andrew J. Fields, Lena Christiansen, Melissa H. Wong, Lucia Carbone, Frank J. Steemers, and Andrew Adey. 2017. Sequencing thousands of single-cell genomes with combinatorial

indexing. *Nature Methods* 14, 3 (2017), 302–308. https://doi.org/10.1038/nmeth.4154

[408] Vasily Volkov. 2016. *Understanding latency hiding on GPUs*. Ph. D. Dissertation. EECS Department, University of California, Berkeley, Berkeley, CA, USA. Advisor(s) Demmel, James W. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html UCB/EECS-2016-143.

[409] Ze-ke Wang, Bingsheng He, and Wei Zhang. 2015. A study of data partitioning on OpenCL-based FPGAs. In *FPL*. IEEE, Washington, DC, USA, 1–8. https://doi.org/10.1109/FPL.2015.7293941

[410] Jan Wassenberg and Peter Sanders. 2011. Engineering a multi-core radix sort. In *Euro-Par (LNCS, Vol. 6853)*. Springer-Verlag, Berlin, Germany, 160–169. https://doi.org/10.1007/978-3-642-23397-5_16

[411] Albert X. Widmer and Peter A. Franaszek. 1983. A DC-balanced, partitioned-block, 8B/10B transmission code. *IBM Journal of Research and Development* 27, 5 (1983), 440–451. https://doi.org/10.1147/rd.275.0440

[412] Fuhui Wu, Qingbo Wu, Yusong Tan, Lifeng Wei, Lisong Shao, and Long Gao. 2013. A vectorized k-means algorithm for Intel Many Integrated Core architecture. In *APPT (LNCS, Vol. 8299)*. Springer Berlin Heidelberg, Heidelberg, Germany, 277–294. https://doi.org/10.1007/978-3-642-45293-2_21

[413] Haicheng Wu, Gregory Frederick Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel Weaver: Automatically fusing database primitives for efficient GPU computation. In *MICRO*. IEEE/ACM, New York, NY, USA, 107–118. https://doi.org/10.1109/MICRO.2012.19

[414] Haicheng Wu, Gregory F. Diamos, Jin Wang, Si Li, and Sudhakar Yalamanchili. 2012. Characterization and transformation of unstructured control flow in bulk synchronous GPU applications. *Int. J. High Perform. Comput. Appl.* 26, 2 (2012), 170–185. https://doi.org/10.1177/1094342011434814

[415] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. Q100: The architecture and design of a database processing unit. In *ASPLOS*. ACM, New York, NY, USA, 255–268. https://doi.org/10.1145/2541940.2541961

[416] Ren Wu, Bin Zhang, and Meichun Hsu. 2009. Clustering billions of data points using GPUs. In *UCHPC-MAW*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/1531666.1531668

[417] Xilinx 2022. *Versal architecture and product data sheet: Overview*. Xilinx. https://www.xilinx.com/support/documentation/data_sheets/ds950-versal-overview.pdf Version 1.15.

[418] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya G. Parameswaran. 2018. Helix: Holistic optimization for accelerating iterative machine learning. *PVLDB* 12, 4 (2018), 446–460. https://doi.org/10.14778/3297753.3297763

[419] Rengan Xu, Frank Han, and Quy Ta. 2018. Deep learning at scale on Nvidia V100 accelerators. In *PMBS@SC*. IEEE Computer Society, Los Alamitos, CA, USA, 23–32. https://doi.org/10.1109/PMBS.2018.8641600

[420] Makoto Yabuta, Anh Nguyen, Shinpei Kato, Masato Edahiro, and Hideyuki Kawashima. 2017. Relational joins on GPUs: A closer look. *IEEE Trans. Parallel Distrib. Syst.* 28, 9 (2017), 2663–2673. https://doi.org/10.1109/TPDS.2017.2677451

[421] Zhaofeng Yan, Yuzhe Lin, Lu Peng, and Weihua Zhang. 2019. Harmonia: A high throughput B+tree for GPUs. In *PPoPP*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/3293883.3295704

[422] Ke Yang, Bingsheng He, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, Pedro V. Sander, and Jiaoying Shi. 2007. In-memory grid files on graphics processors. In *DaMoN*. ACM, New York, NY, USA, 5:1–7:7. https://doi.org/10.1145/1363189.1363196

[423] Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. 2011. Scalable aggregation on multicore processors. In *DaMoN*. ACM, New York, NY, USA, 1–9. https://doi.org/10.1145/1995441.1995442

[424] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The yin and yang of processing data warehousing queries on GPU devices. *PVLDB* 6, 10 (2013), 817–828. https://doi.org/10.14778/2536206.2536210

[425] Chongzhi Zang, Tao Wang, Ke Deng, Bo Li, Sheng'en Hu, Qian Qin, Tengfei Xiao, Shihua Zhang, Clifford A. Meyer, Housheng Hansen He, Myles Brown, Jun S. Liu,

Yang Xie, and X. Shirley Liu. 2016. High-dimensional genomic data bias correction and data integration using MANCIE. *Nature Communications* 7, 1, Article 11305 (April 2016). https://doi.org/10.1038/ncomms11305

[426] Florian Zaruba, Fabian Schuiki, and Luca Benini. 2021. Manticore: A 4096-core RISC-V chiplet architecture for ultra-efficient floating-point computing. *IEEE Micro* 41, 2 (2021), 36–42. https://doi.org/10.1109/MM.2020.3045564

[427] Steffen Zeuch, Sebastian Breß, Tilmann Rabl, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, and Volker Markl. 2019. Analyzing efficient stream processing on modern hardware. *PVLDB* 12, 5 (2019), 516–530. https://doi.org/10.14778/3303753.3303758

[428] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1996. BIRCH: An efficient data clustering method for very large databases. In *SIGMOD*. ACM, New York, NY, USA, 103–114. https://doi.org/10.1145/233269.233324

[429] Yuhao Zhang, Frank Mcquillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano, and Arun Kumar. 2021. Distributed deep learning on data systems: A comparative analysis of approaches. *PVLDB* 14, 10 (2021), 1769–1782. https://doi.org/10.14778/3467861.3467867

[430] Zuyu Zhang, Harshad Deshmukh, and Jignesh M. Patel. 2019. Data partitioning for in-memory systems: Myths, challenges, and opportunities. In *CIDR*. www.cidrdb.org, 1–8. http://cidrdb.org/cidr2019/papers/p133-zhang-cidr19.pdf

[431] Xia Zhao, Almutaz Adileh, Zhibin Yu, Zhiying Wang, Aamer Jaleel, and Lieven Eeckhout. 2019. Adaptive memory-side last-level GPU caching. In *ISCA*. ACM, New York, NY, USA, 411–423. https://doi.org/10.1145/3307650.3322235

[432] Tianhao Zheng, David W. Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. 2016. Towards high performance paged memory for GPUs. In *HPCA*. IEEE Computer Society, Los Alamitos, CA, USA, 345–357. https://doi.org/10.1109/HPCA.2016.7446077

[433] Zion Market Research. 2020. *GPU database market — Global industry analysis*. Retrieved Jan 28, 2022 from https://www.zionmarketresearch.com/report/gpu-database-market

[434] Marcin Zukowski, Sándor Héman, and Peter A. Boncz. 2006. Architecture-conscious hashing. In *DaMoN*. ACM, New York, NY, USA, 6–es. https://doi.org/10.1145/1140402.1140410